Progressive Neural Network Training For The Open Racing Car Simulator

Christos Athanasiadis, Damianos Galanopoulos, and Anastasios Tefas

***Abstract*—In this paper a novel methodology for training neural networks as car racing controllers is proposed. Our effort is focused on finding a new fast and effective way to train neural networks that will avoid stacking in local minima and can learn from advanced bot-teachers to handle the basic tasks of steering and acceleration in The Open Racing Car Simulator (TORCS). The proposed approach is based on Neural Networks that learn progressively the driving behaviour of other bots. Starting with a simple rule-based decision driver, our scope is to handle its decisions with NN and increase its performance as much as possible. In order to do so, we propose a sequence of Neural networks that are gradually trained from more dexterous drivers, as well as, from the simplest to the most skillful controller. Our method is actually, an effective initialization method for Neural Networks that leads to increasingly better driving behavior. We have tested the method in several tracks of increasing difficulty. In all cases the proposed method resulted in improved bot decisions.**

I. Introduction

Nowadays, video games are becoming more and more important, as a hot consumer product, as well as a great opportunity for research in artificial intelligence. The main goal is to offer fun to the player. In previous years this goal has been achieved partly through the visual realism and interesting game scenarios. But every video-game player knew that the current AI in the games was way far from the actual human behaviour. When we are playing a game versus one or more NPC (non-playable character) we can easily realize that we are not playing versus another human because either the other player is too simple to beat, figuring out a specific efficient strategy, or the AI is so complicated that the human loses every time. Artificial intelligence in computer games is infused into NPCs with a view to giving the human player the illusion of a clever human opponent. Initially we have to create a NPC that imitates the behaviour of a human player [1].

However, we have to bear in mind that the NPC must have the ability to adapt depending on the current state and environment and the current opponents in the game. Computational intelligence methods can be implemented to deal with the adaptation task. Such methods can be retrieved from evolutionary algorithms and Neural Networks [2].

Previous approaches to car racing were already developed for the forerunner of TORCS, the robot auto racing simulator (RARS) [3]. For example, Stanley [4] developed a car racing strategy that depended on range-finders and developed a sensory-motor mapping with the incremental neural evolution of augmenting topologies (NEAT) approach. The most approaches are developed for TORCS, some of them are the following.

Julian Togelius and Simon M. Lucas proposed a neural network using evolution both on sensor parameters and neural networks weights . The fitness that is used is calculated as the number of waypoints it has passed, divided by the number of waypoints in the track, plus an intermediate term representing how far it is on its way to the next waypoint. [5]. Moreover, an examination on various versions of Genetic Programming and a comparison against Artificial Neural Network for car racing controller is performed in [6].

The Cognitive BOdySpaces for Torcs-based Adaptive Racing (COBOSTAR), which was developed by M. V. Butz and T. D. Lonneker [7], is divided in two parts: on-track optimization and off-track optimization. Actually, they implemented heuristic functions for mapping input data into decision. These functions were different when the controller was on-track or off-track.

Another approach has been proposed in [8]. The idea behind this bot was to have a driving architecture based on a set of simple controllers. Each controller is applied as a separate module in charge of a basic driving action. Two important modules are the learning module, which finds where the bot has to increase or reduce its speed, and the opponents management module, which adapts the agent behaviour when the opponents are close.

Cardamone's, Loiacono's and Lanzi's approach [9] consists of an evolved neural network, implementing a basic driver behaviour, compounded with code for basic tasks such as the start, the crash-recovery, the gear change, and the overtaking. They use neuro-evolution of augmented topologies(NEAT)[4] with aim to predict target speed and target position for a given input configuration. The implemented fitness functions is the error between the actual values and the predicted ones.

The combination of a fuzzy logic module with a classifier module and a finite state machine is proposed [10] with a view to tackling a variety of TORCS commands. In [11] a study of imitation of controllers behavior is represented. In this approach a human controller, a rule-based controller and a controller based in learning methods is used for data extraction. An artificial neural network is trained with back-propagation algorithm. The decisions that ANN had to approximate was accelerate, break, wheel steering and controllers gear. A human-like controller is proposed on [12], where the author develop a system that learns track's morphology. A

C. Athanasiadis, D. Galanopoulos, and A. Tefas are with the Department of Informatics, Aristotle University of Thessaloniki 541 24, Greece; (emails:chrathan@csd.auth.gr, dgalanop@csd.auth.gr, tefas@aiia.csd.auth.gr).

review paper that contains most of previous approaches can be found in [13][14].

The basic idea in the proposed method, is to use Neural Networks for the decisions of the driver bot. That is, the bot is a single hidden layer NN trained using back-propagation. The training data are collected using other controllers that are utilizing different policies to control the car. That is, the proposed bot is trained by other bots. We expect to enhance the performance of the initial controllers since the NN will smoothly imitate their behaviour. Moreover, in order to improve its performance during the learning phase, we implement a learning chain architecture, which is in fact a sequence of Neural Networks. The weights of these NN are initialized from the previous steps.

