Learning to Drive a Real Car in 20 Minutes

Conference Paper · November 2007

DOI: 10.1109/FBIT.2007.37 · Source: IEEE Xplore

Martin Riedmiller
University of Freiburg
192 PUBLICATIONS **18,782** CITATIONS
SEE PROFILE

Hendrik Dahlkamp
Stanford University
15 PUBLICATIONS **4,461** CITATIONS
SEE PROFILE

Learning to Drive a Real Car in 20 Minutes

Martin Riedmiller
Neuroinformatics Group, Univ. of Osnabrueck
Email: martin.riedmiller@uos.de

Mike Montemerlo, Hendrik Dahlkamp
AI Lab, Stanford University
Email: {montemerlo, dahlkamp}@stanford.edu

Abstract

The paper describes our first experiments on Reinforcement Learning to steer a real robot car. The applied method, Neural Fitted Q Iteration (NFQ) is purely data-driven based on data directly collected from real-life experiments, i.e. no transition model and no simulation is used. The RL approach is based on learning a neural Q value function, which means that no prior selection of the structure of the control law is required. We demonstrate, that the controller is able to learn a steering task in less than 20 minutes directly on the real car. We consider this as an important step towards the competitive application of neural Q function based RL methods in real-life environments.

Figure 1. The car used is a VW Passat, equipped with additional sensors.

1 Introduction

The interest in applying Reinforcement Learning (RL) methods to real life control applications is growing rapidly e.g. [7], [14], [9], [5]. In this paper we focus on situations, where the controller should learn by interacting with the real system only. In particular, for the design of the controller we will not assume, that a system model is available; neither in form of system equations nor in form of a simulator (the latter approach was successfully applied in a number of applications, see e.g. [4], [7]). In contrast to that, here, we only assume that the controller is able to collect state action transitions by observing the real system behaviour while controlling it.

Learning by interacting with the real system directly has an important advantage: the controller is tailored exactly to the behaviour of the real system at hand instead of a more or less exact model of it. The big challenge in learning with real systems lies in the fact that learning must occur in a reasonable amount of time with a reasonable effort: in a real application, one typically can not wait for hundered thousands of episodes, until a controller is learned.

Another important aspect of this paper, is, that we do not need prior knowledge about the policy to be learned. This has the advantage, that we do not constrain the control law a priori to a certain class of controllers. Additionally, the proposed approach is applicable even in situations, where a priori no idea about a working control law is available.

In principle, value function based methods offer the advantage of a very flexible representation of the control policy to be learned. However, in their original on-line learning form, this advantage comes at the cost of very long training times, which makes them unrealistic for real-life applications. Recently, memory based RL approaches have been proposed, that make approximate model-free value iteration algorithms much more efficient by explicilty memorizing and reusing transition information. One of them is Neural Fitted Q Iteration (NFQ) [11]. NFQ stores all transition tuples (state, action, successor state) seen so far and reuses them in every update step of the Q-function. This compensates an essential problem of 'non-local' function approximators: output values at points in input space, that are currently not updated can be deteriorated or 'forgotten'. Combining this explicit memorization of data points with the otherwise good generalisation abilities of multilayer perceptrons leads to a model-free, Q-value function based RL approach, that is highly efficient with respect to the amount of training data needed. NFQ can be seen as an instantiation of the familiy of Fitted Q Iteration algorithms [1], which themselves are a special kind of Fitted Value Iteration algorithms [3], [8].

In this paper, NFQ is used as the core learning algorithm for a learning controller that learns to steer a real car from scratch. The state space for this problem is 6 dimensional and continuous. We demonstrate, that our approach is able to learn a successful steering in less than 20 minutes of driving the real car. The approach is fully data-driven, which means that no simulation model of the car and no pre-defined control law is used for learning.

2 Steering a Robot Car

2.1 Task Description

The task considered is to follow a given track by controlling the steering angle of the steering wheel. The deviation of the track is measured in terms of the cross-track-error (cte), which should be kept small. A side goal of the controller is to show a 'smooth' steering behavior, that gently controls the car.

Steering is an intermediate step to a fully autonomous car. In its full version, the robot car shall also activate brakes and throttle. In the current state of the project, throttle and brake are activated by a human driver.

