Vision-based interaction with virtual worlds for the design of robot controllers

D. d'Aulignac, V. Callaghan and S. Lucas
Department of Computer Science,
University of Essex,
Colchester CO4 3SQ, UK

Abstract

This paper discusses the use of simulated robots in a virtual table tennis environment. Specifically, the use of a two-segment arm for the control of a bat and the possibilities of using vision as a human-computer interface are investigated. The motivation of this work is to provide a tool to test robot controllers in a challenging environment to perform non-repetitive tasks. The benefits of such research include the development of controllers that are able to operate in non-ordered, unpredictable environments, e.g. hazardous environments.

1 Introduction

Currently there is much interest in neural networks for control of real robots. However, to experiment with real systems requires expensive hardware which is often difficult to set up and modify. The alternative is to use simulators which will help to cut down on cost and development time.

This was one of the main reasons that led to the first implementation of a virtual table tennis environment [1]. However, in this version forces are directly applied to the bat, without worrying who or what would achieve this. Work by [5] has produced a robotic arm simulator to test neural networks for hand-eye coordination. We used part of this work to integrate a simple robotic arm into our simulation. Thus forces are not applied directly to the bat, since the latter is coupled to the end-effector of the arm.

Neural networks have been designed or evolved for solving control problems, and separately for strategy games such as tic-tac-toe. Table tennis provides an interesting mix of control problems and strategy problems, making the design of good players extremely challenging.

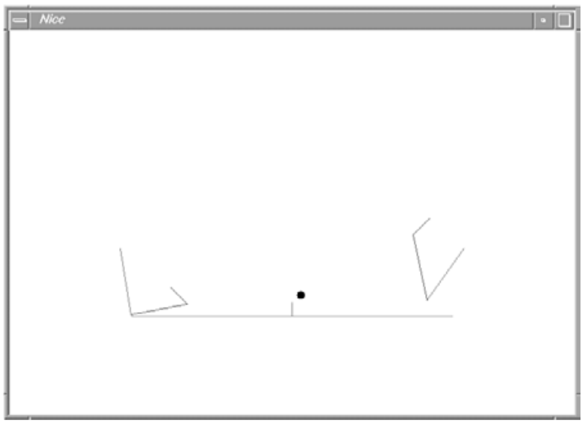

Figure 1: Screen-shot of the display client.

The rest of this paper describes the virtual environment in Section 2; Section 3 describes the human, vision-based controller; Section 4 explains the neural network based approaches used to solve the control of the arm. In Section 5 the possible developments of this project are discussed, before Section 6 concludes on the results so far.

2 The Virtual Environment

Here we choose a 2-dimensional simulated table tennis game as our environment. The table is viewed from the side, with the net located in the middle, and the bats and robotic arms operating in a plane. Figure 1 shows a screen-shot of the display client which visualises the state of the game.

The simulator is implemented in an object-oriented style using C++ and uses sockets to communicate with the controller and display

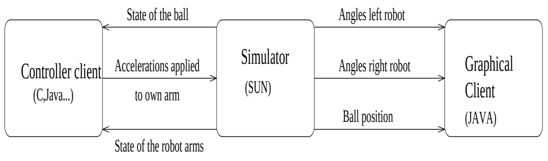

Figure 2: Diagram showing the messages exchanged between clients and server

clients. These are normally implemented in Java to exploit Java's platform independence and networking capabilities. Figure 2 shows how these communicate with each other: the simulation server sends the current state (position and velocity) of the ball and each of the segments that make up the robotic arm, and expects back an acceleration value for each of the links in the arm from the controller client. To play a game two controllers will have to be connected. If display clients are connected the current position of the arms and the ball will be sent to them.

Hence, the implementation has three aspects: the simulation model acting as the server, and the display and controller that are the clients.

2.1 World model server

The world model needs to take into account both the logical model, which defines the rules of the game, and the physical model, that is responsible for applying the laws of physics to the objects in the game. The responsibilities of the logical model are to detect the end of a point, the winner of the point, and decide which player currently has the service. For detecting the end of a point, and the winner of the point, it interacts with the physical model, which updates the dynamic objects and monitors their collisions with the static objects (table, net, floor, and ceiling). After each collision the physical model informs the logical model in order to update its state.