The major problem that the proposed approach tries to solve is the difficulty to train Neural Networks in complex problem solving that usually involves a priori knowledge of the human creators encoded in rule-based systems. That is, the best NPC are very complex and take contradictory decisions in very similar situations in order to perform on the limit. When someone tries to train neural networks straight to the data produced by these complex bots he will find that even if in terms of MSE (Mean Squared Error) the NN obtains a very low value, when it comes to real driving in specific tracks the performance is very poor. This fact can be attributed in overtraining or stuck in local minima. The proposed methodology is to use simpler drivers as first trainers and then to proceed in training with more complex drivers. That is, the NN-bot will learn first to stay in the track and follow smooth driving and then to drive on the limit.

The first step of the proposed approach is the creation of a NN from data obtained from the Simple Driver which is the simplest controller provided with java client. The output weights are used as initialization weights in another NN trained from extracted data from another controller, instead of random initialization. We follow the same policy for a sequence of Neural Networks designed from data from another two controllers. We propose a progressive learning method via initialization of the weights with past experience from other simpler bots. Weight initialization is a state of the art problem in feed-forward neural networks which affects the detection of the global minimum solution, the speed of the convergence, the successful convergence of the NN and also the ability of generalization. In contrast with other learning approaches in TORCS, we did not give effort implementing neural networks to best encode human driving knowledge directly. Instead, we propose a training methodology inspired by the human practice of proceeding to more advanced teachers when you assimilate the knowledge of the current teacher. This is obtained by imitating the other controllers behavior solving the NN weights initialization task by using our learning-chain architecture.

II. TORCS ENVIRONMENT

The open race car simulator (TORCS) provides an open source car racing environment with a very realistic simulator that has a sophisticated physic engine which takes into consideration real car racing issues such as fuel consumption, collisions or traction. Besides that, TORCS offers a very realistic game-play and graphics. It is a well designed simulator which can be compared with the finest race game titles. Additionally, TORCS competition API [15] provides a very flexible and simple client-server architecture. Server stands for the game's functions, and client stands for the car agent handling. The above characteristics justify why it has been used for research purposes in the scientific community, especially for solving the simulated racing car challenge task. In 2011, three different challenges were held; the EVO-2011 in Torino, ACM GECOO-2011 in Dublin and the IEEE CIG-2011 in Seoul.

The first approaches appeared, were rule-based and only slightly optimized on several aspects. Our approach in this challenging task is based on Neural networks architecture of sequential neural networks which is utilized with view to initialize the next sequence. Beginning with a simple driver, we hope to progressively imitate the behavior of the 2009 CIG champion. The proposed controller is based on a feed forward ANN (Artificial Neural Network) that was trained with data generated by several controllers using back-propagation.

TORCS competition API consists of a server component that supports the general TORCS set-up [16] and returns information sensors about the controller and the track. The client component uses these information to apply its strategy. The controllers (clients) run as external programs and communicate with the server. At each game tic the controller receives sensor data that correspond to the car's current state and its surrounding environment (the tracks and the components). The controller has to calculate four output parameters (the wheel steering, the gas pedal, the fuel level and the break pedal). Its strategy depends on the current input (information from sensors). In the proposed method we use learning methods in order to build output commands.

The sensors novelty is that they do not contain the whole track information, but they carry only simulated local information instead. In particular, the available sensors are an angle sensor, which specifies the current angle between the car direction and the track axis, the current speed in longitudinal and transverse axes of the car, 19 range sensors, which sample the free track space in front of the car and they are only valid while on track, 36 opponent sensors, which notice opponents around the car, the current engine speed in rounds per minute, the current gear, the track position with respect to the track edges, and the current rotation speed of the four wheels. Moreover, there is further racing information available including the current lap time, the damage of the car, the distance from the start line along the track line, the total distance raced, the amount of remaining fuel, the last lap time and the standing in the race. For the car control, there is a gas pedal and a brake pedal, gear shifting, and steering values available.

Thus, the controller's strategy cannot receive information about tracks morphology (such as which curve comes next) and it depends only on the local information (the sensors they were described before).

III. Problem statement

It is well known that the training of a NN can be viewed as the optimization (minimization) of the error with respect to the weights. The particular local minimum will define the quality of the Neural Network in terms of learning the specific training sample an in terms of its behavior in unknown input (generalization ability). If the minimum is close to the global solution the performance will be acceptable and the training successful in terms of MSE. On the other hand, there are minima that result in poorly trained networks in terms of generalization. The two important factors that influence the final solution are the weights initialization and the algorithm that is used. Moreover the weights initialization determines the speed of convergence, the probability of convergence and the generalization. Thus it is clear that it is important part of the Neural network. However, the usual initialization approach is to use a random number generator to produce the weights. It seems defective that we leave an important fact at random especially when the optimization task is rather complex. Weight initialization has been widely recognized as one of the most effective approaches in speeding up the training of neural network.

