Learning complex behaviours and Keepaway in 3D Robocup Environment

Nilesh Gupta[1] and Shivaram Kalyanakrishnan[1]

Department of Computer Science, Indian Institute of Technology Bombay
{nilesh,shivaram}@cse.iitb.ac.in

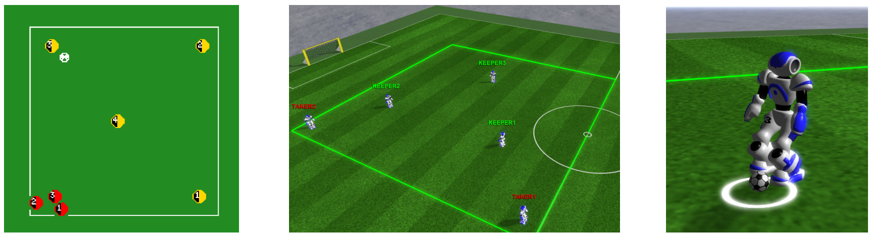
Abstract. In keepaway, one team, the keepers, tries to keep control of the ball for as long as possible despite the efforts of the takers. Keepaway is an important subtask for simulated soccer but yet no significant prior work has been done on Keepaway in 3D Robocup environment. We describe our application of evolutionary algorithms to learn control of complex behaviours in 3D Robocup environment and use it to learn the behaviour of getting possession of ball, which is essential for keepaway. Later, we formulate the learning of higher-level decision policy for keepers in 3 vs 2 keepaway as NEAT (NeuroEvolution of Augmenting Topologies) optimisation task. Our learned policies significantly outperform the hand coded policies.

Keywords: Evolutionary Algorithm · NEAT · Keepaway

1 Introduction

Robocup 3D simulated soccer is used to accurately simulate humanoid agents and physics behind them for international competitions and research challenges. It is a multi agent, fully distributed domain with teammates and adversaries. Main challenges presented by this domain is it's large state space, hidden and uncertain state, multiple independent agents learning simultaneously, and long and variable delays in the effects of actions. Our focus in this article is mainly on a subtask of Robocup 3D simulated soccer, namely keepaway.

In keepaway, a team of keepers tries to maintain the possession of ball as long as possible in presence of a team of adversaries, the takers. Multiple Reinforcement Learning approaches [6] and Evolutionary Learning approaches have been shown to be successful in playing keepaway in Robocup 2D simulator soccer environment, but no significant work has been done to successfully play keepaway in Robocup 3D simulator soccer domain. This is mainly because of the huge complexity of the 3D domain compared to the 2D domain. In 2D domain an agent is represented as a circular rigid body and has access to many convenient parameterized primitive actions such as turn(*angle*), dash(*power*), or kick(*power, angle*) while in the 3D domain an agent is represented as an exact model of an actual humanoid robot (see Fig. 1c) and is exposed to primitive actions which involve applying specified amount of torque to its hinges. Even a 2D domain's primitive action like dash or kick is extremely complex in 3D domain, since to achieve a dash or kick the agent has to figure out correct sequence of torque values to apply across all it's 22 hinges over different timesteps.



(a) 2D Keepaway (b) 3D Keepaway (c) Nao Agent

Fig. 1: Comparison between 2D keepaway and 3D keepaway

Luckily we didn't had to start from scratch as lot of work in developing skills for kick, walk, etc has already been done. All the work reported in this article is built over UT Austin's 2011 base code which is based on UT austin's 2011 RoboCup 3D simulation league team [4]. UT Austin's base code provides basic functionalities for playing soccer in Robocup 3D soccer simulator (such as, omnidirectional walk engine for walking, kicking skills using inverse kinematics, World model and particle filter for localization, etc) but these basic skills have to be optimised as per requirement. So, we treat the optimised version of basic skills (turn(*angle*), move(*pos*) and kick(*pos*)) provided by UT Austin's base code as primitive action. Even after assuming that such primitive actions have been given to us, defining complex behaviours in 3D domain such as get possession of the ball is hard to define because these actions in 3D domain won't be responsive enough to make quick adjustments based on state changes. For example, a simple strategy for intercepting a moving ball of just moving to ball's current location would result in agent reflecting the ball most of the time. Thus, a complex behaviour requires the agent to select the right combination of primitive actions to achieve the goal. After we have defined and learned the complex behaviours, the agent has to learn its high level decision making policies i.e. where to kick the ball (after getting possession of the ball) and how do other keepers position themselves.

