Point-to-Point Car Racing: an Initial Study of Evolution Versus Temporal Difference Learning

Simon M. Lucas and Julian Togelius
Department of Computer Science
University of Essex, Colchester, UK
{sml, jtogel}@essex.ac.uk

Abstract— **This paper considers variations on an extremely simple form of car racing, the challenge being to visit as many way-points as possible in a fixed amount of time. The simplicity of the models enables a very thorough evaluation of various learning algorithms and control architectures, and enables other researchers to work on the same models with relative ease.**

The models are used to compare the performance of various hand-programmed controllers, and neural networks trained using evolution, and using temporal difference learning. Comparisons are also made between state-based and action-based controller architectures. The best controllers were obtained using evolution to learn the weights of state-evaluation neural networks, and these were greatly superior to human drivers.

Keywords: Car racing, reinforcement learning, evolving neural networks.

I. Introduction

This paper introduces a class of point-to-point car racing games, where the racing can take place in a variable number of dimensions, and with alternative car models. The aim is to create a set of benchmark controller learning problems on which to test various machine learning algorithms.

In particular, we are interested in comparing evolution (and co-evolution for multi-car races) with temporal difference learning, but we have made the simulation code freely available on the web to encourage other researchers to try other methods.

In a recent paper [5] Runarsson and Lucas investigated temporal difference learning (TDL) versus co-evolutionary learning (CEL) for small-board Go strategies. There it was found that TDL learned faster, but that with careful tuning, CEL eventually learned better strategies. In particular, with CEL it was necessary to use parent-offspring weighted averaging in order to cope with the effects of noise. This effect was found to be even more pronounced in a follow-up paper by Lucas and Runarsson [4], comparing the two methods for learning an Othello position value function. In this paper we conduct a similar kind of investigation, but for a very different problem: a simplified kind of car racing game. In this paper we consider only a single-player game, and therefore use evolution rather than co-evolution to learn controllers, but the comparison is in a similar vein.

A. Car Racing

Competitive car driving is a problem of great practical importance, and has received some attention from the computational intelligence community. Most often researchers have used various learning methods for developing controllers for car racing simulations or games [3][2]. But computational intelligence techniques have also been applied to physical car racing, famously by Thrun in the DARPA Grand Challenge [8], but also by e.g. Tanev et al. who evolved controllers for radio-controlled toy cars [7]. The radio-controlled toy racing challenge was run as a competition for IEEE Congress on Evolutionary Computation (CEC) for 2003, 2004, and 2005. The challenge was for a computer to drive the car around a simple flat track with walls, the track being around the size of a table-tennis table. The input to the system was from an overhead web-cam. The challenge has yet to see a really high-performance entry (e.g. one competitive with a competent human racer). Footage of the 2003 competition can be viewed here[1]. There are many challenges with the real cars, including problems with computer vision, variable time-delays in image capture and processing, and difficulties in precisely modelling the physics of the car and its interaction with the track, the walls, and the other car in the case of multi-car racing.

On the other hand, there is much to be learned from studying simulations of the racing challenge. While we hope to transfer what we learn from the simulations back to the real-world problem, they are also interesting in their own right (see Togelius and Lucas [9] [11][10]; Togelius *et al* [12]). It was found that controllers based on first-person sensory inputs and neural networks could be evolved to achieve robust driving behaviour over a number of racetracks, perform better than humans when specialised on a particular track, and display interesting competitive behaviour when co-evolved with another car on the same track. The authors have since tried developing controllers for that car simulation using temporal difference learning, but have had little success. It is hoped that the investigations in the present paper throw some light on this, even though the car simulation in the earlier experiments differ from the point-to-point model in some respects, most importantly the presence of impenetrable walls in the former model.

Abbeel and Ng [1], who trained a system to imitate human drivers on a simulation. They argued that specifying a reward function would be hard, hence it was easier to train the system by observation. In our study evolution of state value functions works well, and easily outperforms human drivers

[1] http://www.youtube.com/watch?v=-KvL7zOZNAc

(though our controllers have simpler objectives, in that they have no road rules to obey).

II. Benchmark Requirements

To enable the effective study of machine learning algorithms within a car-racing domain, we began with a set of requirements:

- The capability of generating a large number of random tracks with very little effort.
- The model must be fast to compute. Evolutionary algorithms may require millions of simulated time steps in order to converge.
- The sensor inputs for a controller should be reasonably simple; this encourages more members of the research community to participate.
- The setup should present sufficient challenge to recognize skill. Furthermore, this should be related to driving skill, rather than just traveling-salesman style route optimization.