Weight initialization techniques have been extensively examined in feed-forward neural networks. Some approaches rely on defining an optimal distribution and range for the weights either empirically or based on characteristics of the neurons, Zhang and Ciesielski proposed a centred initialization approach for solving the object detection task. Instead of using the classical random approach, they place the highest initial weights at the center of the input field. Weights values decrease uniformly to the perimeter [17]. Several approaches, [18], were compared in 8 benchmark problems and it was concluded that the method proposed in [19] is the most effective. Moreover, a number of techniques rely on either neglecting the scaling effect or approximating the sigmoidal activation function to obtain least squares based techniques. Our method rely on empirically based approaches. We try to give a quick and effective solution for the initialization problem based on the fact that the experience of previous controllers, will lead to a fast and accurate learning process.

Our architecture is based on Multi-Layer-Perceptron Networks. For MLP architecture we define $\mathbf{z^l}$ and $\mathbf{y^l}$ the output of the l^{th} layer before and after the use of the activation function. The weight matrix and bias $\mathbf{W^l}$ and $\mathbf{b^l}$ for the l^{th} layer. Let $\mathbf{x}$ be the input vector. The number of the neurons in each layer is denoted by n_l and the size of the input vector is n_0. The training data is composed by input data and the desired output (x_t, d_t^L). We wish the neural network output to approximate as much as possible the desired output (training label) [20].

Assuming that we have a Neural Network with one hidden layer. We can represent the NN with the above equation for the hidden layer(2_{nd} layer):

$$z_i(t) = f(\sum_j w_{1_{jk}}(t) x_j(t) + b_1(t)) \tag{1}$$

where f is the activation function which in our case is hyperbolic tangent, $\mathbf{w_1}, \mathbf{b_1}$ weights and bias in the hidden layer, $\mathbf{x}$ the input vector and the $\mathbf{z}$ the output of the neurons in the hidden layer. And secondly in the output layer equation (3_{rd} layer):

$$y_i(t) = f(\sum_j w_{2_{jk}}(t) z_j(t) + b_2(t)) \tag{2}$$

weights and bias are fixed during the forward and backward pass of back-propagation algorithm. Also we can say that general $z_l = W * z_{l-1} + b_l$ where z_0 correspond to input vector. The algorithm is revealed by the following equations:

$$E(t) = \frac{1}{2} \sum_{k=1} (d_k(t) - y_k(t))^2 \tag{3}$$

where d_k is the target value for dimension k. We want to know how to modify weights in order to decrease E. Using gradient descent:

$$w_{ij}(t+1) - w_{ij}(t) \simeq \frac{\partial E(t)}{\partial w_{ij}(t)} \tag{4}$$

both for hidden layers and output layers.

As previously mentioned, the initial weights are important in the learning convergence. If we could approximately choose the weights close to the global optimum, then the Back-propagation or any other gradient descent algorithm could take the networks weights toward the optimum fast and reliably. Thus, the suggested approach in solving the initialization issue is based on the observation that if we use weights which will result real-time in an output close to the desired output then we are close to the global solution. In our work we reached the conclusion that using a learning chain from Neural Networks trained with data from several existed controllers and using their weight to initialize each other with a specific order we can approximate in each NN faster and more robustly the desired output. By comparing the output that each input weight produces and the desired output of the controller we put the NN (corresponding to each controller) in a unique order. Our learning chain is consisted from 4 states. Each state corresponds to a NN. In each state we use different extracted data from different controllers. Besides the first state, each state feedback with the calculated weights the following state with view to initialize fast and accurate the next Neural Network.

Figure 1 illustrates the problem of finding the global solution. Assume that our scope is to approximate as much as possible Cobostar behavior. We can say that the Simple driver tries to solve Cobostar approximation task by giving a simple solution which leads to poor results and in not an acceptable solution. Simplicity with a more complex driving behavior is closer to the Cobostar behavior. Correspondingly Tiede and Ebner controller representation is the closest solution regarding the global solution which is in our case the Cobostar behavior. Hence, we can see the problem of initialization as finding a quick and a good local solution close to the global minima. That's the reason why we use weight initialization from Tiede and Ebner bot in order to approximate Cobostar. In the same line, simplicity is used to Tiede and Ebner approximation task as a quick and good initial solution. Finally, in the same manner we initialize Simplicity approximation using Simple driver and the initial NN that imitates Simple driver driving behavior using random initialization.

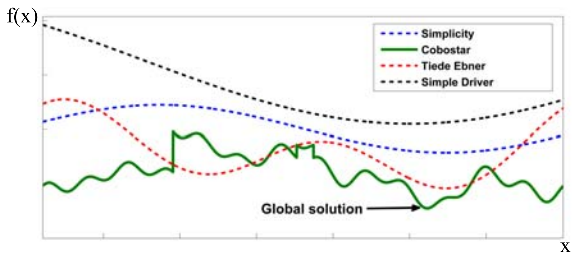

Fig. 1. Global minimum approximation.

