To appear in *AAMAS Adaptive Learning Agents Workshop* (ALA 2012), Valencia, Spain, June 2012.

Using Dynamic Rewards to Learn a Fully Holonomic Bipedal Walk

Patrick MacAlpine
Department of Computer Science
The University of Texas at Austin
Austin, TX 78701, USA
patmac@cs.utexas.edu

Peter Stone
Department of Computer Science
The University of Texas at Austin
Austin, TX 78701, USA
pstone@cs.utexas.edu

ABSTRACT

This paper presents the design and learning architecture for a fully holonomic omnidirectional walk used by the UT Austin Villa humanoid robot soccer agent acting in the RoboCup 3D simulation environment. By "fully holonomic" we mean the walk allows for movement in all directions with equal velocity. The walk is based on a double linear inverted pendulum model and was originally designed for the actual physical Nao robot. Parameters for the walk are optimized for maximum speed and stability while at the same time a novel approach of reweighting rewards for walking speeds in the cardinal directions of forwards, backwards, and sideways is utilized to promote equal walking velocities in all directions. A variant of this walk which uses the same walk engine, but is not fully holonomic as it employs three different sets of learned walk parameters biased toward maximizing forward walking speed, was the crucial component in the UT Austin Villa team winning the 2011 RoboCup 3D simulation competition. Detailed experiments reveal that adaptively changing the weights of rewards over time is an effective method for learning a fully holonomic walk. Additional data shows that a team of agents using this learned fully holonomic walk is able to beat other teams, including that of the 2011 RoboCup 3D simulation champion UT Austin Villa team, that utilize non-fully holonomic walks.

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

In this paper, we investigate learning a fully holonomic humanoid walk in the robot soccer domain. By "fully holonomic" we mean a walk that allows for movement in all directions with equal velocity. This is in contrast to the "non-fully holonomic" learned walk used by the 2011 RoboCup[1] 3D simulation league champion team UT Austin Villa. The 2011 UT Austin Villa team's walk employs three different sets of learned walk parameters biased toward maximizing forward walking speed as the kinematics of the simulated robot model inherently allow for walking forwards faster than walking sideways. Although the learned walk of the 2011 UT Austin Villa soccer agent was the key component in the team winning the 3D simulation competition [9], its heavy emphasis on forward walking speed causes in to be significantly slower in other walking directions such as backward and sideways. This lack of speed when not moving forward slows the agent's reaction time and does not allow for quick changes of direction. In order to decrease this delay in changing directions, we would like to learn a set of walk parameters for the walk engine mentioned in Section 3 that allows for equal velocities in all walk directions.

The research reported in this paper is performed within a complex simulation domain, with realistic physics, state noise, multi-dimensional actions, and real-time control. In this test domain, teams of nine autonomous humanoid robots play soccer in a physically realistic environment. Though no simulation perfectly reflects the real world, the physical realism of the RoboCup domain enables pertinent research on realistic robotic tasks such as humanoid locomotion. An important advantage of working in simulation is that extensive experimentation and learning is possible without risk of mechanical wear and tear on any physical device.

As each robot is controlled through low-level commands to its joint motors, getting the robot to walk without falling over is a non-trivial challenge. In this paper, we describe a parameterized omnidirectional walk engine, whose parameters directly affect the speed and stability of the robot's walk, such as step height, length, and timing. Optimal parameters would allow the robot to stably walk as fast as possible in all situations. However, the set of possible walking directions is continuous, so it is infeasible to learn specific parameters for each direction. Therefore, the robot learns a general set of parameters with the goal for it to be able to move equally well in all directions.

The primary contribution of this paper is a methodology for adaptively changing the weights of rewards over time to encourage fast yet close to equal speeds for movement in all directions. There has been some related work in the area of bipedal locomotion such as using dynamic shaping rewards to integrate prior domain knowledge into the learning process for faster walking speeds [8]. Our work differs in that we are not incorporating domain knowledge into the learning process, but are instead trying to learn a walk with two conflicting objectives: fast walking speed and equal walking

[1] http://www.robocup.org/

velocities in all directions. These objectives clash with each other as an increase in speed in one direction often results in a decrease in speed in the perpendicular direction due to the robot's kinematics and joint structure.