2.1.1 The ball

The ball is a dynamic yet passive object, which gets hit around according to the laws of physics. The physical model accurately describes most of the features of the real game (except for the missing third dimension) including gravity, air resistance, the effects of spin on ball trajectory and collisions, and the coefficients of friction between bat and ball and between table and ball. These effects can also be switched off in order to provide a simpler game environment if necessary.

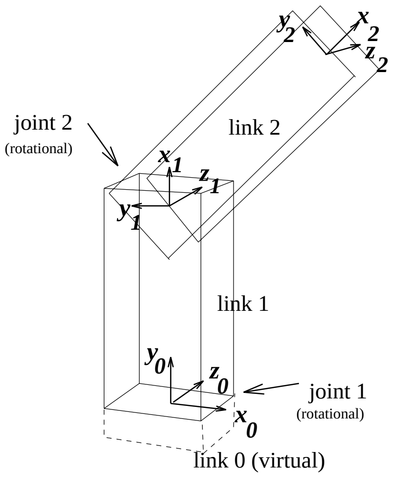

Figure 3: Assigning frames to the robot. (Adapted from a figure in [5])

i	θ_i	α_i	a_i(cm)	d_i(cm)
1	θ_1	0	50	0
2	θ_2	0	50	0

Table 1: Denavit-Hartenberg table for the robot.

2.1.2 The bats

The bats are active dynamic objects. On collision with the ball they will change the trajectory of the latter depending on the bat's velocity, angle, and mass. For the purpose of introducing spin we assume that the ball always rolls and never slides on the bat, due to the friction of the material used to manufacture modern bats.

2.1.3 The arm

The robot arm is a two-segment arm. To fit into the 2-dimensional simulation it has rotational joints only operating in a plane. The assignment of frames to links is chosen in accordance with [2] (see Figure 3). The corresponding Denavit-Hartenberg table is given in Table 1.

Applying accelerations on the links results in the movement of the arm; the kinematics of robot arm are examined for this. Homogeneous matrix transforms are used to work out the *direct kinematics* for the arm, i.e. the position of the links in Cartesian space. To work out the velocity of the end-effector (important since the bat is attached to it) we used the recursive Newton-Euler Formulation, which works out the velocities from the base to the tip of the manipulator. We choose this method since it is the computationally much more effective than its closed form. Throughout the system it is aimed for a rigourous use of the SI units which would make an eventual hardware implementation in the future as easy as possible.

2.2 Display Client

This is used to display an animated view of a game. For the machine based controllers it is entirely unnecessary, and games can be played much faster without one (this being advantageous, for example, when evolving controllers, which requires to play many games). However, it is useful to observe the traits of various machine-based players in order to better understand their strength and weaknesses. Also, it is essential if it is required to allow human players to play.

2.3 Controller Client

The controller will receive and up-to-date state of the world model, including position and velocity of the ball, bats, and arms and is then expected to return the accelerations on the links of the arm it controls.

3 A vision-based human controller

The basic idea behind the visual interaction for the control of the simulated robot arm is to colour code the joints on the human arm captured by the camera. This will make it possible to find the position of the joints with respect to each other.

To find out where in the image the coloured patches marking the joints are, we find the Euclidean distance in the HSI space between every pixel in the image to the colour we are looking for. We then assume the pixel with the smallest distance from the colour we are seeking is the corresponding label. Knowing where the labels are we can work out the angles using simple trigonometry. These are then sent to the simulation via the socket. The advantage of using the HSI colour space over the RGB format is that we can eliminate the intensity of a pixel [4]. Thus we get better colour constancy over varying lighting conditions. We ignore pixels with very small intensity values, since the colour of very dark regions is ambiguous. Also very bright light can cause problems since 'white spots' may appear making it hard or impossible to identify the colour correctly. The latter is correlated to the reflective properties of the material used for the colour patches.