2.2 Hardware and Sortware

The robot car is equipped with several sensors that allow to determine its position both with respect to global coordinates (GPS), and to relative coordinates with respect to the current shape of the road (e.g. visual lane detection). This information delivers the pose estimation of the car. From this pose estimation, essential inputs to the controller are determined, like e.g. the deviation of the car from the given track.

On the output side, the car allows to set the steering angle of the wheel by activating a motor coupled to the steering wheel. Currently, the steering controller outputs the wheel angle, which is then transformed into control signals to the motor controller.

2.3 Classical Controller Design

A good controller must not only reduce the error as quickly as possible, but must also consider the dynamic behavior of the vehicle - e.g. in order to avoid break-outs by too harsh steerings. Also, some kind of pro-activeness is expected: a good steering controller should also consider to some extent the future curvature of the road [2]. An anlytical controller design based on these considerations can be found in [6]. This controller was developped based on careful analytical considerations, numerical parameter search methods and handtuning. In contrast, the learning controller proposed in the following, shall learn the control law from interaction with the real car only.

3 Reinforcement Learning and NFQ

3.1 RL Basics

The control problems considered can be described as Markovian Decision Processes (MDPs). An MDP is described by a set S of states, a set A of actions, a stochastic transition function $p(s, a, s')$ describing the (stochastic) system behavior and an immediate reward or cost function $c : S \times A \rightarrow \mathbf{R}$. The goal is to find an optimal policy $\pi^* : S \rightarrow A$, that minimizes the expected cumulated costs

$$J^\pi(s) = E \sum_{t=0}^{\infty} c(s_t, \pi(s_t)), s_0 = s \tag{1}$$

for each state. In particular, we allow S to be continuous and assume A to be finite for our learning system. The transition model p is assumed to be unknown to our learning system (model-free approach). Decisions are taken in regular time steps with a constant cycle time.

3.2 Q-Learning and Neural Networks

The idea of classical Q-learning is to allow model-free Reinforcement Learning by iteratively learning an optimal value function over state-action pairs [15]. Typically, it is applied on-line, which means, that after each observation of a transition of the system, the corresponding value of the Q-function is updated by the following rule:

$$Q_{k+1}(s, a) := (1-\alpha)Q(s, a)+\alpha(c(s, a)+\gamma \min_b Q_k(s', b))$$

where s denotes the state where the transition starts, a is the action that is applied, and s' is the resulting state. α is a learning rate that has to be decreased in the course of learning in order to fulfill the conditions of stochastic approximation and γ is a discounting factor (see e.g. [13]). It can be shown, that under mild assumptions Q-learning converges for finite state and action spaces, where a table-based representation of the Q-function can be used. If every state action pair is updated infinitely often, in the limit, the optimal Q-function is reached. The greedy exploitation of this optimal Q-function finally yields the optimal policy.

To deal with continuous state spaces, the above Q-learning rule can be adapted to be realized in a function approximator. In the following, we will use a neural network of type multi-layer perceptron to store the Q-value function. Although the theoretical convergence guarantees from above do not hold any more in this case, empirically a lot of successful applications of neural Q functions have

```
NFQ_main() {
input: a set of transition samples $D$;
output: neural Q-value function $Q_N$
    k=0
    init_MLP() $\rightarrow Q_0$;
    DO {
        generate_pattern_set
        $P = \{(input^l, target^l), l = 1, \ldots, \#D\}$ where:
            $input^l = s^l, a^l$,
            $target^l = c(s^l, a^l, s'^l) + \gamma\, min_b Q_k(s'^l, b)$
        generate_extra_patterns()
        Rprop_training($P$) $\rightarrow Q_{k+1}$
        k:= k+1
    } WHILE ($k < N$)
```

Figure 2. Main loop of NFQ.

been reported. One particular problem when directly implementing the Q-learning rule in a multi-layer perceptron, is that each update for one state-action pair might induce unforeseeable changes at the Q-values for other state-action pairs - disturbing or even destroying the effort done so far. In practice, this might lead to very long training times, often requiring several hundred thousands of trials until a successful policy is learned. The following will report about an idea to better deal with that problem and as a consequence to reduce training effort drastically.