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 the non-fully holonomic multiple parameter set walk optimization framework used by the 2011 champion UT Austin Villa agent. In Section 6 we discuss a fully holonomic walk optimization framework using dynamic rewards. Game performance results of agents with different learned walks are given in Section 7, and Section 8 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 [4, 10], to high-level decision-making problems [7, 11].

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 2011 team used an omnidirectional walk engine based on one that was originally designed for the real Nao robot [5]. 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 year's set of unidirectional walks [12].

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. [5], 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 [5], 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

Figure 1: A screenshot of the Nao humanoid robot (left), and a view of the soccer field during a 9 versus 9 game (right).

stance feet.

Notation	Description
maxStep_i^*	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^*	Factors of how fast the step sizes change
y_{sep}	Separation between the feet
x_{offset}^*	Constant offset between the torso and feet
x_{factor}^*	Factor of the step size applied to the forwards position of the torso
err_{norm}^*	Maximum COM error before the steps are slowed
err_{max}^*	Maximum COM error before all velocity reach 0

Table 1: Parameters of the walk engine with the optimized parameters starred.

We now describe the mathematical formulas that calculate the positions of the feet with respect to the torso. More than 40 parameters were used but only the most important ones are described in Table 1. Note that many, but not all of these parameters' values were optimized as described in Section 4.

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 $\text{step}_{i,t+1} = \text{step}_{i,t} + \delta(\text{desired}_{i,t+1} - \text{step}_{i,t}) \forall i \in \{x, y, \theta\}$. 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 $\phi_{start} \leq \phi \leq \phi_{end}$, and $t = \frac{\phi - \phi_{start}}{\phi_{end} - \phi_{start}}$ is the current fraction through the phase. At each time step, ϕ is incremented by $\Delta\text{seconds}/\phi_{length}$, until $\phi \geq \phi_{end}$. At this point, the stance and swing feet change and ϕ is reset to ϕ_{start}. Initially, $\phi_{start} = -0.5$ and $\phi_{end} = 0.5$. However, the start and end times will change to match the previous pendulum, as given by the equations

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

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

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

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

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

If a side velocity is requested, y_{stance} is augmented by

$$\begin{aligned}
y_{frac} &= \begin{cases} 0 & \text{if } t < f_s \\ 0.5(1 + \cos(\pi \frac{t - f_s}{f_m})) & \text{if } f_s \leq t < f_s + f_m \\ 1 & \text{otherwise} \end{cases} \\
\Delta y_{stance} &= \text{step}_y * y_{frac}
\end{aligned}$$

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

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

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

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

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

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

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 [5].

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 80ms. 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

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

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. We refer to the agent that uses this walk as the *Initial* agent.

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 [6], which has been successfully applied previously to a similar problem in [12]. 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 210,000 evaluations in less than a day. 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 14 parameters 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. The 14 optimized parameters are starred in Table 1. Note that maxStep_i represents 3 parameters. Also, while f_g and f_s where chosen to be optimized, their complements f_a and f_m were just set to $(1 - f_g)$ and $(1 - f_m)$ respectively.

5. NON-FULLY HOLONOMIC WALK MULTIPLE SUBTASKS OPTIMIZATION

This section details how a non-fully holonomic multiple parameter set walk was optimized for use in the champion 2011 UT Austin Villa agent. This section serves to give both context and contrast to that of the fully holonomic walk optimization, described in Section 6, which utilizes the goToTarget optimization task presented in Section 5.1. Before describing the procedure for optimizing the walk parameters, we provide some brief context for how the agent's walk is typically used. These details are important for motivating the optimization procedure's fitness functions.

During gameplay the agent is usually either moving to a set target position on the field or dribbling the ball toward the opponent's goal and away from the opposing team's players. Given that an omnidirectional walk engine can move in any direction as well as turn at the same time, the agent has multiple ways in which it can move toward a target. We chose the approach of both moving and turning toward a target at the same time as this allows for both quick reactions (the agent is immediately moving in the desired direction) and speed (where the bipedal robot model is faster when walking forward as opposed to strafing sideways). We validated this design decision by playing our agent against a version of itself which does not turn to face the target it is moving toward, and found our agent that turns won by an average of .7 goals across 100 games. Additionally we played our agent against a version of itself that turns in place until its orientation is such that it is able to move toward its target at maximum forward velocity, and found our agent that immediately starts moving toward its target won by an average of .3 goals across 100 games. All agents we compared used walks optimized by the process described in this section.