During normal operation, target joint values θ_d are detected from the visual input. The desired velocities to move the joints to the target is defined by $\dot{\theta}_d$. The accelerations $\ddot{\theta}_d$ are set such as to attain the velocities $\dot{\theta}_d$. Thus, the values θ_d, $\dot{\theta}_d$, and $\ddot{\theta}_d$ specify the new path. This new path results, at the following time slot, in the setting of the current acceleration $\ddot{\theta}$ to $\ddot{\theta}_d$. The acceleration is maintained at this level until either $\dot{\theta} = \dot{\theta}_d$ or $\theta = \theta_d$. In the former case, $\ddot{\theta}$ is set to 0 until $\theta = \theta_d$. In the latter, both $\dot{\theta}$ and $\ddot{\theta}$ must be set to 0 to stop the robot. The correct value for $\ddot{\theta}_d$ must be given, and a 0 for $\dot{\theta}_d$ to implement this.

Although the main reason for implementing this vision-based controller was to provide training data for supervised neural networks, an added benefit of this is that it becomes possible to play virtual table-tennis with two human players over the network, assuming that we have two SGI machines with cameras.

4 Neural Network Based Controllers

The main aim is to design robot controllers and train them to play a 'good' game. Two approaches were tested. Firstly, a single network was used and trained by the human, vision-based controller. For this approach the multilayer perceptron (MLP) was used. For the second approach the task was divided into a small number of neural networks. Each network was trained to do a particular task. Then, they were all combined to integrate a player.

4.1 Inputs and Outputs

A realistic input vector for a neural network would be the position and velocity of the ball, and the position and velocity of the segments in the arm that it controls. The output would consist of the acceleration applied to the arm segments and the torque that is applied to the bat.

However, for simplicity at this stage the torque is ignored. We are just interested in moving the arm, while leaving the bat at a fixed angle. Using the human, vision-based controller a set of training patterns is produced. This should include as many representative cases as possible.

4.2 Single Module Network

The results of the single module network, at first, appeared to be strange. The MLP was trained on the training data, and repeatedly tested on the test data until the test-set error reached a minimum. These learned weights were then hard-wired into a robot controller to play in an actual game. The network generally performed poorly, frequently missing the ball and resting at one position without moving.

The most probable cause of this is that while the network behaves well in the regions which the human player has explored, it has no reason to behave well outside of these regions. As soon as the neural network begins to stray from what the human controller has done in a given situation, the problem accumulates — and the arm is rapidly sent into regions of the input space where the human has never explored.

Perhaps a further problem is the pseudo-random behaviour of the human player (i.e., there is no unique way to play a shot) and that the arm controled by the human remains mostly in a 'resting position' after playing a shot and waiting for the opponent to execute his. If a lot of the outputs are the same the easiest way for the MLP to minimise the error is to assume outputs are constant. This results in the arm controlled by this neural network to remain at this resting position without much other movement.

4.3 Modular neural networks

The second approach is to decompose the task into a number of smaller tasks. Each smaller task can then be handled by an independent specialist neural network. With this modular approach, it is of course possible to have different modules based on different paradigms. Also, modularity is of paramount importance in good engineering design. In this way, it is possible to identify which neural network modules are performing well and which ones are performing poorly.

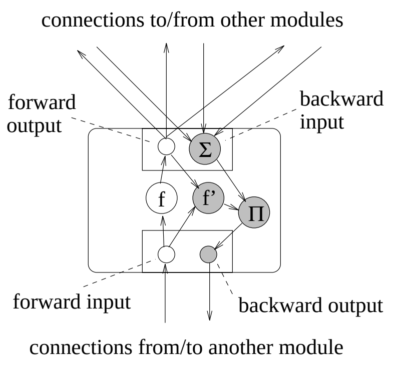
connections to/from other modules
forward output
backward input
Σ
f
f'
Π
forward input
backward output
connections from/to another module

Figure 4: A general forward-backward node.

We decompose the problem into three parts: prediction of the intercept positions, prediction of intercept velocities and movement of the arm to achieve the desired intercept.