3.3 Neural Fitted Q Iteration (NFQ)

The crucial idea underlying NFQ is the following: Instead of updating the neural value function on-line after each sample (which leads to the above problems), the update is performed off-line considering the entire set of transition experiences done so far [11]. Experiences therefore are collected in triples of the form (s, a, s') by interacting with the (real) system. Here, s is the original state, a is the action applied and s' is the resulting state. The set of experiences is called the sample set $\mathcal{D}$.

The NFQ algorithm is displayed in figure 2. It consists of two major steps: The generation of the training set P and the training of these patterns within a multi-layer perceptron. The input part of each training pattern consists of the state s^l and action a^l of training experience l. The target value is computed by the sum of the transition costs $c(s^l, a^l, s^{l+1})$ and the expected minimal path costs for the successor state s'^l, computed on the basis of the current estimate of the $Q-$function, Q_k.

The consideration of the entire training information instead of updating on-line after each sample, has an important further consequence: It allows the application of advanced supervised learning methods, that converge faster and more reliably than online gradient descent methods. In our implementation of NFQ, we use the Rprop algorithm for fast supervised learning [12]. The training of the pattern set is repeated for several epochs (=complete sweeps through the pattern set), until the pattern set is learned successfully.

3.4 Applied Extensions to NFQ

To make NFQ work in practice, some further details of the algorithm must be clarfied.

Additional training patterns The immediate cost function that will be used for the steering task (see section 4.2.1) is kind of a 'canonical' choice, that serves us as a standard for a broad range of set-point regulation problems. It reflects the desire to quickly bring a system output close to its target value and keep it there. The control task at hand is non-episodic, meaning that there is no predefined terminal goal state where the value function takes a certain fixed value. However, it can be shown, that states, that can be permanently kept within the target region by any policy, have optimal expected costs of 0. Therefore, for those states we can set the target value to 0 without negatively disturbing the learning process. This is called the 'hint-to-goal' heuristic [11]. It can addionally be enhanced by a dynamic procedure: states, which in course of learning are found that they can be kept in the target region, can also be assigned a target value of 0. Empirically, it can be shown, that in the absence of this heuristic, the output of the neural value function gradually approaches 1 for all states (since the target value for each state action pair is at least as large as the value for its successor state).

Automatic scaling of inputs Since at the beginning of the NFQ loop all input patterns are available, it is possible to scale every input to be in the same range, e.g. from -1 to 1.

Sampling of transitions In principle, the transitions used in NFQ can be sampled arbitrarily. However, when learning with a real system, collecting transitions that lie on actual trajectories will be probably the most practical implementation. A straight-forward way to do this is to greedily exploit the current neural Q-function to determine the current policy. We call this procedure 'greedy-sampling'. It is implemented by sampling transitions following the current greedy policy until an episode is terminated. Then the data collected on the trajectory is added to the transition set and one or more NFQ iterations are performed. The new policy is determined by greedily exploiting the resulting new Q function. In principle, exploration can be added; however in the following experiments, the policy was determined without exploration.

4 An RL Controller for Steering

4.1 Task specification

The control task is to autonomously steer a robot car close to a given track by controlling the angle of the steering wheel. Sensory inputs are the pose of the car (provided by a GPS system), and the speed of the car. The controller output is the desired angle of the steering wheel, in the range of ± 520 degrees. The controller acts with 20 Hz, corresponding to a control interval of $50ms$. The deviation of the car from the given track is measured by the 'cross-track-error (cte)'. As a strict constraint, we request the cte to be always less than 0.5m.

The learning controller shall learn by directly interacting with the real car itself. Observations of sensory values of car behavior recorded during driving are the only source of information, meaning that the approach is purely data-driven. No model, neither analytical nor in terms of a simulator, are considered for learning.