Dribbling the ball is a little different in that the agent needs to align behind the ball, without first running into the ball, so that it can walk straight through the ball, moving it in the desired dribble direction. When the agent circles around the ball, it always turns to face the ball so that if an opponent approaches, it can quickly walk forward to move the ball and keep it out of reach of the opponent.

Similarly to a conclusion from [12], we have found that optimization works better when the agent's fitness measure is its performance on tasks that are *executed during a real game*. This stands in contrast to evaluating it on a general task such as the speed walking straight. Therefore, we break the agent's in-game behavior into a set of smaller tasks and sequentially optimize the parameters for each one of these tasks. Videos of the agent performing optimization tasks can be found online.[2]

5.1 Go to Target Parameter Set

In order to simulate common situations encountered in gameplay, the walk engine parameters are optimized for a goToTarget subtask.[3] 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.

$$reward_{target} = d_{target}\frac{t_{total}}{t_{taken}} - Fall \quad (1)$$

$$reward_{stop} = -d_{moved} - Fall \quad (2)$$

The goToTarget optimization includes quick changes of

[2] www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/AustinVilla3DSimulationFiles/2011/html/walk.html

[3] Note that we use three types of notation for each of goToTarget, *GoToTarget*, *goToTarget*, to distinguish between an optimization task, an agent created by this optimization task and a parameter set. Similarly for "sprint" and "initial".

- Long walks forward/backwards/left/right
- Walk in a curve
- Quick direction changes
- Stop and go forward/backwards/left/right
- Switch between moving left-to-right and right-to-left
- Quick changes of target to simulate a noisy target
- Weave back and forth at 45 degree angles
- Extreme changes of direction to check for stability
- Quick movements combined with stopping
- Quick alternating between walking left and right
- Spiral walk both clockwise and counter-clockwise

Figure 2: GoToTarget Optimization walk trajectories

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. After running this optimization seeded with the initial walk engine parameter values we saw a significant improvement in performance. Using the parameter set optimized for going to a target, the *GoToTarget* agent was able to beat the *Initial* agent by an average of 8.82 goals with a standard error of .11 across 100 games.

5.2 Sprint Parameter Set

To further improve the forward speed of the agent, we optimized a parameter set for walking straight forwards for ten seconds starting from a complete stop. The robot was able to learn parameters for walking .78 m/s compared to .64 m/s using the *goToTarget* parameter set. Unfortunately, when the robot tried to switch between the forward walk and *goToTarget* parameter sets it was unstable and usually fell over. This instability is due to the parameter sets being learned in isolation, resulting in them being incompatible.

To overcome this incompatibility, we ran the goToTarget subtask optimization again, but this time we fixed the *goToTarget* parameter set and learned a new parameter set. We call these parameters the *sprint* parameter set, and the agent uses them when its orientation is within 15° of its target. The *sprint* parameter set was seeded with the values from the *goToTarget* parameter set. By learning the *sprint* parameter set in conjunction with the *goToTarget* parameter set, the new *Sprint* agent was stable switching between the two parameter sets, and its speed was increased to .71 m/s. Adding the *sprint* parameter set also improved the game performance of the agent slightly; over 100 games, the *Sprint* agent was able to beat the *GoToTarget* agent by an average of .09 goals with a standard error of .07.

5.3 Positioning Parameter Set

Although adding the *goToTarget* and *sprint* walk engine parameter sets improved the stability, speed, and game performance of the agent, the agent was still a little slow when positioning to dribble the ball. This slowness is explained by the fact that the goToTarget subtask optimization emphasizes quick turns and forward walking speed while positioning around the ball involves more side-stepping to circle the ball. To account for this discrepancy, the agent learned a third parameter set which we call the *positioning* parameter set. To learn this set, we created a driveBallToGoal2[4] optimization in which the agent is evaluated on how far it is able to dribble the ball over 15 seconds when starting from a variety of positions and orientations from the ball. The *positioning* parameter set is used when the agent is .8 meters from the ball and is seeded with the values from the *goToTarget* parameter set. Both the *goToTarget* and *sprint* parameter sets are fixed and the optimization naturally includes transitions between all three parameter sets, which constrained them to be compatible with each other. Adding the *positioning* parameter set further improved the agent's performance such that it, our *Final* agent, was able to beat the *Sprint* agent by an average of .15 goals with a standard error of .07 across 100 games. A summary of the progression in optimizing the three different walk parameter sets can be seen in Figure 3.