IV. Progressive training

The first step towards learning from other AI bots is to build a training set that will be large enough in order to appropriately sample the input/output space. It is obvious that since the trainer is another bot we would like this bot to be one of the best. The objective then is to select a learning machine (Neural networks in our case) that will be able to be trained using the training set. Since the task is rather complex the danger of overtraining or premature convergence to local minima is evident. Indeed, we first tried to create a Neural Network based on data collected from Cobostar [7]. We have noticed that it is not feasible to direct learn Cobostar's behavior. This is happening since Cobostar is a sophisticated driver which takes conflicting decisions in sequential instances. Thus, a neural network with random weight initialization is prone to stuck in a poor local minimum solution. Instead, we propose the progressive training of the NN-bot using other AI bots from the simpler to the more complex in order to achieve increased generalization ability.

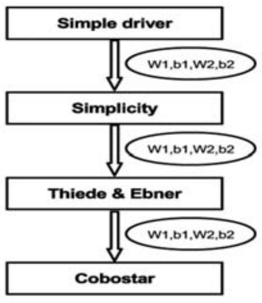

Fig. 2. Sequence of Neural Networks

The learning chain begins by obtaining data from four controllers. The first is the "Simple Driver" which is provided from java client, the second is the "Simplicity" which took part in CIG 2009 by Wong Ka Chung, the third is "Tiede and Ebner" which also took part in Torcs competition in CIG 2009 and finally the "Cobostar" [7] which was the champion of the same competition. We found that there is a correlation between the controller's desired output and the corresponding Neural Network approximations. The neural network output trained from Simple driver data is closer to Simplicity behavior and sequentially the NN-bot trained with Simplicity data is closer to Tiede and Ebner behavior and finally the closest controller to Cobostar is the NN-bot trained with Tiede and Ebner data. Therefore the above order is utilized in the proposed learning chain. This order is shown in Fig. 2. A related work can be found here [21] where the authors investigate how to transfer driving behavior between two games Vdrift and TORCS. We created an artificial neural network sequence with the above order, and utilized back-propagation for training. More sophisticated training algorithms may be also used but their comparison is not in the scope of this work. The neural networks used a specific structure with one hidden layer which is sufficient to approximate a bound continuous function with a specific mean square error [22], [23]. The number of neurons in the hidden layer is another critical parameter. The number of neurons was selected using cross-validation in the training set in order to avoid over-training. After experiments we found out that 50 neurons are enough to approximate the given functions and thus we have used this value in all experiments in order to focus only on the impact of the proposed progressive training approach. As an activation function for the NN we have used the hyperbolic tangent

$$f(t) = \frac{e^t - e^{-t}}{e^t + e^{-t}}. \quad (5)$$

The Neural Network weights determine the controller's decisions, which will define the behaviour of our bot in real-time gaming. Using the back-propagation algorithm we were able to train neural networks to return the desired output data. Thus, we can use the outputs of the trained neural networks in order to control the acceleration and steering of the car.

Neural network outputs take values in $[-1, 1]$. When it comes to acceleration we use a simple rule to divide the result to acceleration and break. When the output returns 1, it means that our controller has to fully accelerate and when it returns -1 controller has to fully break. The objective was to succeed convergence in bots behaviour (between the controller that was used as trainer and our agent).

The training parameters for every bot and the two architectures are shown in the tables II, III, IV, V for both decisions (acceleration and wheel steering). A technique to estimate the best learning parameters is implemented. Firstly, we keep a list of parameters which lead to minimum mean square error after grid search in the parameter space. Two tracks different than the ones used for the training data extraction are used for parameter evaluation. We keep the weights which lead to the best controller performance in these unknown tracks. Finally, the trained models are tested in another seven tracks which are not included in the training or evaluation set. The same procedure is followed for all the compared models.

The training data was received for each controller from three tracks fig. 5. For each track we obtained data for two laps. Instead of using all the data that the server of TORCS provide as sensoring data, we use only a subset. For each track we used 10.000 input states for each controller. Each input state is composed of 9 input data that represent the controller's environment and two output values that correspond to acceleration and wheel steering. These two outputs are in fact the desired outputs of the implemented Neural Networks. That is, we train one neural network that decides about the acceleration and the other decides about the steering. All the values are normalized in $[-1, 1]$. The input vector consists of the following sensors :

- Angle to track axis $[-\pi, \pi]$
- track position $[0, \pi]$
- speed $(0, +\infty]$
- Gear $[0, 6]$
- 7-11 Track edge sensors $[0, 100]$

These sensor outputs are considered as inputs to the proposed NN-bot and the NN-bot must decide and sent to the server the following two control values in range $[-1, 1]$:

- accelerate/break
- steering

Besides acceleration, break and steer the controller must sent to the server also the gear change decision. The gear change policy is hard coded and it is based on the received sensor RPM. The table I points out when the controller has to increase or decrease gear.