The first point regarding random tracks is important to allow testing of the controllers on large numbers of tracks that were unseen during training. This ensures that we are testing general driving behaviour, rather than the specialised ability to race a small number of tracks.

To get an idea of the difficulty of the problem, a keyboard controller was implemented and used by the authors to drive cars as quickly as they possibly could using the normal and holonomic cars in two dimensions. Best performances obtained in each case were between 12 and 16 waypoints. The evolved neural networks easily surpassed this, achieving best performances of over 40 waypoints.

III. The Models

A simple class of model that meets the above requirements is point-to-point racing. This can work in n-dimensional space in general, but in this paper we restrict it to one or two dimensional racing. In each case, the challenge is to touch as many waypoints as possible within a fixed number of time steps. The waypoints must be touched in order, with only p visible at any one time. For this paper we fix $p = 3$, which rewards some planning ability while forcing the controller to cope with some uncertainty.

In this paper three models are considered: one dimensional, two dimensional holonomic, and two dimensional car.

A. Random Track Creation

Each waypoint is drawn randomly from a uniform distribution on the unit hypercube. This is done using an instance `r` of Java's `java.util.Random` class as a pseudo-random number generator, and taking the value of each dimension in turn from call to `r.nextDouble()`. If a no-argument constructor is used to create `r`, then a different random track is created each time (up to the number of possible random seeds), but passing an `int` (e.g. `r = new Random(10)` enables a simple way to specify a pre-defined track (in this case, it would be referred to as Track 10)).

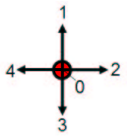

Fig. 1. The five possible actions (force vectors) available to a Holonomic controller.

B. One Dimensional

For the one-dimensional case, the car is represented as a point on a line with unit mass, displacement s and velocity v. Time t is discretized, and at each time t the controller selects one of three possible accelerations $a_t \in \{-a, 0, a\}$. To visit a waypoint during a move at time t, the requirement is that the waypoint at position x lies between s_t and s_{t+1}.

C. Two Dimensional Holonomic

To illustrate the two-dimensional case, imagine racing a vehicle around a large flat airfield, where an ordered set of random waypoints pop up. In the holonomic case, the vehicle has no heading. At each time-step the controller chooses either to apply no force, or a force vector of magnitude $||a||$ in one of the four orthogonal directions, making for five possible actions in total (figure 1). At each time step, the displacement and velocity are updated using Newtonian mechanics for a point-mass.

For waypoint touching calculations (and for car collisions, in the case of multi-car races), both the waypoints and the vehicles are modelled as discs with radius r_w and r_c respectively. A waypoint w is touched at time t if the following condition is satisfied:

$$||s - w|| \leq r_w + r_c$$

A slightly more complex but conceptually more accurate test would be whether the rectangle swept out by the width of the car in moving from s_t to s_{t+1} overlapped the waypoint w, but the simpler test was deemed adequate for our purposes, provided that the radii were made sufficiently large. In all our simulations, both r_x and r_c were set equal to twice the magnitude of the acceleration $||a||$, where $||a|| = 0.01$

D. Two Dimensional Car

This is similar to the above two dimensional holonomic case, except that the car now has a heading which is distinct from its velocity (allowing for some skidding), and that all forces are now applied in relation to the heading of the car. Just as in a real car, accelerating forward means accelerating along its heading vector.

IV. Software Implementation

All software has been implemented in Java. All car controllers implement the `Controller` interface (figure 2). A controller is given the immediate set of next n waypoints, passed in an array called `track` such that the next one is always in `track[0]`, the one after next in `track[1]` and so on. Each time that a car passes a waypoint, the

```
public interface Controller {
  public int action(WayPoint[] track, ICar c);
}
```

Fig. 2. The interface for a Car Controller. It takes an array of waypoints, and the current state of the car as inputs, and returns an action choice as its output.

```
public interface ICar {
    public Vector2d s();
    public Vector2d v();
    public Vector2d heading();
    public double rad();
    public ICar update(int action);
    public ICar copyAndUpdate(int action);
}
```

Fig. 3. The interface for a Car in two dimensions. The current implementations of this are HoloCar, and Car.

track array is shifted along, and a new random waypoint is inserted into `track[n-1]`. Waypoint positions are sampled from a uniform distribution in the range 0 to 1, using the instance method `r.nextDouble()` from Java's `java.util.Random` class.