4.2 Setup of the learning task

4.2.1 Immediate costs

The RL task is formulated as the minimisation of expected cumulated costs over time. The cost function $c(s,u) : S \times U \rightarrow \Re$ is chosen to be:

$$c(s,u) = \begin{cases} 0 & , \text{ if } |cte| < 0.05m \text{ (success)} \\ +1 & , \text{ if } |cte| > 0.5m \text{ (failure)} \\ 0.01 & , \ else \end{cases}$$

The cost function reflects the desire for a fast reduction to a low error with some tolerance ($\pm$ 0.05m) in accuracy. A cte-value of more than $\pm$ 0.5m is considered a failure and must be avoided under all circumstances. This is reflected by punishing failure states with a final cost of +1, which is the upper bound of the sigmoidal output neuron of the neural Q function.

4.2.2 Controller inputs

The controller input consists of 5 continuous variables. The aim is to represent kinematic and dynamic information about the state of the car, that is relevant for the steering task. The most important information is the cross-track-error (cte) and its first-order time derivative ($\dot{cte}$). Since the dynamic behavior of the car crucially depends on its speed, the speed of the rear wheels, v_{rear} is the third input. The fourth input is the 'heading-error', which measures the difference between the heading angle of the car and the curvature of the track. The fifth input variable is the so called 'yaw-rate-matching', which measures the difference of the speed of change of the yaw angle with respect to the speed of change of the track curvature. The latter two variables shall provide the controller with the possibility of a 'pro-active' steering, by taking the development of the track into account.

The considerations that lead to the selection of the above variables where based on the discussions of relevant inputs for an analytical controller design [6]. We found this to be a reasonable starting point; testing other combinations of input variables is an area of future research.

4.2.3 Q function and policy

To represent the Q-function, a neural network of type multi-layer perceptron is used. The control action is determined by selecting the action that minimizes the Q-value of the current state, i.e.

$$\pi(s) = \arg\min_u Q(s,u).$$

To do this, for each action respective input vectors have to be propagated through the net and the minimum output value has to be determined. Since the action set typically consists of only few actions, and the neural net computations are fast, this operation is not time-critical.

4.2.4 Actions

The output of the controller delivers the angle of the steering wheel. In a first experiment, we provided only two actions to the controller: steering wheel left or steering wheel right, i.e. $U = \{-360^o, +360^o\}$. When testing this approach on a simulation of the car, the controller actually learned a successful control policy, even when only these two coarse actions are available. The cross-track-error can be successfully held within $[-0.25m, 0.1m]$ and in particular never exceeds $\pm 0.5m$ (see figure 3). However, on a real car, this behavior would not be accetable: the harsh steering leads to lateral accelerations of the car up to 8 m/s^2, which would very likely push the car over. One obvious solution is to extend the action set to have a finer granularity of actions. However, the controller would then still have the principle possibility to go from one extreme to the other. Our solution to the above problem is described in the next section.

4.3 The I-DOE architecture

The idea of 'dynamic output elements' (DOEs) [10] is to enhance the core RL controller by an additional dynamic operator, that takes the RL action as input and computes the final control signal as its output. The crucial point of this arrangement is, that rather than determining the control signal directly, now the *sequence* of RL actions determines the magnitude of the control signal. In prinicple, a DOE can