TABLE I
CONTROLLERS PERFORMANCES

Current Gear	Gear Up	Gear down
1	9200	0
2	9400	3300
3	9500	6200
4	9500	7000
5	9500	7300
6	–	7700

TABLE II
NEURAL NETWORK PARAMETERS FOR PROGRESSIVE INITIALIZATION, STEER DECISION

Method	lr Rate	mom	epochs	MSE	Duration (sec)
Simple Driver	0.001	0.01	20	0.000212	33.56
Simplicity	0.01	0.01	20	0.00323	36.93
Tiede Ebner	0.0011	0.02	20	0.0789	35.55
Cobostar	0.0006	0.01	20	0.0014	37.11

A. Learning chain architecture

The proposed learning chain architecture is a sequence of Neural Networks that are trained with data obtained from different AI controllers. The first NN is trained with the data collected from the simple driver with random weight initialization, the second NN is trained using data collected from the Simplicity and weight initialization the final weights

TABLE III
NEURAL NETWORK PARAMETERS FOR RANDOM INITIALIZATION, STEER DECISION

Method	lr Rate	mom	epochs	MSE	Duration (sec)
Simple Driver	0.001	0.01	20	0.000212	33.56
Simplicity	0.01	0.01	20	0.0189	38.66
Tiede Ebner	0.0001	0.01	20	0.098	35.04
Cobostar	0.0041	0.01	20	0.0021	38.01

TABLE IV
NEURAL NETWORK PARAMETERS FOR PROGRESSIVE INITIALIZATION, ACCELERATION DECISION

Method	lr Rate	mom	epochs	MSE	Duration (sec)
Simple Driver	0.1	0.1	20	0.000125	49.89
Simplicity	0.1	0.1	20	0.00332	54.65
Tiede Ebner	0.004	0.1	20	0.021	51.54
Cobostar	0.0005	0.01	20	0.09	53.01

of the previous NN, the third NN is trained using data from Thiede and Ebner and initialized using the previously obtained weights and finally the fourth NN is trained using data from the Cobostar and initialized using the weights obtained by the Thiede and Ebner data. That is, every NN uses the weights from the previous state. The final controller handled from neural networks tries to learn Cobostar's decisions. Using the proposed approach, the Neural network converges faster, and using the past experience from simpler bots enhances its performance. The standard alternative to the proposed approach is to try to train a NN directly using the training data obtained by the most sophisticated bot. Results show that the proposed algorithm is effective since it reduces the lap time obtained with the random initialization approach. Moreover, the generalization ability of the proposed method is illustrated in the experimental results Section. The results in unknown tracks reveal that our method is more effective in handling unknown situations.

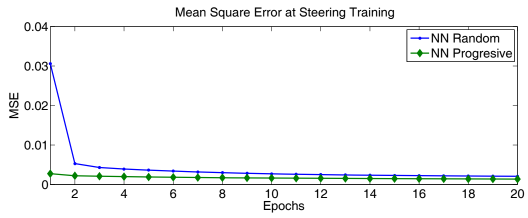

Fig. 3. Mean square error for acceleration.

V. EXPERIMENTAL RESULTS

In this Section several experiments that highlight the performance of the proposed NN-bot are presented. More specifically we are comparing the two NN architectures, with progressive and random weight initialization in 4 crucial factors: the quality of the solution, which is the actual performance of

TABLE V
NEURAL NETWORK PARAMETERS FOR RANDOM INITIALIZATION, ACCELERATION DECISION

Method	lr Rate	mom	epochs	MSE	Duration (sec)
Simple Driver	0.1	0.1	20	0.000125	49.89
Simplicity	0.01	0.01	20	0.00689	52.67
Tiede Ebner	0.0001	0.01	20	0.018	50.55
Cobostar	0.0041	0.01	20	0.19	53.12

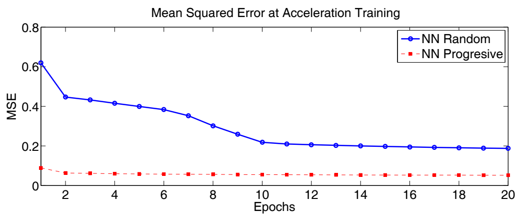

Fig. 4. Mean square error for steer.

the system in several tracks, the experimental convergence of our networks in learning process, the speed of the convergence and the generalization ability. To prove the superiority of the proposed method we have tested the two architectures with the best set of learning parameters which we find for both cases. Table VI displays the performances of controllers in several tracks with the best training parameters. The first comparison parameter which is the most important is the real performance of both controllers. The lap times in several tracks reveal the actual difference between the two architectures. It is obvious that the proposed approach attains performance that is very close to the trainer whereas the standard NN training approach fails to imitate the driving behaviour of Cobostar.

Additionally, figures of acceleration - track axis and wheel steering - track axis of the two controllers and the Cobostar trainer reveal the decision that the three controllers sent to the game server for the same track and the same lap. As we mentioned before we found the learning parameters with the referred method in the previous section. The second comparison will be the ability for a fast and accurate convergence. This will be reflected by the mean square error convergence that we have in the two Neural network architectures. Finally we will investigate the performance of two controllers in unknown tracks and extract useful conclusions about the generalization capability.