To allow state-based control methods, all the software car-models implement a `copyAndUpdate()` method. This method takes a proposed action as an update, and returns an updated copy of the state of the simulation having taken that action. This enables *what if* style forward planning, and in particular, makes it straightforward to implement value based controllers either for reinforcement learning or evolution. The method is also used by the Fixed Controllers (see Section V).

The behaviour of a car is abstracted through the interface (pure virtual class) `ICar`, shown in figure 3. This enables *exactly* the same controller to be used to drive either a holonomic (figure 4) or a normal car (figure 5). In this code, `s` and `v` are instances of `Vector2d`, a class for two-dimensional vectors, and `t` is the time interval. The `add` method performs weighted addition. The normal car also includes a heading vector `h` and a steering constant: the angle turned through per unit of forward movement. This was set to 8.7 radians to give a reasonable turning circle for the car (this value may sound very high, but the car typically has a speed of less than 0.05 units per time step).

V. Fixed Controllers

For each problem setup, we implemented two hand-coded controllers: *Greedy* and *Heuristic*.

```
public HoloCar update(int action) {
    s.add(v, t);
    s.add(a[action], 0.5*ac*t*t);
    v.add(a[action], ac*t);
    return this;
}
```

Fig. 4. The vector arithmetic to update the position and velocity of a holonomic car.

```
public Car update(int action) {
    return update(acc[action], steer[action]);
}

public Car update(double acc, double steer) {
    s.add(v, t *0.5);
    v.add(h, acc * t);
    h.rotate(steer * h.scalarProduct(v));
    double mag = h.scalarProduct(v);
    v.set(h);
    v.setMag(mag);
    s.add(v, t *0.5);
    return this;
}
```

Fig. 5. The vector arithmetic to update the position and velocity of a normal car.

The algorithm for each of these controllers is simple: consider the state of the system after each possible action (i.e. the set of afterstates in reinforcement learning terminology [6]), and select the action that leads to the best score. In this case, the scores are penalty values, so this means the action with the lowest score will be selected.

The score function used for the greedy controller is simply the Euclidean distance between the car and the next waypoint. The heuristic controller improves on this by adding in a penalty term proportional to the square of the car's speed. The square of the speed is used on the basis that stopping distance is proportional to this.

The greedy algorithm performs poorly; its failure to consider the velocity of the car leads to a tendency to significantly overshoot each waypoint. This is because given the current state of the system, the action that takes the car closest to the next waypoint usually involves accelerating toward the waypoint.

The addition of the velocity penalty in the heuristic controller leads to much better performance. It is instructive to consider the code for this controller, which is shown in Figure 6. Note that setting the velocity penalty to zero gives the Greedy Controller.

One of the great strengths of state-based controllers is that they can work with little modification across a range of problems. For example, the code in Figure 6 was used *unchanged* to control a holonomic car, and a non-holonomic car (both in two dimensions), each one achieving reasonable performance. It may seem surprising that the same heuristic works quite well for either type of car. This can be understood by the nature of the heuristic: *reward being close to the next waypoint, but penalise driving too fast*. In this way, driving style is highly abstracted from the details of any particular car.

The heuristic controller was applied to a normal car on Track 29 (recall naming convention from Section III-A) with the velocity penalty set to 4.0. After the seventh waypoint it gets stuck, oscillating in small orbital steps around the eighth waypoint. Hence, after 500 time steps it only achieves a score of 7. Changing the velocity penalty alters the behaviour sufficiently to avoid this, and increasing the penalty to 4.8

```
package games.twod;

public class HeuristicController implements
  Controller, Constants {
  // set to zero for a 'Greedy' Controller
  static double velocityPenalty = 4.75;

  public int action(WayPoint[] track, ICar hc) {
      // step over each possible action
      // and take the one that leads to
      // the lowest score
      int action = -1;
      double best = Double.MAX_VALUE;
      for (int i = 0; i < nActions; i++) {
          ICar next = hc.copyAndUpdate(i);
          double score = score(track, next);
          if (score < best) {
              best = score;
              action = i;
          }
      }
      return action;
  }

  public double score(WayPoint[] trk, ICar hc) {
      double dist = trk[0].p.dist(hc.s());
      return dist + velocityPenalty *
         hc.v().mag() * hc.v().mag();
  }
}
```

Fig. 6. The Java code for a Heuristic State-Based Controller (complete implementation)

leads to a much improved score of 29. This is achieved through impressive driving behaviour, with appropriate use of forward and reverse driving. The exact same heuristic controller (with velocity penalty set to 4.0) on the same track, but controlling a holonomic car, achieves a score of 25, compared to a score of 26 when the velocity penalty is set to 4.8. In this case, more careful driving leads to less overshoot and a slightly higher score. This class of heuristic controller never gets stuck in a stable or oscillating orbit around a waypoint when applied to a holonomic car, but this is a significant danger when applied to a normal car.