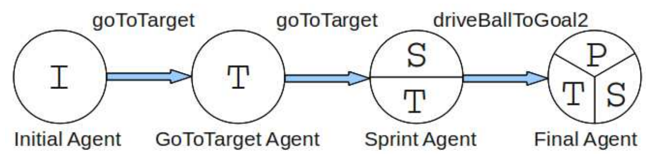

Figure 3: UT Austin Villa walk parameter optimization progression. Circles represent the set(s) of parameters used by each agent during the optimization progression while the arrows and associated labels above them indicate the optimization tasks used in learning. Parameter sets are the following: I = *initial*, T = *goToTarget*, S = *sprint*, P = *positioning*.

6. FULLY HOLONOMIC WALK OPTIMIZATION

One weakness of the non-fully holonomic walk learned in Section 5 is that the optimization process's heavy emphasis on forward walking speed results in significantly slower speeds in other walking directions such as backward and sideways. While this is somewhat mitigated by having an agent always turn to face the direction it is moving, and thereby quickly switch to walking at full speed in the forward direction, the turning process slows down the agent and does not allow for quick changes of direction.

In order to decrease this delay in changing directions we would like to learn a set of walk parameters for the walk engine mentioned in Section 3 that allows for equal velocities in all walk directions. With such a fully holonomic walk there will be no need, and resulting delay, for the robot to change its orientation as it changes direction. As the kinematics of the simulated robot model inherently allow for walking forwards faster than walking sideways, attempting to maximize walking speeds in all direction is likely to learn a walk engine parameter set biased toward a faster forward walk at the expense of slower speed in the sideways direction.

[4]The '2' at the end of the name driveBallToGoal2 is used to differentiate it from a driveBallToGoal optimization that was used in [12].

To account for the potential of the speed for any direction of the walk to dominate over the speed of other directions during the optimization process, we propose individually tallying the amount of reward given to walking in the three cardinal directions of forwards, backwards, and sideways, and then reweighting the rewards accumulated for these directions during the next iteration of CMA-ES so as to give more influence to directions in which the agent is walking slower. We implemented this idea by modifying the goToTarget optimization task mentioned in Section 5.1 to only count the positive reward from Equation 1 for three different walk targets that are part of the walk trajectories of the first item in Figure 2: long walks in the forward, backward, and sideways directions with each walk having a duration of 7 seconds. The positive rewards for these three walk targets are then all multiplied by weights whose values we adjust across iterations of CMA-ES, but whose sum is always normalized to 1, to calculate the overall positive portion of the fitness given to an agent (shown in Equation 4 for iteration i). During the first iteration of CMA-ES the values of these weights are all initialized to be 1/3 (Equation 3).

$$wgt_{i=1\{fw/bw/sw\}} = 1/3 \tag{3}$$

$$fit_{i\{positive\}} = \begin{array}{l} rew_{i\{fw\}} * wgt_{i\{fw\}} \\ +rew_{i\{bw\}} * wgt_{i\{bw\}} \\ +rew_{i\{sw\}} * wgt_{i\{sw\}} \end{array} \tag{4}$$

After every iteration of CMA-ES the three cardinal direction walk rewards (speeds) of the member of the population with the highest fitness are examined and used to calculate new reward weights for the next iteration of CMA-ES based on the following equations:

$$rew_{i\{fw/bw/sw\}} = max(rew_{i\{fw/bw/sw\}}, .1) \tag{5}$$

$$rew_{i\{max\}} = max(rew_{i\{fw,bw,sw\}})$$

$$wgt_{i+1\{fw/bw/sw\}} = rew_{i\{max\}}/rew_{i\{fw/bw/sw\}} \tag{6}$$

$$wgt_{i+1\{tot\}} = sum(wgt_{i+1\{fw,bw,sw\}})$$

$$wgt_{i+1\{fw/bw/sw\}} = wgt_{i+1\{fw/bw/sw\}}/wgt_{i+1\{tot\}} \tag{7}$$

Equation 5 first ensures that all direction rewards are positive which is necessary for the computation of reward weights. Reward weights are computed in Equation 6 and are equal to the factor that a directional reward needs to be multiplied by in order to be equal to that of the maximum directional reward. Finally Equation 7 normalizes all reward weights to sum to 1. We refer to the agent that uses this formulation for updating reward weights as the *DynamicRewards* agent.