Our article is articulated in a similar flow tackling each problem at a time. In section 3, we discuss optimisation of the primitive skills required for keepaway and present methods using CMA-ES which has had previous success in these optimisations. In section 4, we give a general framework for defining complex behaviours and show how the control can be modelled as a feed forward neural network. After we have modelled the control as a neural network we can optimise it using evolutionary algorithms like NEAT. We also give method to initialize the

NEAT optimisation process with good seed to minimize the training time and motivate the optimisation towards the candidates that we expect to be good. We later use this framework to learn complex behaviour of getting possession of the ball. In section 5, we define behaviours of keepers and takers and map the keepaway task onto learning evaluation function for different candidate targets. We use NEAT to learn this evaluation function.

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.

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.

Keepaway is played between a team of takers and keepers inside a confined boundary. For our experiments we choose boundary of dimension $10m$ x $10m$. An episode of 3 vs 2 keepaway starts with the two takers being born at adjacent corners of the boundary and keepers being born at vertices of an equilateral triangle centered around the center of the playing area. To give keepers some initial advantage ball is initially placed very near to a keeper. The episode ends when the distance between any taker and ball becomes less than a threshold (0.2) or any keeper or ball goes out of the boundary.

3 Kick optimisation and Ball Anticipation

3.1 Kick optimisation

All the work reported in this article is built over UT Austin's 2011 base code. Although UT Austin's base code provides basic functionalities for playing soccer in Robocup 3D soccer simulator (such as, omnidirectional walk engine for walking, kicking skills using inverse kinematics, World model and particle filter for localization, etc), but these skills need to be optimised as per requirement. For playing keepaway, a robust, precise and mid-range (since keepaway is played within a confined boundary) kick is required. Also, the approach to kick parameters need to be optimised so as to get fast executions of kick.

We combined all the kick parameters and approach to kick parameters and try to optimise them so as to get best candidate parameters as per the fitness function defined below, which is combination of following parameters :

- $time\_factor = episode\_end\_time - episode\_start\_time$
- $angle\_factor = 2^{-(angle(ball\_finish, ball\_start, target)^2/180.0)}$
- $distance\_factor = max(distance(ball\_start, ball\_finish), 6.0)$

$$episode\_fitness = \begin{cases} -1 & \text{Failure} \\ distance\_factor * angle\_factor/time\_factor & \text{Otherwise} \end{cases} \tag{1}$$

Each candidate parameter is evaluated over 12 episodes and the fitness for the candidate is determined by summing over each episode's fitness. Each episode starts with ball coming from center of the field with some initial velocity and then the agent attempts to kick the ball to a target. The initial velocity and the target are fixed differently for different episodes, so that we don't over fit for a particular kind of scenario.

For optimisation we use Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) [2]. In CMA-ES, new candidate solutions are sampled according to a multivariate normal distribution in $\mathbb{R}^n$. Recombination amounts to selecting a new mean value for the distribution. Mutation amounts to adding a random vector, a perturbation with zero mean. Pairwise dependencies between the variables in the distribution are represented by a covariance matrix. The covariance matrix adaptation (CMA) is a method used to update the covariance matrix of this distribution. CMA-ES has had previous success for optimising walk and kick in Robocup 3D domain [1].