Importantly, this illustrates that even with this simple setup, simulated with a few straightforward Java classes, complex behaviour patterns can emerge, and interesting control problems arise, providing a good test for machine learning algorithms.

VI. Learning State Value Functions

In this section we report on some initial work comparing evolutionary algorithms with reinforcement learning to learn state value functions that lead to high performance behaviour.

We apply the same techniques to the three classes of problem: one-dimensional, two-dimensional holonomic, and two-dimensional normal car.

Experiments were made with various setups, but it was found that in general much better results could be obtained by using prior knowledge of the problem domain to construct

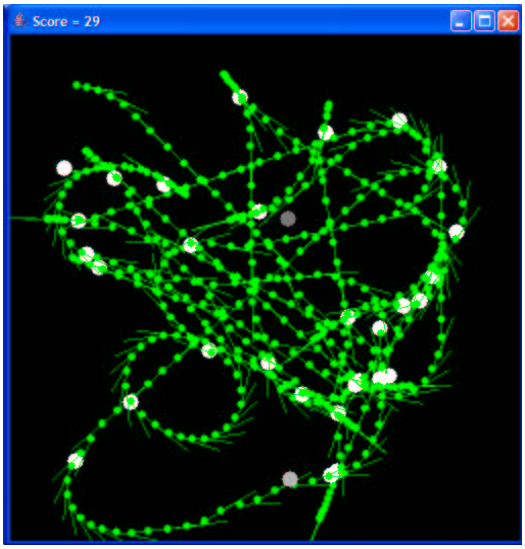

Fig. 7. Trace of the heuristic controller applied to a normal car with velocity penalty set to 4.8. The controller does not get stuck, and performs well with a score of 29 waypoints.

a meaningful feature vector, rather than just feeding it the state of the car i.e. its velocity, and the relative positions of the next waypoints.

A. Evolution

In this section we report on the evolution of state value functions. In performing these experiments, we used our knowledge of the problem domain, together with some experimentation, to design a feature vector. For all of these experiments we used a (15+15) ES, running for 100 generations, and the number of waypoints visited in 500 time-steps as the fitness function (this performance indicator was also used on the test tracks). The randomly generated tracks vary significantly in their difficulty, making fitness evaluation highly noisy. In extreme cases we observed controllers with a score of over forty on one track, only to fail miserably with a score of zero on another track. To counteract this, we experimented with two versions of the EA. In version one, a single new random track was used to evaluate all the individuals in a given generation, while in version two, each individual was evaluated on a new random track. We expected version one to perform better, since each controller would be compared fairly against each other one, but the performance of the two techniques was inseparable. Results quoted are for version two. We used two versions of the feature vector with two and three inputs respectively. The two-input version consisted of the Euclidean distance to the next waypoint, and the square of the car's velocity. The three input version takes these two and also adds a directional feature: the absolute value of the *sin* of the angle between the car's heading and the direction to the next waypoint.

In each case the value function then applies the feature vector as input to a neural network, whose single output is taken to be the value of that state. Figure 8 shows the fitness evolution of an MLP (1 hidden layer with 10

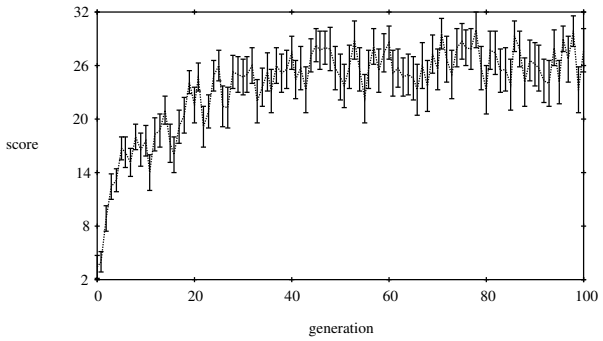

Fig. 8. Evolving a state-value MLP for controlling a normal car.

units) controlling a normal car in two dimensions. Good performance usually evolves in around forty generations. The error bars show the range of fitnesses ($pm\ \sigma$) in each generation. Overall, evolving state evaluation networks was by far the successful of all methods studied in this paper (see Results Summary Section). This setup was also used to evolve state-based controllers for the one-dimensional problem, which achieved scores at least as good as the action-based controllers reported in the next section.