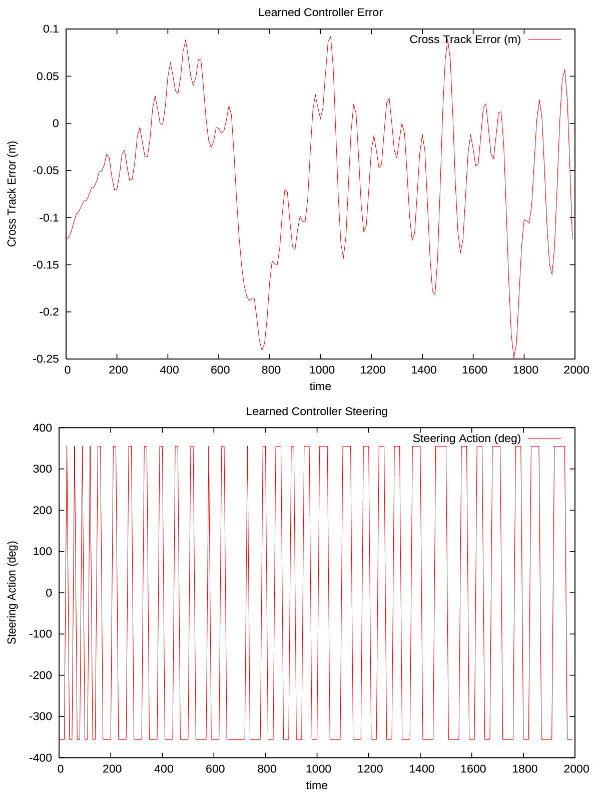

Figure 3. Preliminary experiments on a simulated car: behavior of basic 'bang-bang' RL controller using two actions $\pm 360^o$. Upper figure: The cross-track-error (cte) can successfully be kept between the critical values. Lower figure: The steering angle of the steering wheel switches between the extreme values $\pm 360^o$ in a 'bang-bang' control fashion. On a real car, this would cause unacceptable lateral accelerations.

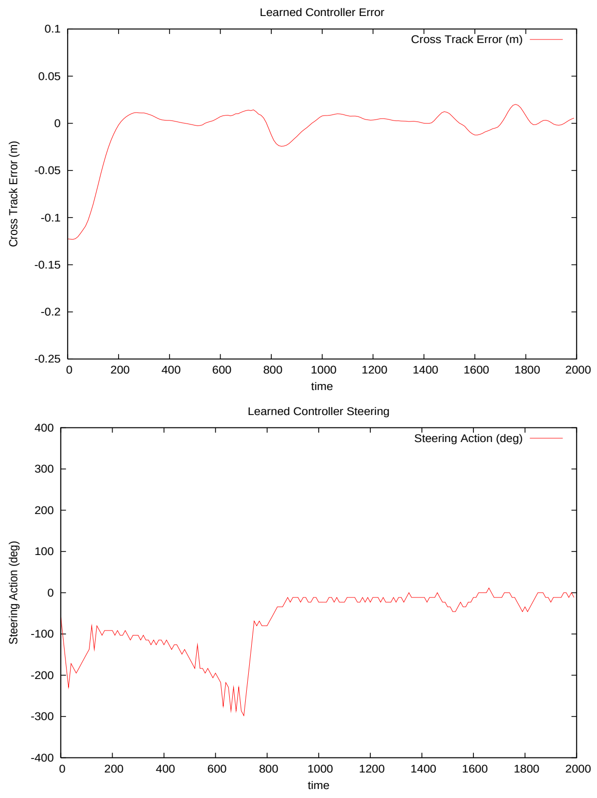

Figure 4. Preliminary experiments on a simulated car: behavior of an RL controller enhanced with an I-DOE. Upper figure: The cross-track-error (cte) is much lower than in the bang-bang RL controller version. Lower figure: The steering angle of the steering is much smoother than in the bang-bang case. The improved behavior is due to the enhanced control capabilities, of which the controller has learned to make use of.

realize arbitrary dynamics - integration, low-pass filtering, etc. A crucial point is to extend the input state of the RL controller by the state of the DOE, in order not to violate the Markov property of the modified system, that is now controlled by RL.

For the task at hand, we use a DOE with a simple integrator property (I-DOE), i.e. the DOE sums the RL actions over time. The state of the I-DOE accordingly evolves by $s_{DOE}(t) = s_{DOE}(t-1) + u'(t)$, where $u'(t)$ is the action selected by the RL core controller. The output of the DOE in this case is equal to the current state of the DOE, i.e. $u(t) = s_{DOE}(t)$. This DOE output $u(t)$ is the control signal (i.e. the steering wheel angle) applied to the car. To describe the complete state of the original plant and DOE, the input of the RL core controller is extended by the DOE state information $s_{DOE}()$.

The action set for the RL DOE-controller is selected to be $U' = \{\pm 60^o, \pm 10^o, 0\}$. For example, to reach a steering wheel angle of -360^o, the RL core controller therefore has to select the action -60^o six times in a row.

Within this framework, the desire for a smooth steering can be expressed easily by rewarding zero actions (i.e. no changes of the steering angle) with low costs. In our case, we realized this by changing the first row in equation 4.2.1 to

$$c(s,u) = 0, \text{if } |cte| < 0.05m \text{ and } u == 0$$

The resulting I-DOE controller shows much smoother behavior than the bang-bang controller previously, both with respect to a low error and a smooth control signal (see figure 4).