4.3.1 Prediction of desired intercept positions

To find the positions of the segments in the arm such that the bat on the end effector will intercept the ball we use the experiences provided by the human, vision-based controller. As inputs we take the position and velocity of the ball when it leaves the opponents bat, and the outputs are the positions of the arm. An MLP is used to predict the positions to intercept, trained on the recorded experiences form the human player. This implicitly solves the inverse kinematics for the intercept points. The MLP accurately predicts the intercepts for shots that are not far outside the input space it has been trained upon.

4.3.2 Prediction of desired intercept velocities

Based again on the human, vision-based controller, another MLP is trained on the velocities of the links in the arm at the intercept. The velocities on the links must be such so as to return a good shot. For this task, neural networks have been trained successfully. Thus, if we cheat and warp the arm to the desired position with the desired velocity for each link at the correct time, we have a combination of two neural networks that can play.

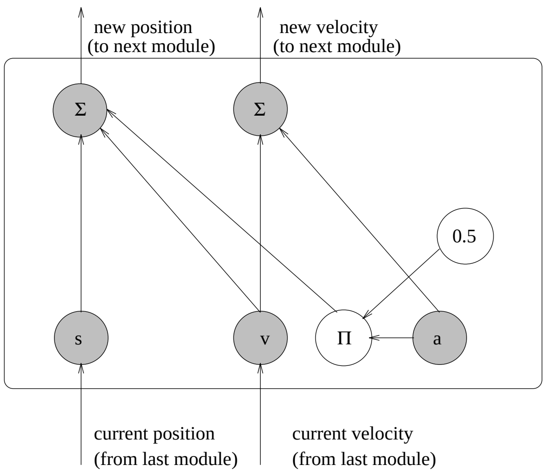

Figure 5: Forward cycle of module that updates position and velocity according to their initial values and the acceleration

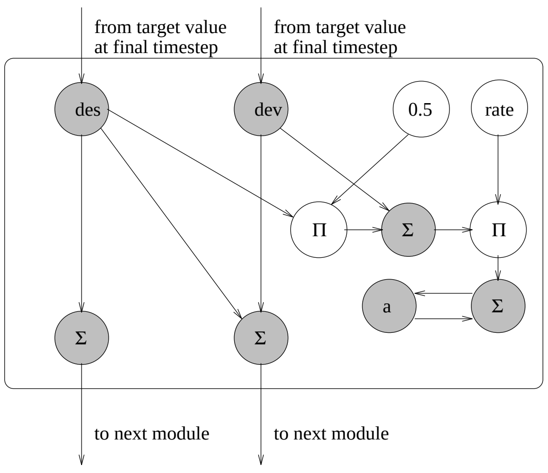

Figure 6: Backwards cycle of module that updates the acceleration according to the derivative of the final error in position and velocity with respect to the acceleration

4.3.3 Moving the arm

The third network module has the task of moving the arm over successive time intervals in order that at the time of intercept, the arm has the desired position and velocity. To achieve this we use forward backwards modules [3].

The general form of the forward-backward module is shown in Figure 4. It computes function f for its forward behaviour. For its backward behaviour it computes the derivative f' of the function with respect to the input (or the partial derivative w.r.t. that particular input), multiplies it by the accumulated backward error (in the Σ cell) and passes it back to its backward output.

At each time-step the position and velocity of the segments in the arm are updated according to Newtonian laws, which are well known. If the acceleration at each time-step and the number of time-steps is knows, it is possible to find the final position and velocity. The idea is that given the error in position and velocity after the final time-step, it is possible to back-propagate this error through the network to make the corresponding adjustments to the acceleration applied at each time-step.

Figure 5 shows the forward cycle of a module that updates the velocity and position of a link in the arm according to Newton's Laws.