B. TDL for learning state-values

This work is still in progress. For the one dimensional problem, TDL behaves erratically. When it does learn, it learns very rapidly, and achieves competitive scores.

Early results on the two-dimensional track are very dependent on the car type. For the normal car, TDL was often competitive with evolution, with the very best learned controllers having very similar performance. While evolution also worked well for the holonomic car, TDL failed badly for this case. For the normal car, when TDL did learn it often achieved high performance within the first 10 epochs, and in these cases offered much faster learning than evolution. We are currently exploring hybrid algorithms designed to exploit the best of each method.

VII. Learning action value functions

When a model of the agent is not available to the controller, the only way to learn a policy with temporal difference learning is to learn the value of (state, action) pairs. Once those values are learnt, the controller works by following a greedy policy: for a given state, look through all possible actions and select the action that has the highest value. The first issue to grapple with here is that of how the function mapping from states to values should be represented.

A. Controller representations

The two main ways of representing learnable (state, action) to value mappings are as tables and via function approximators. Table representations have the obvious drawbacks that they are discrete, requiring the designer to choose how to divide up a continuous multidimensional input space into a finite (and, to keep the number of evaluations required low, rather small) number of table cells. Function approximators, such as neural networks trained with backpropagation (which we use in this paper), might seem like an obvious choice, but often perform badly in practice on this sort of problem.

1) The one-dimensional model: For the one-dimensional model, we devised a table of 3 to the power of 4, or 81 cells. The position in the first dimension was selected depending on the speed of the car, and the next two depending on distances to the current and next waypoint; the first cell in a dimension was selected if the value was below -0.1, the third if above 0.1 and the second otherwise. The resulting row is interpreted as the value of the three possible actions for this state.

The alternative representation is based on an MLP with one hidden layer and *tanh* activation function. The inputs to the neural network is the position of the car, the speed of the car, the positions of the two waypoints, a bias, and the action whose value is sought. The hidden layer has size five. The single output is interpreted as the value of the action.

2) The two-dimensional holonomic model: The action-value table for the holonomic agent had one more dimension than for the one-dimensional model. The row containing the action values is obtained through selecting positions in the first four dimensions based on speed in the horizontal and vertical dimensions, and the relative position of the waypoint in the horizontal and vertical dimensions.

Another was designed around an MLP, just like in the one dimensional case, taking as inputs the position of the car, the velocity vector , the position of the next waypoint, and the action. The output is interpreted as the value of that action.

3) The two-dimensional car model: In the case of the non-holonomic two-dimensional model, we used "first person" inputs, that is, inputs translated and rotated so as to accomodate for the position and orientation of the car. (Note that the holonomic agent does not have an orientation.) The table-based controller had $3^3 * 5$, or 135 cells. Selection was done on the following three dimensions: speed, angle to the next waypoint and distance to the next waypoint. Cut-off values for these dimensions were set to 0.08, 0.3 and 0.31 respectively after an exhaustive search.

Likewise, the MLP-based controller takes speed, angle and distance to the next waypoint, and the action to evaluate as input, and outputs an estimate of the value of that action.

B. Temporal difference learning

On-policy temporal difference learning of action values is called Sarsa, and Sarsa(0) is defined by Sutton and Barto thus: [6]

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

Before this equation can be put to practical use, we must decide on values for the discount rate (γ) and the reinforcement regime. Both decisions greatly affect not only how fast learning happens but also whether anything is learnt at all.

In the experiments below, the reinforcement regime was that each time step, a reward of 1 was delivered if the car passed a waypoint, and 0 otherwise. The discount rate was set to 0.002. These parameters were arrived at through extensive

experimentation, but there is no guarantee that these are the optimal values. Initially, we tried to give a small negative reward in the timesteps when a waypoint was not passed, but we found the present setting to work better.

Another parameter is the exploration value, or the probability of the controller taking a random action instead of the greedy action with regard to its action-value estimates. This value is set to 0.1 during training, and set to 0 when evaluating the controller.