Note that in addition to the positive portion of the fitness for an agent computed in Equation 4, the agent also still receives negative rewards for falling and movement when told to stop for all walk targets as described in Equations 1 and 2 in Section 5.1. This is done to ensure that the agent learns a walk that can quickly stop and is stable in the same way as the walk produced by the *goToTarget* parameter set in Section 5.1. As the agent no longer receives positive rewards for moving to all targets as in the original goToTarget optimization, and instead only receives a positive reward for moving to targets for a weighted total of 7 seconds, we need to reduce the value of the negative rewards so as to preserve the ratio between possible attainable positive and negative rewards present in the original goToTarget optimization. We found that not reducing the value of negative rewards causes the agent to learn walking parameters that keep it stationary. This is because the negative rewards incurred by moving when told to stop, and also potentially falling, dominate the possible rewards gained in just 7 seconds of measured positive movement toward walk targets. Our solution to this was to weight negative rewards by the ratio of current time moving to targets for which positive rewards are recorded to that of the same time as in the original goToTarget optimization (7/124.1).

An alternative to calculating new reward weights using the directional rewards of the member of the population with the highest fitness is to instead use a weighted average of the directional rewards of the top half of the population with the highest fitness. This weighted averaging is what CMA-ES does at the end of every iteration to update the mean of the multivariate Gaussian distribution it is sampling parameters from. This weighted averaging is computed by the following equations for which members of a population are first ranked and sorted in descending order of fitness ($i = 1$ for highest fitness member):

$$weight_i = log(popsize/2 + 1/2) - log(i)$$

$$weights_{sum} = \sum_{i=1}^{popsize/2} weight_i$$

$$weight_i = weight_i/weights_{sum}$$

$$rew_{avg\{fw/bw/sw\}} = \sum_{i=1}^{popsize/2} rew_{i\{fw/bw/sw\}} * weight_i$$

We call the agent that uses weighted averages of distance rewards to update reward weights the *DynamicAvgRewards* agent.

In addition to the two agents that change the weight of rewards over time (the *DynamicRewards* and *DynamicAvgRewards* agents), for comparison purposes we also ran optimizations on the modified goToTarget task for a couple of agents that do not modify rewards. The first of these is the *StaticRewards* agent which is optimized in the same way as the *DynamicRewards* agent except that it holds each of the directional reward weights constant at 1/3. Unlike the *Final* agent used by UT Austin Villa in the 2011 competition, which controls its orientation differently for different tasks as described in Section 5, the *DynamicRewards*, *DynamicAvgRewards*, and *StaticRewards* agents all directly move in the direction of their targets without purposely modifying their orientations in any way. The second agent which does not modify directional reward weights is the *FaceForward* agent. This agent always turns to face whatever target it is moving to and is similar to the the *GoToTarget* agent except that it was optimized using the modified version of the goToTarget task. All optimizations were done using CMA-ES with a population size of 150 across 200 generations.

Figure 5 shows how the directional weights for the *DynamicRewards*, *DynamicAvgRewards*, and *StaticRewards* agents vary across iterations of CMA-ES. Note that the weights used by the *StaticRewards* agent are fixed and that the weights shown in this figure are only computed and displayed for comparison purposes to show what the weights chosen would be if the agent was dynamically adjusting its weights. The *DynamicRewards* and *DynamicAvgRewards* agents' weights are very close together with a slightly higher weight for the forward direction (meaning that the forward direction is producing a slightly lower reward than the other directions). The fluctuation in weights is a little smoother

Table 2: Directional walking speeds of learned walks for different agents described in Section 6 as well as the *Final* agent used by UT Austin Villa in the 2011 RoboCup competition. All speeds are in m/s and were measured by recording the distance the agent traveled in ten seconds when starting from a complete stop.