3.2 Ball Anticipation

For playing keepaway in Robocup 3D environment, planning ahead in future is very important since the agents are not agile enough to respond quickly to state

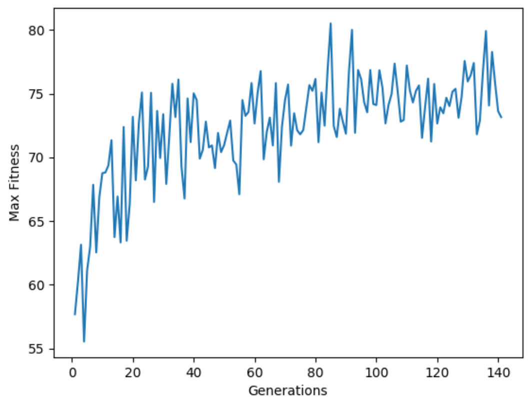


Fig. 2: CMA-ES kick optimisation curve, max fitness vs generation

changes. One important prediction that is required for planning ahead is prediction of ball's stopping position. The challenges in predicting ball's stopping position is that the agent gets to know noisy readings of ball's positions and that too at intervals of 3 time steps of simulation.

Let at time $t = t_0$ ball starts moving and at time $t = t_n$ ball halts. Let $t = t_1, t_2, ..., t_{n-1}$ be the timestamps in between t_0 and t_n. Let the ball position at time $t = t_i$ be $ballpos_i$ and the position where ball halts be $ballfinish^*$ i.e $ballfinish^* = ballpos_n$. For $i = \{1, .., n\}$, define

$$ballvel_i = \frac{(ballpos_i - ballpos_{i-1})}{(t_i - t_{i-1})}$$

$$\overline{ballfinish_i} = \alpha * ballvel_i$$

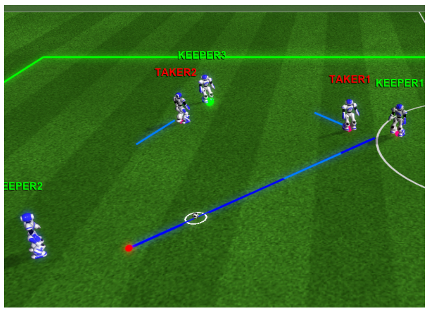
Now at each timestep t_i we make estimate $ballf\hat{i}nish_i$ for $ballfinish^*$ as,

$$ballf\hat{i}nish_i = \frac{\sum_{j=1}^{j=i}(\lambda^{i-j} * \overline{ballfinish_i})}{\sum_{j=1}^{j=i}\lambda^{i-j}}$$

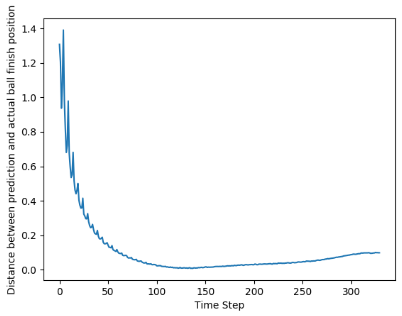
Essentially what we are trying to do here is that at each timestep we make a linear in $ballvel_i$ estimate of $ballfinish^*$. But since a linear estimate might not generalize well over all sample points and we observe noisy readings of $ballpos_i$, we take the running average over all the linear estimates $\overline{ballfinish_i}$ and give that as a better estimate for $ballfinish^*$. α and γ were later tuned to minimize following loss function :

$$loss = \sum_{i=1}^{i=M}\sum_{j=1}^{j=n_i} distance(ballf\hat{i}nish_{j,i}, ballfinish_i^*)^2$$

Here the first summation indicates that the loss is calculated over M trajectories. The predictions obtained after few timestamps are very near to $ballfinish^*$. These estimates are later used as feature for both learning behaviour of getting possession of ball and defining behaviour of keepers in keepaway.



(a)



(b)

Fig. 3: (a) Screenshot of ball's predicted trajectory and $ball\hat{finish}_t$, (b) distance($ball\hat{finish}_t, ballfinish^*$) vs t curve averaged over 100 trajectories

4 Learning to Get Possession of Ball

Getting possession of the ball is not an easy task for the agent. For getting possession of the ball the agent has to first decide whether to intercept the ball (if ball is moving too fast) or just go to the ball's finish position, then the agent has to decide when to position around the ball to get ready for next pass so that it doesn't waste time in positioning. These decisions are hard to encode manually that's why we need a framework for learning complex behaviour.

