Abstract

RoboCup 3D simulated soccer presents many challenges to reinforcement learning methods, including a large state space, hidden and uncertain state, multiple independent agents learning simultaneously, and long and variable delays in the effects of actions. In keepaway, one team, “the keepers”, tries to keep control of the ball for as long as possible despite the efforts of “the takers”. In this project we explore the optimization methods, architecture and design of various high level macro actions required for keepaway in robocup 3D environment.

1 Introduction

Reinforcement learning (Sutton and Barto, 1998) is a theoretically grounded machine learning technique intended to enable a self-governing agent to maximize its long term reward by means of rehashed experimentation in, and interaction with, its environment. RoboCup simulated soccer has been used as the basis for successful international competitions and research challenges (Kitano et al., 1997). As presented in detail by Stone (2000), it is a fully distributed, multi-agent domain with both teammates and adversaries.

In robocup simulated soccer, there is hidden state, meaning that each agent has only a partial world view at any given moment. The agents also have noisy sensors and actuators, meaning that they do not perceive the world exactly as it is, nor can they affect the world exactly as intended. In addition, the perception and action cycles are asynchronous, prohibiting the traditional AI paradigm of using perceptual input to trigger actions. Communication opportunities are limited, and the agents must make their decisions in real-time. These domain characteristics combine to make simulated robot soccer a realistic and challenging domain.

As part of this project we intend to develop learning strategies and basic skills required for keepaway in Robocup 3D simulated soccer environment. In keepaway, one team, the keepers, tries to maintain possession of the ball within a limited region, while the opposing team, the takers, attempts to gain possession. Whenever the takers take possession or the ball leaves the region, the episode ends and the players are reset for another episode (with the keepers being given possession of the ball again)

Some literature [5] is already present on high level decision making strategies for keepaway in Robocup 2D simulated soccer environment, but all attempts to replay them in 3D environment are rudimentary. The main reason behind this failure is that a 2D dot is too simplistic model of a real humanoid robot and even a task like kick (which is fairly simple in 2D environment) in 3D environment demands the agent (which has a partially observed probabilistic model of the world) to get in the right orientation and select the correct sequence of torques applied across its 22 hinges.

Most of the work done in this project is on developing skills and behaviours required for keepaway in 3D Robocup setup. We have built upon UT Austin’s 2011 base code [3] which includes basic machinery required for a robocup agent such as:

- Omnidirectional walk engine based on a double inverted pendulum model
- A couple basic skills for kicking one of which uses inverse kinematics
• Particle filter for localization and Kalman filter for tracking objects
• All necessary parsing code for sending/receiving messages from/to the server

2 Domain Description

The RoboCup 3D simulation environment is based on SimSpark, a generic physical multiagent system simulator. SimSpark uses the Open Dynamics Engine (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, which has a height of about 57 cm, and a mass of 4.5 kg.

![Simulated Nao Robot](image)

Figure 1: Simulated Nao Robot

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.

3 Related Work

3.1 2D KeepAway

In [5] Stone, et. al. describe learning algorithm for high level decision making in 2D keepaway. They mapped keepaway onto discrete time, episodic, semi-Markov decision process (SMDP). Each player learns independently and may perceive the world differently. For each player, an episode begins when the player is first asked to make a decision and ends when possession of the ball is lost by the keepers. The learner’s choose actions from following higher level macro actions:

- **HoldBall()**: Remain stationary while keeping possession of the ball
- **PassBall(k)**: Kick the ball directly towards keeper k
• GetOpen() : Move to a position that is free from opponents and open for a pass from the ball’s current position
• GoToBall() : Intercept a moving ball or move directly towards a stationary ball
• BlockPass(k): Move to a position between the keeper with the ball and keeper k

On this SMDP, they use the SMDP version of the Sarsa(λ) algorithm with linear tile-coding function approximation (also known as CMACs) and replacing eligibility traces. Each player learns simultaneously and independently from its own actions and its own perception of the state.

3.2 Optimization of skills in 3D environment

[1] and [4] discusses algorithms and their aspects related to optimization of skills in Robocup 3D environment. [4] presents the design and learning architecture for an omnidirectional walk used by a humanoid robot soccer agent acting in the RoboCup 3D simulation environment. They optimized the walk parameters using the Covariance Matrix Adaptation Evolution Strategy (CMAES) algorithm [2].

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.