Agent	Forward	Backward	Sideways
DynamicRewards	.42	.53	.48
DynamicAvgRewards	.45	.53	.51
StaticRewards	.58	.52	.37
FaceForward	.74	.35	.03
Final	.71	.40	.21

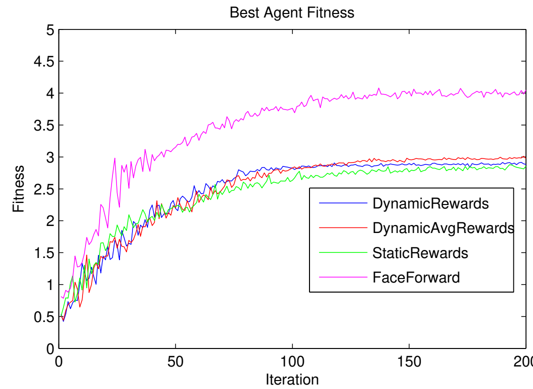

Figure 4: Best agent fitness across iterations of CMA-ES.

for the *DynamicAvgRewards* agent than that of the *DynamicRewards* agent as it is using averaging. The *StaticRewards* agents shows a considerable difference in weights, however, with a much higher weight (and thus lower reward) coming from the sideways direction. This is an indicator of the optimization finding it easier to optimize for speed in the forward direction than that of the sideways direction.

In Figure 4 the best agent fitnesses across iterations of CMA-ES are shown for the agents described in this section. Here we see that all agents' fitnesses start to plateau around iteration 100. The static and dynamic rewards agents all have similar fitnesses while the *FaceForward* agent has a higher fitness. The probable cause for this is because the lengthy long walks of 7 seconds in each direction allow the *FaceForward* agent plenty of time to turn and walk in the forward direction for which it is being optimized for.

Directional walking speeds of the different agents described in this section, as well as the *Final* agent used by UT Austin Villa in the 2011 RoboCup competition, and described in Section 5, are shown in Table 2. Here we see fairly close values across all directions for both the dynamic rewards agents. This gives proof that either method of adjusting the weights of rewards across iterations is a viable solution for learning a walk that is close to fully holonomic. The *StaticRewards* agent has a forward walk speed over 50% faster than it's side walk speed which shows evidence of the forward walk speed being easier to increase at the expense of the sideways walk speed. The *FaceForward* agent has the highest speed for the forward walk which is not surprising as this is the direction it is walking in for most of the time during optimization. Despite never walking backwards during optimization the agent still has a backward speed close to half its forward speed. This suggests a correlation in movement between walking forward and backward as they both lie in the sagittal plane. Walking sideways (movement in the opposite coronal plane) on the other hand never occurs during optimization which results in a speed in this direction of nearly 0. The *Final* agent from the 2011 competition has very good forward speed similar to the *FaceForward* agent, which it was generally optimized for, but also has much better sideways speed than the *FaceForward* agent due to using multiple learned parameter sets including the *positioning* parameter set for which sideways movement was included as part of the optimization.

7. AGENT GAME PERFORMANCE RESULTS

In Table 3 we present game results of the agents with different walk parameter sets described in Section 6 as well as the *Final* agent used in the 2011 competition and described in Section 5. Both the *StaticRewards* and *FaceForward* agents do very poorly against the other agents. Empirical evidence shows that the *StaticRewards* agent is too slow when trying to move sideways, which happens roughly half the time, allowing other agents to easily get around or steal the ball from the agent. The *FaceForward* agent, on the other hand, gets mired down in constantly turning and trying to adjust its position when around the ball as it is unable to move sideways.

The *DynamicRewards* and *DynamicAvgRewards* agents perform very similarly which isn't surprising considering how close their walk speeds are in Table 2. What is surprising is that they are both able to beat the champion *Final* agent from the 2011 competition which uses three learned walk parameter sets instead of just one. Despite the *Final* agent having a significantly faster top walking speed than both of the dynamic reward agents, the dynamic reward agents are much faster positioning around the ball due to the *Final* agents slow sideways speed and need for turning to adjust its orientation while circling the ball. While the average goal difference that the *DynamicRewards* agent beat the *Final* agent by across 100 games was only .20 goals, this still translated to 29 goals for with 9 against and a record of 23 wins with only 7 losses and 70 ties.