4.1 Defining Complex Behaviours

We represent a complex behaviour as a state machine whose transitions are governed by the world state. Formally a complex behaviour is defined as

$$\text{ComplexBehaviour} : (\boldsymbol{invoke}, \boldsymbol{abort}, \boldsymbol{getSkill}, \boldsymbol{action\_map}, \boldsymbol{state}, \boldsymbol{nextstate})$$

where, $\boldsymbol{state}$, $\boldsymbol{nextstate} \in \{1, .., n\}$ and $\boldsymbol{action\_map}$ is a mapping between $\boldsymbol{state}$ to a primtive action or a complex behaviour. $\boldsymbol{invoke}$ is a procedure which sets the state to the initial state. $\boldsymbol{abort}$ is a procedure which sets the state to some appropriate state when we want to abort the current behaviour. $\boldsymbol{getSkill}$ is a procedure which makes state transitions based on the world state and returns a primitive action based on variables $\boldsymbol{state}$ and $\boldsymbol{nextstate}$. Note a primitive action can also be wrapped as a complex behaviour with only single state.

We define following 4 primitive actions to choose from for getting possession of ball :

- **INTERCEPT** : move(*intercept_point*), where *intercept_point* is defined as the point perpendicular to ball's trajectory from agent's position
- **GO_TO_FINISH** : move($ballfinish^*$), where $ballfinish^*$ is ball's finish position
- **POSITION** : try to position around the ball to prepare for the kick
- **HOLD** : remain still at current location

World state is represented as list of following state variables :

- distance($agent, ball$); distance($agent, ballfinish^*$); distance($ball, ballfinish^*$);
- angle($ball, agent, ballfinish^*$);
- angle($target, agent$);
- distance($agent, ball$) − distance($agent, prev\_ball$);

The state transitions define the control of a complex behaviour. Algorithm 1 describes how the state transitions can be done using an ANN. The ANN takes the world state as input and outputs n real values, the $arg\,max_{i\in\{1,..,n\}}\,output[i]$ is chosen as $\boldsymbol{nextstate}$. Note that $\boldsymbol{state}$ is changed in the next iteration so that we can call abort and invoke of appropriate states.

Algorithm 1: getSkill

```
  Input: world state
  Output: primtive action
1 $\boldsymbol{state} \leftarrow \boldsymbol{nextstate}$
2 $ann$.load($world\_state$)
3 $output[1,..,n] \leftarrow ann$.activate()
4 $\boldsymbol{nextstate} \leftarrow arg\,max_{i\in\{1,..,n\}}\,output[i]$
5 if $\boldsymbol{state} \neq \boldsymbol{nextstate}$ then
6     $action\_map[\boldsymbol{state}]$.abort()
7     $action\_map[\boldsymbol{nextstate}]$.invoke()
8 return $action\_map[\boldsymbol{state}]$.getSkill()
```

4.2 Learning Control using NEAT

Once we have defined the control using an ANN we can try to refine this ANN to suit our goal. Since its hard to come up with a convex loss function over all inputs to use methods involving backpropogation, we try to optimise the ANN using Neuro Evolution of Augmenting Topologies (NEAT) [5].

NEAT is a genetic algorithm for the generation of evolving artificial neural networks. It alters both the weighting parameters and structures of networks, attempting to find a balance between the fitness of evolved solutions and their diversity. It is based on applying three key techniques: tracking genes with history markers to allow crossover among topologies, applying speciation (the evolution of species) to preserve innovations, and developing topologies incrementally from simple initial structures (”complexifying”).

Each candidate of NEAT was evaluated on the same task as defined in section 3.1 for kick optimisation but with different fitness function. We change the fitness for each episode to the time taken till attempting to kick the ball.

5 Keepaway in 3D Robocup Domain

In this section we define the behaviour of keepers and takers in 3 vs 2 keepaway. Although our study and results are limited to 3 vs 2 keepaway but this description is not limited to 3 vs 2 keepaway and can naturally be extended for **m** vs **n** keepaway.