Tables VI, VII show the best lap time and the average time in several tracks, that are illustrated in Figure 6, for the three controllers, i.e., the Cobostar and the controllers using the two NN architectures. In all cases Progressive initialization architecture attains better lap times and average compared to random initialization architecture. In some tracks it is noticeable that our proposed architecture achieves lap times very close to the trainer's corresponding lap times.

Figures 7, 9, 11, 13 show the difference in acceleration decision between random network which is the neural network with random initialization and progressive network which

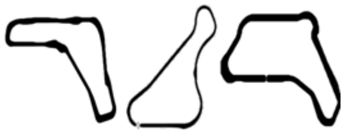

Fig. 5. Tracks from which we obtained the training data.

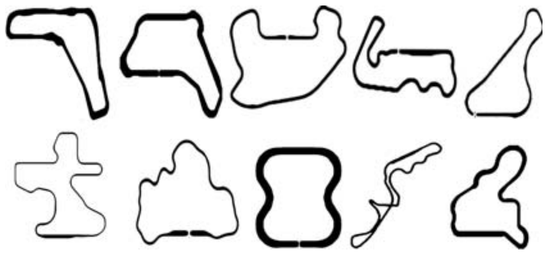

Fig. 6. Tracks of the game, used for testing generalization capability.

is the controller implemented with our proposed method. Respectively figures 8, 10, 12, 14 visualize the differences in wheel steering decision between the two compared controllers. More specifically, we plot the speed-distance from start line decision and distance from track axis-distance from start line decision. It is obvious that the progressive network succeeds a more smooth trajectory in contrast to the random network. This is figured in all distance from track axis-distance from start line plots. In the same line, it is clear that progressive network behavior in case of speed resembles the trainer's behavior more than the behavior of the random network. Both controller's results correspond to their best lap time in the specific tracks. It is obvious, in both cases that our controller manage to keep a trajectory close to that of Cobostar. On the other hand, random initialized controller takes decisions that are rather different in some cases compared with the decisions of the Cobostar controller.

The quality and the speed of convergence is another comparing factor of both architectures. The mean square error of neural network convergence is presented in figures 3, 4. These MSE is extracted from neural networks which are trained from Cobostar data and with the best learning parameters found. We can notice that in both cases of steering and acceleration, progressive network succeeds to converge faster obtaining smaller MSE.

Last but not least is the comparison of the generalization ability. This test is actually the results of both architectures in unknown tracks, from which we did not extract learning data. We can see from the performance Tables VI, VII that the difference in best and average lap times between the two architectures increases in the seven unknown tracks. This leads us to the conclusion that our method is more robust in unknown situations.

VI. CONCLUSIONS

In this work we proposed a novel method of Neural Network training for The Open Car Racing Simulator. In our learning chain architecture we propose a learning sequence which

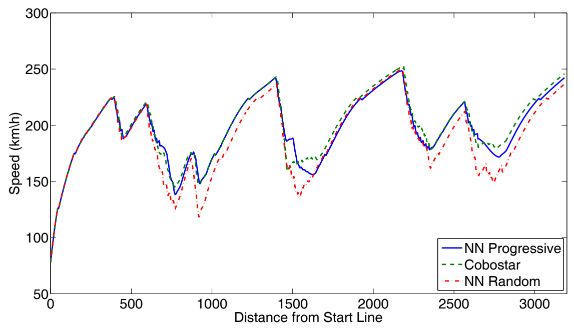

Fig. 7. Speed results in CG track2.

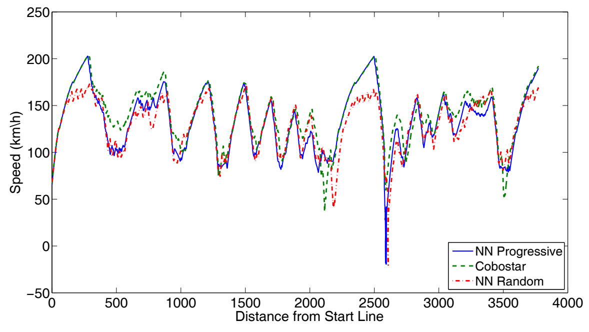

Fig. 11. Speed results in CG Alpine2.

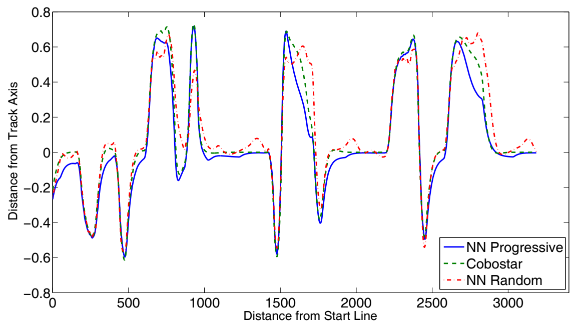

Fig. 8. Steer results in CG track2.