4.4 The final controller structure

4.4.1 Principle considerations

Driving the robot car with people sitting inside requires that the car is controlled safely even when learning. Since at the beginning of learning, the RL controller will not be able to stay on the track, precautions have to be taken in order to assure safety. There are several possibilities to deal with that issue:

A. Stop the car automatically, whenever the RL controller failed. Ask the human driver to bring the car back to a non-failure state and start a new RL training episode (a refinement of this without a human driver would be to virtually reduce the cross-track-error to a valid state by virtually moving the target track).

B. Use an automatic 'recovery' control policy, that takes over, whenever the RL controller fails. This has the advantage, that driving can be continued without stopping and with only limited human interaction. Of course, this solution requires a recovery policy that brings the car back closer to the track, but as we argue in the sequel, this recovery policy can be arbitrary simple.

Note that in both cases, recovery policies are only used to re-establish a valid state for the RL controller and *not* to produce examples of good steering. No data collection is done during the regime of the recovery control policy; it therefore can be arbitrary simple without influencing the behavior of the RL learning process. For our experiments, we pursued the second approach, since it allows for a fully-automated and convenient operation of the learning process during driving. For the recovery policy, we used an analytically derived steering controller developped in [6].

4.4.2 Implementation details

Whenever the RL controller reports a failure ($cte > 0.5m$), the recovery controller immediately takes over. It controls the car, until the state is back within the valid region of the RL controller. Then the RL controller takes over again. To avoid the system to permanently switch between the two controllers, a hystereses mechanism was applied: After failure, the RL controller is allowed to take over again, only if the error is below a certain threshold, i.e. $cte < 0.1m$.

During RL control, the controller continously stores the observed transitions triples (s, u, s') to its transition database. When a failure occured, the following things happen: 1. the episode is terminated, 2. the controller triggers a background process to do one iteration of NFQ to produce a new Q-net based on the complete transition database collected so far, 3. the recovery controller takes over (for safety reasons). In the cycles following a failure, the RL controller actively tests, if the cte fulfills the re-entry criterion ($cte < 0.1m$) and if the background learning process has produced a new neural Q-net. If both pre-conditions are fulfilled, the RL controller takes over again and starts its next episode.

This procedure allows the human driver to only activate throttle and brake - steering is done completely autonomously and continuously either by the RL controller or by the recovery controller. The learning process is controlled completely autonomously. In a successful experiment, we expect the RL controller to incrementally increase its active time in course of learning.

5 Learning on the Real Car

5.1 NFQ parameter settings

The control architecture applied was the I-DOE RL controller enhanced with a recovery controller that takes over in case of failure as described in section 4. Control interval is 50ms, control signal is the steering wheel angle of the car. The action set for the RL controller is $U' = \{\pm 60^o, \pm 10^o, 0\}$. The controller is running on a laptop, that communicates with the car steering software via inter process communication.

The neural RL controller starts completely from scratch, with the weights of the neural Q function randomly initialized within the range ± 0.5. As the neural Q function, a multi-layer perceptron was applied consisting of one input layer, two hidden layers with 10 neurons each and an output layer with one output neuron. For all neurons, sigmoidal activation functions with range $(0, 1)$ were used. The inputs consist of 5 continuous variables describing the state of the car, one additional input for the state of the I-DOE, and another input for the candidate action. Transition data was collected in a greedy-sampling manner, i.e. in each episode, the current Q function was greedily exploited to determine the new control (and sampling) policy. After each episode, one iteration of NFQ is performed. The neural network training method applied is Rprop, running for 300 epochs. Rprop was applied with its standard parametrisation [12]. For the training pattern set, 100 extra-patterns according to the 'hint-to-goal' heuristic were generated. The immediate cost function used was the one described in section 4.2.1.

Activating throttle and brake was done by a human driver. Speeds of the car therefore vary with the mood of the driver. For learning, speeds were typically between 4m/s to 7.5m/s. The learned controller was also tested for speeds up to 9m/s.