8. SUMMARY AND DISCUSSION

We have presented the design and learning architecture for a fully holonomic omnidirectional walk used by the UT Austin Villa humanoid robot soccer agent acting in the RoboCup 3D simulation environment. The key to our optimization method is using a novel approach of reweighting rewards for walking speeds in the cardinal directions of forwards, backwards, and sideways to promote equal walking velocities in all directions. A team of agents using this learned fully holonomic walk, which consists of just a single learned parameter set, is able to beat the UT Austin Villa 2011 RoboCup 3D simulation champion team that uses a

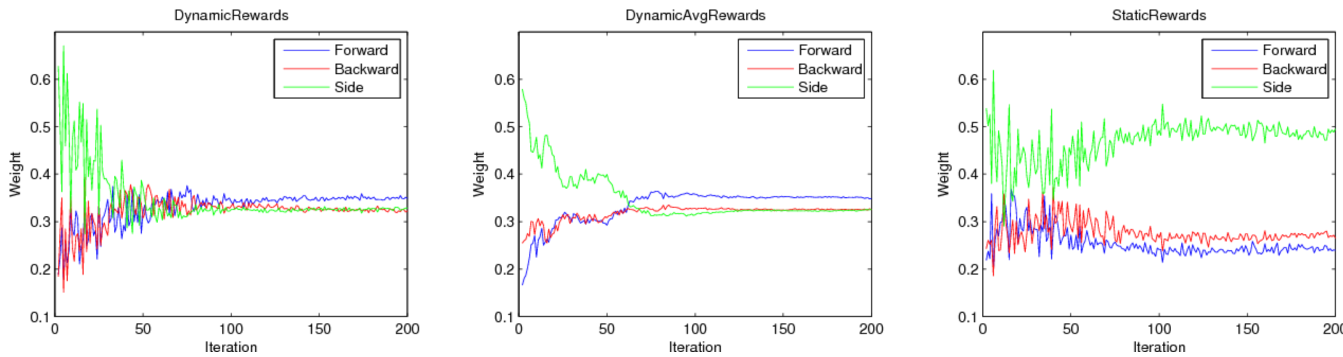

Figure 5: Directional reward weights during the course of optimization for the the *DynamicRewards* agent (left), *DynamicAvgRewards* agent (center), and *StaticRewards* agent (right). Note that the weights used by the *StaticRewards* agents are fixed and that the weights shown in this figure are only computed and displayed for comparison purposes to show what the weights chosen would be if the agent was dynamically adjusting its weights. Higher weights correlate to lower relative rewards.

Table 3: Game results of agents with different walk parameter sets described in Section 6 as well as the *Final* agent used in the 2011 competition and described in Section 5. Entries show the average goal difference (row − column) from 100 ten minute games. Values in parentheses are the standard error.

	Final	FaceForward	StaticRewards	DynamicAvgRewards
DynamicRewards	0.20(.08)	3.27(.09)	3.18(.11)	-0.06(.07)
DynamicAvgRewards	0.10(.07)	3.49(.11)	2.88(.11)	
StaticRewards	-2.77(.13)	0.22(.06)		
FaceForward	-2.99(.12)			

non-fully holonomic walk employing multiple walk parameter sets. This is a significant accomplishment as the 2011 UT Austin Villa team won all 24 games it played during the RoboCup competition while scoring 136 goals and conceding none.

Our ongoing research agenda includes applying what we have learned in simulation to the actual Nao robots which we use to compete in the Standard Platform league of RoboCup. Additionally, we would like to learn multiple parameter sets for the fully holonomic walk, specialized for walking in each of the different cardinal directions, in a similar fashion to how the *sprint* parameter set was learned in Section 5.2.

More information about the UT Austin Villa team, as well as video highlights from the 2011 competition, can be found online at the team's website.[5]

Acknowledgments

This work has taken place in the Learning Agents Research Group (LARG) at UT Austin. Thanks especially to UT Austin Villa 2011 team members Daniel Urieli, Samuel Barrett, Shivaram Kalyanakrishnan, Michael Quinlan, Francisco Barrera, Nick Collins, Adrian Lopez-Mobilia, Art Richards, Nicolae Ştiurcă, and Victor Vu. Also thanks to Yinon Bentor and Suyog Dutt Jain for contributions to early versions of the optimization framework employed by the team. 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.

9. 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] 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.

[5] 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.

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

[7] 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.

[8] A. Laud and G. Dejong. Reinforcement learning and shaping: Encouraging intended behaviors. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 355–362. Morgan Kaufmann, 2002.

[9] P. MacAlpine, D. Urieli, S. Barrett, S. Kalyanakrishnan, F. Barrera, A. Lopez-Mobilia, N. Ştiurcă, V. Vu, and P. Stone. UT Austin Villa 2011: A champion agent in the RoboCup 3D soccer simulation competition. In *Proc. of 11th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, June 2012.

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

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

[12] 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.

[5] `www.cs.utexas.edu/~AustinVilla/sim/3dsimulation/`