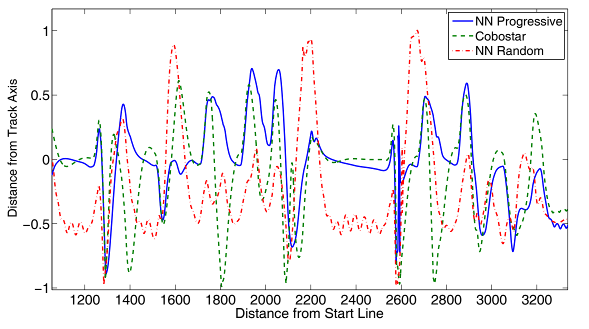

Fig. 12. Steer results in Alpine2.

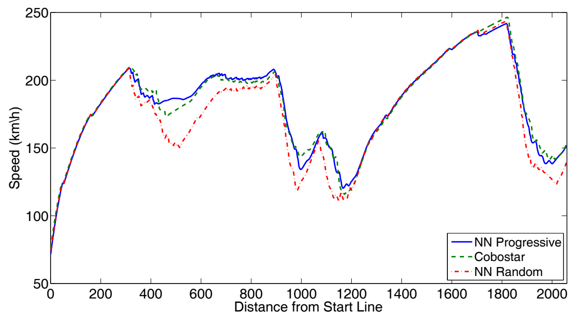

Fig. 9. Speed results in CG track1.

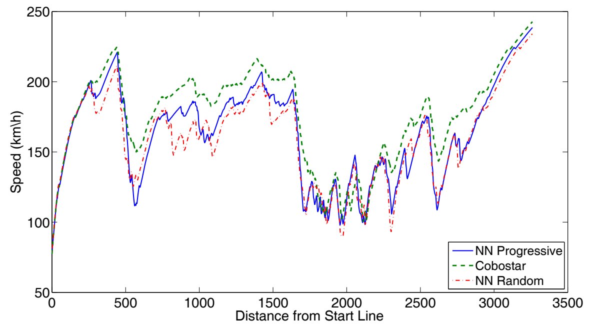

Fig. 13. Speed results in CG Alpine2.

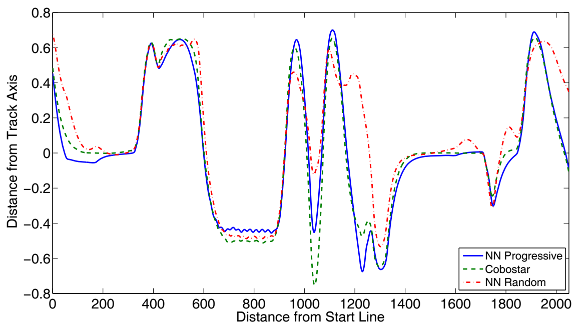

Fig. 10. Steer results in CG track1.

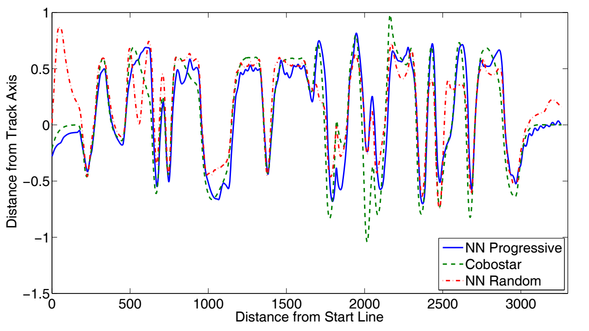

Fig. 14. Steer results in Alpine2.

TABLE VI
BEST LAP TIMES FOR THREE CONTROLLERS

Track/controller	Cobostar	Progressive	Random
CG Speedway1	38.66	39.51	42.26
CG Speedway2	54.03	56.58	59.57
Forza	99.12	102.43	103.89
CG Speedway3	71.42	74.25	79.99
Wheel1	80.28	88.42	101.52
Wheel2	123.14	135.36	155.26
Etrack4	108.81	121.54	127.76
Alpine2	99.98	106.37	119.02
Eroad	63.86	66.54	67.61
Etrack5	26.88	28.73	28.82

TABLE VII
AVERAGE LAP TIMES FOR THREE CONTROLLERS

Track/controller	Cobostar	Progressive	Random
CG Speedway1	41.05	41.64	43.12
CG Speedway2	55.67	57.85	61.23
Forza	102.91	103.62	104.81
CG Speedway3	74.01	77.53	81.19
Wheel1	82.37	90.09	119.64
Wheel2	129.60	142.60	157.74
Etrack4	111.81	124.54	130.76
Alpine2	102.05	112.37	123.13
Eroad	66.86	71.54	72.61
Etrack5	28.01	31.14	36.21

consists of several steps. At each stage the proposed NN-bot is trained using data obtained from a more sophisticated controller. This corresponds to progressive training with more skilled trainers. The connection between the training stages is the use of the weights obtained from the previous stage that encode the knowledge of the previously used trainers. The experimental results highlight that the proposed method managed to attain superior performance compared to the classical method of random initialization at each stage. The criteria of comparison are based on the real-time driving performance, the ability of effective and fast convergence in learning phase and the capability of generalization which is in our case the ability of the controller to take decisions in unknown tracks. That is, the proposed approach offers an efficient methodology for NN training when training data of progressive complexity for the same task can be obtained.