1) The one-dimensional model: Sarsa was moderately successful with the table-based function representation, but highly unreliable. Only about half of all runs ended up in a fitness above 20, and some ended at fitnesses close to 0, but when it did learn a good policy it did so quickly, usually within 1000 episodes (evaluations). The best controller out of 50 runs had fitness 54.6.

With a neural network-based function representation, Sarsa-learning fared much worse. The best controller found scored a paltry 3.1. There seemed to be intermittent unlearning involved during the learning process, with the fitness of the controller sometimes fluctuating wildly between episodes, occasionally peaking around score 20 only to drop to one or two in the next episode. Contrary to our expectations, lowering the learning rate did not resolve this issue.

2) The two-dimensional holonomic model: All our attempts at learning behaviour in the holonomic model were unsuccessful; not a single controller with fitness of at least 1 was produced this way, neither with the table-based nor with the MLP-based representation.

3) The two-dimensional car model: TDL with the table-based representation was as unreliable as in the one-dimensional model, but it did at least in some cases learn controllers with performance significantly better than random. Still, the best-performing learned controller had a fitness of only 4.7. The training runs that succeeded did so within 1000 epochs. In contrast to the td-learned controllers for the one-dimensional case, which performed better when occasional random movement was turned off (epsilon set to 0) after training, the controllers learned for the car model needed a (small) non-negative epsilon in order to perform.

TDL with the MLP-based representation was unable to learn any meaningful policy at all.

C. Evolution

We used a 15+15 evolution strategy to optimise the (state, action) to value mappings. For the both the connection weights of the neural networks and the values in the cells of the tables, we used Gaussian mutation with standard deviation 0.1.

1) The one-dimensional model: The table-based function representation turned out to be eminently evolvable, with every single run producing a good policy within a few generations. However, little progress was made beyond the first 50 generations, despite many runs of 500 generations being made. The best controller found using this method and representation scored 71.2.

The neural network-based function representation could also be used to evolve good controllers, but not as good as the table-based representation. The best controller produced after 500 generations had fitness 45.5.

2) The two-dimensional holonomic model: We were unable to evolve good action value functions for the holonomic agent, just like we were unable to learn them with td-learning. The best evolved table-based controller had fitness 0.3 and the best evolved MLP-based controller scored 0.1.

3) The two-dimensional car model: Evolution reliably produced somewhat capable action value function based controllers for the car model. The best evolved table-based controller (from several runs that produced similar controllers) scored 14.4, and basically drives well, apart from the occasional case of "orbiting".

Very similar results were achieved using the MLP-based representation, with the best evolved controller scoring 15.7.

D. Combining evolution and temporal difference learning

As we have seen, evolution and temporal difference learning can both be used to learn functional (state, action) to value mappings, although with varying speed, reliability and ultimate success. Another interesting question is whether the two methods learn different solutions to the problems. Especially, it might be suggested that while Sarsa learns *approximately correct* action values, evolution will learn *values that work*, regardless of whether they are correct or not. To investigate this, we tried applying our evolutionary algorithm to solutions that had been learned with Sarsa, and Sarsa to solutions that had been evolved.

1) The one-dimensional model: Using the table representation, neither further evolving a learned controller or further learning an evolved controller made any significant difference at all. The fitness of the best TDL-trained controller was around 54, both before and after 500 generations of evolution. Conversely, the fitness of the best evolved controller was around 71 during thousands of epochs of further td-learning. In other words, Sarsa performs much better than it usually does if initialised with an evolved controller, while evolution performs much worse if initialised with a Sarsa-trained controller. There seems to be a sort of lock-in effect.

In case of td-learning on top of evolution, the same effect was found for the neural network representation: no change. But in case of evolution on top of td-learning, fitness increased from 3.1 to 68.8.

2) The two-dimensional holonomic model: The performance of the action value based holonomic controllers is abysmal whether they are learned or evolved, and no significant changes were observed when one method was applied upon the results of another.

3) The two-dimensional car model: Seeding the Sarsa algorithm with evolved action value based car controllers had just the same effect as it had in the one-dimensional model: the further td-learning made no significant change, neither negative or positive. Or in other words, td-learning performs better when initialised with an evolved controller. Seeding

evolution with the results of Sarsa made no significant difference to seeding evolution with random controllers: good controllers evolved just as quickly.