5.1 Keepers

At any instant each keeper takes a role, named *K1*, *K2* and *K3*. The keeper closest to $ballfinish^*$ (finish position of ball, if ball is moving, else ball's position) takes the role of *K1* keeper. *K1* keeper is the main keeper which makes decisions of where and when to kick the ball. Each keeper independently assesses if it is *K1* keeper or not, if it is not then it listens to *K1* keeper to get its role. *K1* assigns roles to other teammates as following :

- Suppose at timsestep t, *K1* selects $target_t$ as its kick target, then the keeper closest to $target_t$ takes the role of *K2* keeper.
- The remaining keeper takes the role of *K3* keeper.

K1 Keeper At each timestep t, *K1* evaluates the best $target_t$ and tries to maintain the possession of the ball. If *K1* has ball's possession then it decides whether to HOLD the ball or not. Deciding whether to HOLD or not is based on the parameter $min(distance(ball, taker_1), distance(ball, taker_2))$, if $min(distance(ball, taker_1), distance(ball, taker_2)) > hold\_threshold$ then *K1* holds the ball, else attempts to kick at $target_t$. So, essentially the choices *K1* can learn are the choices of $target_t$ based on the state of keepaway and we hope to learn better choices of $target_t$ to yield long episodes of keepaway.

K2 Keeper Behaviour of *K2* keeper is simple, it just tries to reach the $target_t$ broadcasted by *K1*. If it reaches the desired location then tries to face the ball. Note, choice of $target_t$ determines the positioning of *K2* keeper, so a good $target_t$ must take into account *K2*'s current position too (choosing a $target_t$ which is very far from *K2*'s current position is bad).

K3 Keeper Behaviour of *K3* keeper is also kept simple, it just tries to reach a fixed location, *home_position*, and observes the ball after reaching the desired location. *home_position* for each keeper is different and based on where that keeper was born.

5.2 Takers

Takers strategy is fixed and they don't try to learn or improve their strategy. We compare the performance of keepers against two fixed strategies for takers, GREEDY and GREEDY+RANDOM.

GREEDY In this strategy both takers are greedy in the sense that they both greedily move towards ball position. This is the same takers strategy that Stone et al used for 3 vs 2 keepaway in 2D Robocup soccer simulator. Since the movements of agents are not fast in the 3D domain, we don't expect this strategy to be very competitive in 3D keepaway.

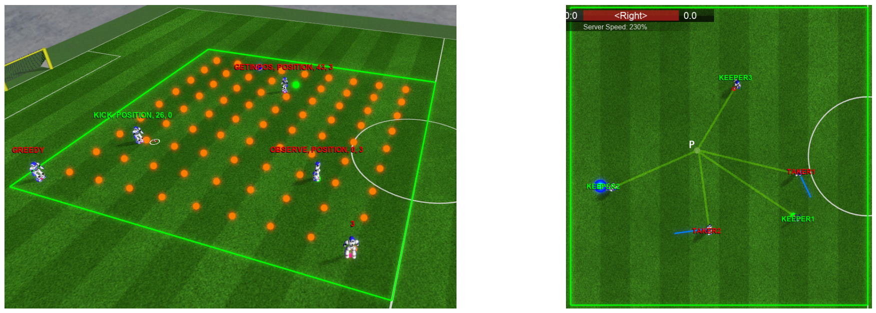
GREEDY+RANDOM In this strategy the taker closest to the ball's position greedily moves towards the ball and the other taker randomly selects a keeper other than the *K1* keeper and follows this randomly selected keeper. To avoid thrashing, the random selection of the keeper to follow is only done when a pass is made.

5.3 Mapping Keepaway onto NEAT optimisation

The way we have defined keepers behaviour in 3D domain, our agent has to select targets from continuous domain, unlike the previous studies done on keepaway in 2D domain where the agents learn to select actions from HOLD, PASS-K2, PASS-K3 (hold the ball, pass to K2 or pass to K3). Selecting targets from continuous domain is not feasible or desirable for many reasons, that's why we select target from a finite set S of points spread out across the field and choose the most promising. Set S constitutes of points from a uniform grid overlaid on the field with each cell of 1 x 1 dimension and 10% margin on each side. For a 10 x 10 field there would be total 9 x 9 = 81 points for evaluation.