REFERENCES

[1] J. E. Laird and M. van Lent, "Human-level AI's killer application: Interactive computer games," in *Proceedings of the 17th National Conference on Artificial Intelligence*. AAAI Press / The MIT Press, 2000, pp. 1171–1178.

[2] S. Petrakis and A. Tefas, "Neural networks training for weapon selection in first-person shooter games," in *Proceedings of the 20th international conference on Artificial neural networks: Part III*, ser. ICANN'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 417–422.

[3] "Robot auto racing simulator. available http://torcs.sourceforge.net,," 2006.

[4] K. Stanley, N. Kohl, R. Sherony, and R. Miikkulainen, "Neuroevolution of an automobile crash warning system," in *Proceedings of the 2005 conference on Genetic and evolutionary computation, Washington DC*, ser. GECCO '05. ACM, 2005, pp. 1977–1984.

[5] J. Togelius and S. Lucas, "Evolving robust and specialized car racing skills," in *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, 2006, pp. 1187 –1194.

[6] A. Agapitos, J. Togelius, and S. M. Lucas, "Evolving controllers for simulated car racing using object oriented genetic programming," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation, London*, ser. GECCO '07. ACM, 2007, pp. 1543–1550.

[7] M. V. Butz and T. D. Lönneker, "Optimized sensory-motor couplings plus strategy extensions for the torcs car racing challenge," in *Proceedings of the 5th international conference on Computational Intelligence and Games, Milano*, ser. CIG'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 317–324.

[8] E. Onieva, D. A. Pelta, J. Alonso, V. Milanés, and J. Pérez, "A modular parametric architecture for the torcs racing engine," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium, Milano*, sept. 2009, pp. 256–262.

[9] L. Cardamone, D. Loiacono, and P. Lanzi, "Learning drivers for torcs through imitation using supervised methods," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium, Milano*, 2009, pp. 148 –155.

[10] D. Perez, Y. Saez, G. Recio, and P. Isasi, "Evolving a rule system controller for automatic driving in a car racing competition," in *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium, Perth Australia*, dec. 2008.

[11] J. Muñoz, G. Gutierrez, and A. Sanchis, "Controller for torcs created by imitation," in *Proceedings of the 5th international conference on Computational Intelligence and Games, Milano*, ser. CIG'09, 2009, pp. 271–278.

[12] J. Muñoz, G. Gutiérrez, and A. Sanchís, "A human-like torcs controller for the simulated car racing championship," in *Computational Intelligence and Games (CIG), 2010 IEEE Symposium, Copenhagen*, aug. 2010, pp. 473–480.

[13] D. Loiacono, P. Lanzi, J. Togelius, E. Onieva, D. Pelta, M. Butz, T. Löandnneker, L. Cardamone, D. Perez, Y. Sáez, M. Preuss, and J. Quadflieg, "The 2009 simulated car racing championship, milano," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 2, no. 2, pp. 131 –147, june 2010.

[14] D. Loiacono, J. Togelius, P. L. Lanzi, L. Kinnaird-Heether, S. M. Lucas, M. Simmerson, D. Perez, R. G. Reynolds, and Y. Sáez, "The WCCI 2008 simulated car racing competition," in *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, 2008, pp. 119–126.

[15] D. Loiacono, L. Cardamone, and P. L. Lanzi, "Simulated car racing championship 2009: Competition software manual," Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milan, Italy, Tech. Rep., 2009.

[16] "Torcs the open race car simulator. available: http://torcs.sourceforge.net."

[17] M. Zhang and V. Ciesielski, "Centred weight initialization in neural networks for object detection," in *Proceeding of the 22nd Australian Computer Conference*. Springer, 1999, pp. 39–50.

[18] C. Hernandez-Espinosa and M. Fernandez-Redondo, "Multilayer feedforward weight initialization," in *Neural Networks, Proceedings. IJCNN '01. International Joint Conference on*, vol. 1, 2001, pp. 166 –170 vol.1.

[19] L. Wessels and E. Barnard, "Avoiding false local minima by proper initialization of connections," *Neural Networks, IEEE Transactions on*, vol. 3, no. 6, pp. 899 –905, nov 1992.

[20] D. Erdogmus, E. Fontenla-Romero, and R. Jenssen, "Accurate initialization of neural network weights by backpropagation of the desired response," in *Neural Networks, 2003. Proceedings of the International Joint Conference on*, vol. 3, july 2003, pp. 2005 – 2010 vol.3.

[21] L. Cardamone, A. Caiazzo, D. Loiacono, and P. L. Lanzi, "Transfer of driving behaviors across different racing games," in *Computational Intelligence and Games (CIG), 2011 IEEE Conference on, Seoul*, 2011, pp. 227–234.

[22] S. S. Haykin, *Neural networks and learning machines, Multi Layer Perceptron pp. (152-258) 3rd Edition*, 2008.

[23] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural Networks, 1989. IJCNN., International Joint Conference on*, 1989, pp. 593 –605 vol.1.