VIII. Learning controllers directly

The above approaches learn controllers that are based on some sort of explicit representation of the values of states or actions. These are, to our knowledge, the only types of controllers that can be learnt using temporal difference learning. But evolutionary algorithms are also capable of solving reinforcement learning through direct search in policy space, creating controllers that don't necessarily directly represent state or action values. In case of point-to-point car racing, such controllers would take (aspects of) the state of the car as inputs and output the desired action to take. We chose to represent the controllers as standard three-layer neural networks with *tanh* activation function, and used the same Evolution Strategy as above.

A. The one-dimensional model

For the one-dimensional model, the inputs to the controller were the speed of the car, the position of the car, and the positions of the current and next input. Four hidden neurons were used. The single output was interpreted thus: if above 0.1, move right; below -0.1, move left; otherwise no action. This configuration reliably produced good controllers within 100 generations. Many runs of 500 generations each were done, and they all produced controllers of similar fitness, in the range 77.0-77.3.

B. The two-dimensional holonomic model

In the two-dimensional model, the neural network had seven inputs: the current velocity in the vertical dimension, current velocity in the horizontal dimension, difference between current position and those of the current and next waypoints in both dimensions, and a bias. Ten hidden neurons were used, and the two outputs were interpreted as movement commands like in the one-d example, with one output controlling horizontal and the other vertical movement.

Several modifications of the action selection mechanism (including interpreting the outputs as desired speeds) and the input representation (including rotation and translation according to the position and heading of the agent) were tried, as well as minor changes to the fitness function. Despite all this, we were unable to evolve good controllers for the holonomic model. The best controllers have fitnesses between 1 and 1.5, and they typically perform long trajectories around the game area, only occasionally hitting a waypoint.

C. The two-dimensional car model

Here, we fed the neural network with the velocity in vertical and horizontal dimensions, the magnitude of the velocity vector, and the distance and relative angle to the next waypoint. The output was interpreted as in the holonomic model, but unlike the holonomic model good controllers reliably emerge from the evolutionary process. Several runs of 500 generations each produced controllers of similar fitness, the best of them scoring on average 20.1. This controller consistently keeps a high speed and only rarely misses a waypoint, and then only "orbits" briefly.

IX. Results Summary

Based on our initial experiments with all of the above methods, we then tested some of them more exhaustively. For these tests, we ran each experiment (corresponding to a single row of each table) twenty times. Then, for each learned controller, we tested it on 100 tracks that were unseen during training. This methodology was chosen since the result of any single learning experiment can be highly variable.

A. One-D Results

Both evolution and temporal difference learning were able to learn good controllers for this simple version of the car racing problem. Evolution was more reliable and produced better end results, while td-learning learned faster.

B. Car Results

Results for the normal car are shown in table I. Each of these results used the full three-input feature vector. The evolved perceptron significantly outperforms the hand-coded heuristic, while the best method is the state-based MLP with a mean score of 35.0. Both the action-perceptron and the action-MLP achieved similar poor scores - just the action percptron is shown here, with a mean score of 1.8.

TABLE I
Mean of 20 learning runs for the normal car.

Method	Mean	s.e.
EVO-State-MLP	35.0	0.24
EVO-State-Perceptron	30.2	0.33
TDL-State-Perceptron	26.2	1.4
Hand-State-Heuristic	18.8	1.0
EVO-Action-Perceptron	1.8	1.3

While the reliability of a method is important, sometimes only the best result counts, and we have therefore also included the fitness of the best controller obtained through all of the different methods, tested on 1000 tracks, for the normal car in two dimensions (see table II).

TABLE II
Best controllers found for the normal car, tested on 1000 tracks.

Method	Td-learning	Evolution
State-MLP	-	36.7
State-Perceptron	31.1	31.5
Action-Table	4.7	14.4
Action-MLP	0	15.7
Direct-MLP	-	20.1

C. Holonomic Car Results

Here, for the learned networks, we used both the two and three input feature vectors, as indicated by a 2 or a 3 after the named method. The results are shown in table III, but only show results for evolution; TDL did not work well for any of

the holonomic state controllers, rarely exceeding an average score of 7.0. TDLs best result on a single epoch during learning was 26.0, which occurred on the 10th epoch of a particular run. By the end of that run, this good behaviour had been unlearned however, and the average score of the last twenty epochs was only 7.0 (approx). Note that the third input of the directional difference feature makes the problem significantly harder to learn, both for the Perceptron and the MLP, but it makes the MLP significantly worse, with a mean performance of 11.3. The three-input perceptron is less affected, with a mean score of 18.5. Removing the third input takes the Perceptron up to a mean score of 26.7, approximately equal to the heuristic controller, but both these are surpassed by the 2-input MLP, which has a mean performance of 32.2. The fact that these networks are *so* debilitated by an extraneous input came as a surprise to us, though given many more generations, evolution might learn to ignore this.