Figure 6 shows the backward cycle of a module that updates the acceleration according to the derivative of the final error in position and velocity with respect to the acceleration. Since

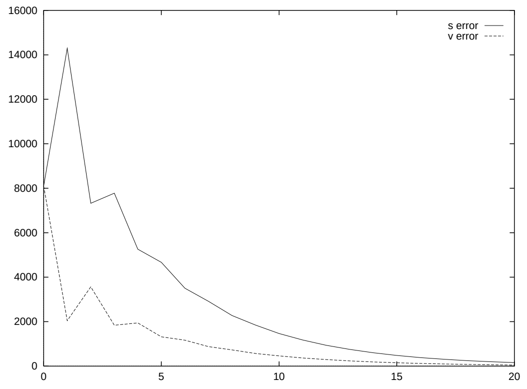

Figure 7: The mean-squared-error from the target values for position and velocity plotted against the number of iterations

the error normally is reduced over several time-steps, there will be as many modules as there are time-steps. The modules are connected to each other as shown in the figures.

Thus, for example, if one segment of the arm was at position 0.0 and velocity 0.0, but the target position and velocity in 200 time-steps was 90.0 and -90.0 respectively, the link would have to move *past* the 90.0 degrees mark, only then to move back again to achieve the -90.0 velocity at the final time-step. In Figure 7 it can be seen how both the mean-squared-error in position and velocity decreases over 20 iterations. The mean-squared-error after 100 iterations is $3.41331e^{-06}$ for the position and $1.06807e^{-06}$ for the velocity.

5 Discussion

We are at the stage where a multi-module neural network can play and game of table tennis within its current environment. The next stage in the work is to take the most successful neural network individuals and apply tournament based evolutionary methods to evolve better players.

The current implementation of the game has been designed to be an accurate simulation of a real, but 2-dimensional, table tennis environment. There are several ways the set-up can be varied. The game can be made simpler by eliminating the effects of spin and air resistance. Alternatively, the parameters which control these can be adjusted to increase their effect, hence making the game more difficult.

Other ways in which the game can be made more difficult are: extend the simulation to three dimensions; limit the amount of computation allowed for each controller at each time-step. The latter would have the effect of favouring controllers that not only play a good game, but do so within some bounded amount of computation. Both these ideas, of course, are essential in the sense of using the evolved controllers in a real-time, 3-dimensional world such as ours. Before these are explored, however, there is plenty of scope for improving the performance of current robot players.

6 Conclusions

This paper has described a framework that allows the development and evaluation of robot controllers in a simulated table-tennis environment. From the experiments conducted we conclude that using a single neural network, trained on the experiences provided by a human player through the vision-based interface, is largely unsuccessful. Better results can be achieved by dividing the task into smaller sub-tasks, thus simplifying the problem but also ensuring modularity, which allows to test modules separately. This provides us with a better insight into the operation of the controller.

Already the project has generated a good deal of interest on the Internet. Hopefully, with the introduction of this new version of the simulator, allowing controllers to connect to a server via a network, the added ease of use will encourage more people to submit their controllers to an Internet-based tournament. It would be interesting and could be insightful which type of architecture performs best.

Finally, this kind of work provides a natural bridge into the design of real robot game players (i.e. to play table tennis against each other, and/or against humans on a real table tennis table). By developing the details of the robot controller in a virtual environment, much of the design work can be done much more quickly than if having to deal with real robots.

Acknowledgements This work was partially supported by post-graduate EPSRC grant (Ref: 96419892) and UK EPSRC grant GR/J86209.

Related web sites For more information on the project visit our project home-page: http://peipa.essex.ac.uk/vase/projects/table-tennis/

References

[1] D. d'Aulignac, A. Moschovinos and S. Lucas. Virtual table tennis and the design of neural network players. In *Int'l Conf. on Artificial Neural Networks and Genetic Algorithms*, April 1997.

[2] K. S. Fu, R. C. Gonzalez, and C. S. G. Lee. *Robotics: Control, Sensing, Vision, and Intelligence.* McGraw-Hill, 1987.

[3] S.M. Lucas. Forward-backward building blocks for evolving neural networks with intrinsic learning behaviours. In *Lecture Notes in Computer Science (1240): Biological and artificial computation: from neuroscience to technology*, pages 723 – 732. Springer-Verlag, Berlin, (1997).

[4] R. Jain, R. Kasturi, and B. G. Schunck. *Machine Vision.* McGraw-Hill, 1995.

[5] P. van der Smagt. Simderella: a robot simulator for neuro-controller design. *Neurocomputing*, 1994.