If we had to map this kind of policy as a Reinforcement Learning problem then the action would correspond to choosing a target from the set S. The action space would be very large for previously successful RL methods like Q learning to gather enough sample to learn. Also, simulating an episode of keepaway in 3D simulator takes much more time than 2D simulator since 3D simulator has to simulate complex physics behind agent's motion.

So, instead of formulating the task as RL problem we formulated the task as learning a function $cost(F)$ which scores each point in S based on features from feature set F. For a potential target $p \in S$, features are defined as following :



(a) (b)

Fig. 4: (a) Target points for *K1* (b) distance features at point P

- $f_1 : min(abs(angle(bal, p, taker_1)), abs(angle(ball, p, taker_2)))$
- $f_2 : \frac{1}{distance(p,taker_1)} + \frac{1}{distance(p,taker_2)}$
- $f_3 : (distance(p, ball) - 6)^2 + 1$
- $f_4 : distance(p, closest\_keeper)^2$

For playing keepaway in 3D domain, its very important to have a sense of how things are going to evolve in future and choose targets which are not only good now but will become or remain good in future as well, since positioning to kick at certain target will take time. For doing so, some features must capture the dynamics of takers. Taking the immediate velocity is one possibility but its not always a good indicator of where a taker is headed (if a taker is roaming around a point immediate velocity will change drastically). To mitigate this we define $\hat{taker_i}$, for $i \in \{1, 2\}$, as

$$\hat{taker_i} = \frac{\sum_{j=1}^{j=t}(\lambda^{t-j} * taker_i^j)}{\sum_{j=1}^{j=t} \lambda^{t-j}}$$

where $taker_i^j$ is the position of $taker_i$ at time step $t = j$. Now we use these running averages to define features that capture the motion of takers more effectively as following :

- $f_5 : abs(angle(taker_1, \hat{taker_1}, p))$
- $f_6 : abs(angle(taker_2, \hat{taker_2}, p))$
- $f_7 : distance(taker_1, \hat{taker_1})$
- $f_8 : distance(taker_1, \hat{taker_1})$

These are the 8 features in the feature set F. The solution to be learned is an evaluation function *cost* over its 8 feature variables. We represent the evaluation function *cost* as a neural network that computes a real value for a target location $p \in S$ given the 8 dimensional input features. We use NEAT to learn the neural network behind the evaluation function. A particular candidate f is evaluated on 20 episodes of keepaway and the reward for each episode is number of passes made in that episode. We define number of passes in an episode as total number of kick attempts made in that episode. The fitness function for f is the sum of reward of each episode. We start NEAT with initial population of 60 random candidates with no hidden nodes. We compare the learned functions with the hand coded evaluation function (Algorithm 2).

Algorithm 2: Cost : Hand-Coded

```
Input: features $F = \{f_i \,|\, \forall\, i \in \{1,..,8\}\}$ at point $p \in S$
Output: Value at $p$
1 $taker\_coming\_towards \leftarrow false$
2 $taker\_going\_away \leftarrow false$
3 if$((f_5 < 30 \; and \; f_7 > 0.5) \; or \; (f_6 < 30 \; and \; f_8 > 0.5))$ then
      $taker\_coming\_towards \leftarrow true$
4 if$((f_5 > 105 \; and \; f_7 > 0.5) \; or \; (f_6 > 105 \; and \; f_8 > 0.5))$ then
      $taker\_going\_away \leftarrow true$
5 $val \leftarrow f_2 * f_3 / f_1$
6 if$(f_4 > 6 \; or \; f_3 > 4 \; or \; f_1 < 20 \; or \; taker\_coming\_towards)$ then
      return $(val + 10)$
7 if$(taker\_going\_away)$ then
      return $(val/2)$
8 return $val$
```

6 Results

We compare the hand coded evaluation function with the learned evaluation function with respect to both the hold time and number of passes made. The learned evaluation function shows significant improvement over the hand coded cost function. The results are listed in Table 1.