TABLE III
THE RESULTS FOR THE HOLONOMIC CAR.

Method	Mean	s.e.
State-MLP-2	32.2	0.06
State-Perceptron-2	26.7	0.01
State-Perceptron-3	18.5	2.1
State-MLP-3	11.3	0.7
State-Heuristic-2	26.8	0.21

X. CONCLUSIONS

Experiments were made on a variety of problem setups using different feature vectors, and different neural networks (multi and single layer perceptrons). In each case, state-value learning worked much better than action-value learning. Evolution worked more reliably and achieved better final fitness than reinforcement learning, but reinforcement learning, when successful, learned faster. Further, for learning action value functions with TDL, table-based representations were always superior to MLP-based representations. When evolution was used without a model, the direct approach was superior to learning action values. Our suspicion that the method learns differently and not only better or worse was supported by the "lock-in" effect when combining them.

The task is clearly very sensitive to the learning method, the architecture, and the chosen input features, especially the two-dimensional normal car problem. A strong distinction can be seen between methods which work very well, such as evolved state-based MLPs, and those that work very badly, such as TDL-trained action-based MLPs. While it is likely that with more work it would be possible to improve the TDL results, a great strength of the evolutionary methods is the ease with which they can be applied.

The state methods can only be applied when a forward model exists, which is able to predict the next state of the car given the current state and the selected action. This is simple for the simulation, where we had access to the exact forward model. However, for the real-world car racing, such is the advantage of the state-based controller, that this makes it a high priority to learn a forward model from observing the controller actions, if such a model is not provided.

The results in this paper have only addressed single-player maximum distance racing (i.e. how far can one travel given a fixed number of time steps), but in future we shall address multiplayer racing with car collisions.

Finally, we note that to get good performance it was necessary to transform the raw inputs of waypoint positions into some car-centric measures, in particular, the distance to the next waypoint, and the square of the velocity. This is in agreement with similar results for the full car racing problem [9]. We suggest that there is an important role for genetic or evolutionary programming here, to pre-process the raw input data, and then use a combination of RL and Evolution to evolve the value function based on the pre-processed data.

ACKNOWLEDGMENTS

We are grateful to the members of the Natural and Evolutionary Computation Group at the University of Essex, and to Thomas P. Runarsson for useful discussions related to this work.

REFERENCES

[1] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," in *Proceedings of the Twenty-first International Conference on Machine Learning*, 2004.

[2] B. Chaperot and C. Fyfe, "Improving artificial intelligence in a motocross game," in *IEEE Symposium on Computational Intelligence and Games*, 2006.

[3] D. Floreano, T. Kato, D. Marocco, and E. Sauser, "Coevolution of active vision and feature selection," *Biological Cybernetics*, vol. 90, pp. 218–228, 2004.

[4] S. M. Lucas and T. P. Runarsson, "Temporal difference learning versus co-evolution for acquiring othello position evaluation," in *IEEE Symposium on Computational Intelligence and Games*, 2006.

[5] T. P. Runarsson and S. M. Lucas, "Co-evolution versus self-play temporal difference learning for acquiring position evaluation in small-board go," *IEEE Transactions on Evolutionary Computation*, vol. 9, pp. 628 – 640, 2005.

[6] R. Sutton and A. Barto, *Introduction to Reinforcement Learning*. MIT Press, 1998.

[7] I. Tanev, M. Joachimczak, H. Hemmi, and K. Shimohara, "Evolution of the driving styles of anticipatory agent remotely operating a scaled model of racing car," in *Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC-2005)*, 2005, pp. 1891–1898.

[8] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney, "Winning the darpa grand challenge," *Journal of Field Robotics*, 2006, accepted for publication.

[9] J. Togelius and S. M. Lucas, "Evolving controllers for simulated car racing," in *Proceedings of the Congress on Evolutionary Computation*, 2005.

[10] ——, "Arms races and car races," in *Proceeding of Parallel Problem Solving from Nature*. Springer, 2006.

[11] ——, "Evolving robust and specialized car racing skills," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2006.

[12] J. Togelius, R. D. Nardi, and S. M. Lucas, "Making racing fun through player modeling and track evolution," in *Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games*, 2006.