4 Technical Section

The macro actions used in [5] are easy to implement in 2D robocup, since in 2D robocup, agents can execute a parameterized primitive action such as turn(angle), dash(power), or kick(power, angle). None of these actions are present in 3D robocup, the basic actions that an agent can command the server to do is apply specified amount of torque on specified hinge. This makes every high level action (which was taken for granted in 2D setup) very complex in 3D setup. Some of these behaviours are present in UT Austin’s 2011 base code but need optimization. Our work involves optimization of already present skills, development of new skills and framework for high level behaviours required for 3D keepaway.

4.1 Optimization of Kick Skill

Having a kick skill which can efficiently kick ball in right direction and to right distance is very important for keepaway. Kick skill using inverse kinematics is already present in UT Austin’s base code [3] but it is not very robust or accurate in different situations. This motivated us to work on optimization of kick skill. Figure [2] illustrates the program flow of UT Austin’s kick engine.

We ran CMA-ES [2] on control points of the kick trajectory. Each episode consist of agent being born at some random point inside a fixed region. The agent then attempts to kick the ball with the candidate parameters. The episode ends when the kick action is executed and ball has stopped or when the agent was unable to execute kick action (probably because agent fell or kick parameters were invalid). Each candidate parameter is evaluated over 5 episodes each for 5 different target points. For each episode fitness function is calculated as following :

• if agent was not able to execute kick : totalFitness += 1000
• else :
  – totalFitness += angle(target, ballStart, ballEnd) * 5
  – totalFitness += (dis(target, ballStart) − dis(ballEnd, ballStart)) * 20

This type of fitness function gives more emphasis on the angle made between the target, ball’s starting position and ball’s ending position than the difference between target and ball’s ending position, this is because, when passing the ball we assume that a agent will be present there to receive the
Figure 2: The flow of the agent deciding when to kick the ball and how to interpolate the curve created relative to the ball

ball hence direction is more important. Also, in our experimentation we found that it was hard for a particular parameter to generalize over different target distances. Thus, now we plan to have set of 5 parameters, each optimized for different target distances. Figure 3 shows the trend of minimum fitness over generations.

Figure 3: Minimum Fitness over generations

4.2 Receive Skill

A good receive skill is also very important in keepaway, if the receiving agent is not able to receive the ball properly then it would waste lot of time in positioning itself for the kick. Since there was no receive skill present in UT Austin’s base code, we developed our own receive skill. Receive skill consist of following three states:

- Get in the position of freezing ball
- Maintain above stance till ball makes contact
• Go back to normal position without touching the ball

Initially we hardcoded the stances as sequence of fixed frames and then optimized the parameters involved using CMA-ES, similar to what was done in 4.1. Figure 4 illustrates the three states of receive skill.

![Receive skill states](image)

Figure 4: Receive skill states

### 4.3 Action Skill Framework

Each high level behaviour is encoded as state transition machine. A transition from one state to another state happens only when certain conditions are satisfied. In our naive implementation of these high level behaviours on passing task, in which two agents pass ball to and fro indefinitely, we observed that many times the transitions weren’t happening as smoothly as desired. This led to agent falling many times during the task. This motivated us to work on Action Skill Framework which sets some guidelines and regulations for every skill. We established a hierarchy of action skills, namely:

- **Low level Skills**: these define low level skills such as freeze ball, turn, kick, etc
- **Intermediate level Skills**: these define actions which use low level skills to achieve a complex task such as intercept ball
- **High level Skills**: these define roles or high level behaviours of an agent, these skills use only intermediate skills

Each skill from any of the 3 sets, is derived from base class ActionSkill which forces all skills follow certain guidelines and regulations. Now each skill must implement following functions:

- **invoke()**: this must be called prior to invocation of any skill, this prepares the skill
- **abort()**: this must be called before aborting a particular skill, this ensures that the a skill doesn’t end up in bad state
- **getSkill()**: this function return the actual action to take, in order to execute the skill

This hierarchical division helped in optimization process also. Now we could optimize lower level skills first with more direct fitness function and then optimize the intermediate and high level skills with more ambitious fitness function.

## 5 Conclusion

In this project we explored and discussed the optimization methods, architecture and design of various high level macro actions required for keepaway in robocup 3D environment. Although, we have achieved some success regarding establishing a framework, developing new skills and optimizing existing skills but in order to replay keepaway in 3D there are still some things that has to be done such as optimization of the control conditions (the conditions which define the transitions) of skills.

### References