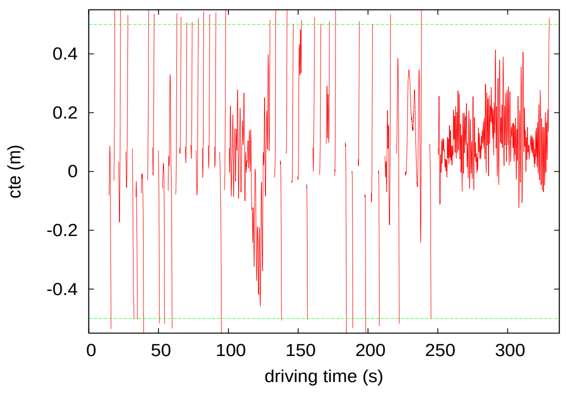

Figure 5. Behavior of the cross-track-error for the RL controller in the first 350 seconds of driving. In the first couple of trials, the controller always quickly fails. After only 90s driving, RL successfully drives for about 30s, after about 250s, RL drives for more than 80s.

5.2 Learning results

As described in section 4 learning is triggered completely autonomously. The only thing the human driver has to do is to operate throttle and brake of the real car. Whenever the pre-condidtions for the RL controller are fulfilled ($cte < 0.1$ and new Q-net available), RL takes over and steers the car. Figure 5 shows the behavior of the of the cross-track-error of RL steering the real car in the first 350 seconds of driving and learning. In the (short) pause between two RL episodes, the recovery controller has control while the RL controller is doing NFQ learning. In the first 90s, whenever the RL controller is in charge, it nearly immediately fails by provoking a cte of more than 0.5m. However, after only 90s of gross driving time, the RL controller shows its first successes by driving about 30s until failure. After 250s gross driving time, RL managed 80s of autonomous steering.

The first controller, that manages a complete round (which was a round trip of about 800m length, corresponding to about 130 seconds driving) was obtained after 56 episodes, after about 11 minutes of driving with the car. That first successful controller still acted pretty ’shakey’ putting the goal of a small error higher than the desire (of the passengers!) for a smooth steering.

The next controller that completed the round was obtained after 60 episodes in total, after about 16 minutes of driving. Figures 6 show the cross-track error which could be successfully held below 0.5m deviation for the entire round. The controller has successfully learned to use its integrating DOE to produce large steering angles if required. At that time, the database contained about $20\,000$ transition tuples.

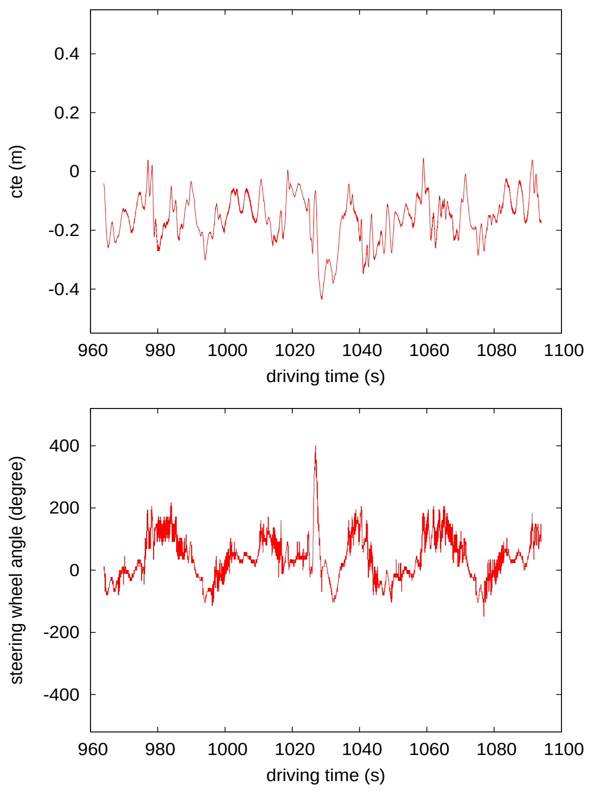

Figure 6. Behavior of RL controller on real car after 60 episodes (about 16 minutes driving). Above: The cross-track-error is successfully kept below 0.5m. Below: Control signal as produced by the RL-DOE controller. The controller learned to generate steering wheel angles of up to 400 degrees, in order to keep the car successfully on track.

With that amount of data, one iteration of NFQ learning took about 30s training time on the laptop used.