Table 1: Comparison of hand coded vs learned evaluation function averaged over 100 episodes

Evaluation Function	Number of Passes	Hold Time
Hand-Coded	3.1 ± 0.062	$31.764 \pm 0.482s$
Learned	4.55 ± 0.129	$43.238 \pm 1.034s$

7 Related and Future Work

Reinforcement Learning on 2D keepaway testbed has been rigourously studied in RL for Robocup Soccer Keepaway by P. Stone et al. In their work they model the keepaway task as an SMDP and apply sarsa(λ) with linear tile coding function approximation to learn the Q values behind the SMDP. Applications of many evolutionary algorithms like NEAT and HyperNEAT has also been studied on 2D keepaway. In almost all of these studies the keeper closest to the ball is allowed to choose an action from action space HOLD, PASS-K2, PASS-K3 and rest of the keeper go to a open position based on hand coded heuristic. In our approach we don't learn the choice of holding the ball, it is fixed based on a hand coded heuristic, rather we learn the choice of potential pass targets and in doing so we also specify positions to move to for other keepers.

In the paper Learning Complementary Multiagent Behaviors: A Case Study by S. Kalyanakrishnan et al. [3], the GetOpen (positioning) strategy for *K2* and *K3* keepers are learned by learning an evaluation function represented as neural network. In their work the architecture of neural network in fixed and the underlying weight parameters of the neural network is learned using cross entropy method. They show that the learned evaluation function compares on par with a well-tuned hand-coded GetOpen policy.

This work shows the feasibility of keepaway in 3D keepaway but there is siginificant scope of improvement. Currently, the *K3* keeper is not doing anything intelligent, ideally *K3* can position itself for next pass. We believe, significant improvements can be done by improving the low level skills of agents such as improving walk and kick skills. Also, we are passing handcoded features to learn keepaway. We believe that the raw world state contains much more relevant information than the handcoded features we are passing but its hard to learn from them. A learning algorithm which can learn from raw world state should produce better keepaway.

8 Conclusion

We have proposed method to define and learn complex behaviours in the challenging environment of Robocup 3D simulated soccer. We offer a different outlook for defining policies of keepers in keepaway and provide a policy search method for learning high level decision making in keepaway. We also show the superiority of the learned policy against hand coded policy based on heuristics. This work shows the possibility and feasibility of keepaway in 3D Robocup domain against competitive adversaries. This work can be used as strong baseline for future works on keepaway in 3D Robocup simulated soccer.

References

1. Depinet, M., MacAlpine, P., Stone, P.: Keyframe sampling, optimization, and behavior integration: Towards long-distance kicking in the robocup 3d simulation league. In: Bianchi, R.A.C., Akin, H.L., Ramamoorthy, S., Sugiura, K. (eds.) RoboCup-2014: Robot Soccer World Cup XVIII. Lecture Notes in Artificial Intelligence, Springer Verlag, Berlin (2015)
2. Hansen, N.: The CMA evolution strategy: A tutorial. CoRR **abs/1604.00772** (2016), http://arxiv.org/abs/1604.00772
3. Kalyanakrishnan, S., Stone, P.: Learning complementary multiagent behaviors: A case study. In: Proceedings of the RoboCup International Symposium 2009. Springer Verlag (2009), http://www.cs.utexas.edu/users/ai-lab/?LNAI09-kalyanakrishnan-1
4. MacAlpine, P., Urieli, D., Barrett, S., Kalyanakrishnan, S., Barrera, F., Lopez-Mobilia, A., Ştiurcă, N., Vu, V., Stone, P.: 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) (June 2012)
5. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evol. Comput. **10**(2), 99–127 (Jun 2002). https://doi.org/10.1162/106365602320169811, http://dx.doi.org/10.1162/106365602320169811
6. Stone, P., Sutton, R.S., Kuhlmann, G.: Reinforcement learning for RoboCup-soccer keepaway. Adaptive Behavior **13**(3), 165–188 (2005)