Having shown that our approach is able to learn from scratch based on a very reasonable amount of real data, we stopped the experiment after 70 episodes in total. Up to that point, the total driving time was only 25 minutes. We consider that to be a very reasonable time for learning a controller completely from scratch.

5.3 Discussion

Considering that our learning approach is not based neither on an initial guess of a parametrized policy nor on a (simulation) model, NFQ has successfully shown its ability of being very data-efficient. A good part of this performance is due to the good generalisation abilities of multilayer perceptrons, which in combination with the explicit memorisation of transitions seems to be one of the key ingredigents of successful value function learning.

The policy learned is successful, but not optimal yet. Further learning should decrease the cumulated cte further. Also, the steering is acceptable, but definitely could be smoother. Further learning means two things: First, more transitions might be collected to have a better representation of the car behavior - which means that learning is continued in the online-greedy manner as described above. In a comparable simulator experiment, we observed very good controllers to occur after about 120 episodes of online-greedy learning.

Another possibility for more learning would be to increase the number of NFQ iterations offline, i.e. without sampling data on the real car in every iteration. Again, in a corresponding simulator experiment, this procedure also produced better controllers when the number of NFQ iterations is increasing.

6 Conclusion

This paper shows the successful application of neural Q value function learning on a real car directly. Using NFQ as the core training method, learning to steer from scratch was possible in less than 20 minutes of driving a real car. No prior policy and no model are given. We describe an architecture, where the use of a recovery controller allows simulateneous learning during driving. However, the use of such a recovery controller is only optional for convenience; at no point it is used to collect data or to serve as an example policy for steering.

References

[1] D. Ernst and a. L. W. P. Geurts. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556, 2005.

[2] T. Gillespie. *Fundamentals of Vehicle Dynamics*. SAE Publications, Warrendale, PA, 1992.

[3] G. J. Gordon. Stable function approximation in dynamic programming. In A. Prieditis and S. Russell, editors, *Proceedings of the Twelfth International Conference on Machine Learning*, pages 261–268, San Francisco, CA, 1995. Morgan Kaufmann.

[4] R. Hafner and M. Riedmiller. Reinforcement learning on an omnidirectional mobile robot. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003), Las Vegas*, 2003.

[5] R. Hafner and M. Riedmiller. Neural Reinforcement Learning Controllers for a Real Robot Application. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 07)*, Rome, Italy, 2007.

[6] G. M. Hoffmann, C. J. Tomlin, M. Montemerlo, and S. Thrun. Autonomous automobile trajectory tracking for off-road driving: Controller design, experimental validation and racing. In *To appear in the Proceedings of the 26th American Control Conference*, New York, NY, July 2007.

[7] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang. Inverted autonomous helicopter flight via reinforcement learning. In *International Symposium on Experimental Robotics*, 2004.

[8] D. Ormoneit and . S. Sen. Kernel-based reinforcement learning. *Machine Learning*, 49(2-3):161–178, 2002.

[9] J. Peters and S. Schaal. Policy gradient methods for robotics. In *Proceedings of the ieee international conference on intelligent robotics systems (iros 2006)*, 2006.

[10] M. Riedmiller. Generating continuous control signals for reinforcement controllers using dynamic output elements. In *European Symposium on Artificial Neural Networks, ESANN'97*, Bruges, 1997.

[11] M. Riedmiller. Neural Fitted Q Iteration - First experiences with a data efficient neural Reinforcement Learning Method. In *Proc. of the European Conference on Machine Learning, ECML 2005*, Porto, Portugal, October 2005.

[12] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In H. Ruspini, editor, *Proceedings of the IEEE International Conference on Neural Networks (ICNN)*, pages 586 – 591, San Francisco, 1993.

[13] R. S. Sutton and A. G. Barto. *Reinforcement Learning*. MIT Press, Cambridge, MA, 1998.

[14] R. Tedrake, T. W. Zhang, and H. S. Seung. Learning to walk in 20 minutes. In *Proceedings of the Fourteenth Yale Workshop on Adaptive and Learning Systems*, 2005.

[15] C. J. Watkins. *Learning from Delayed Rewards.* Phd thesis, Cambridge University, 1989.