Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method

Martin Riedmiller

Neuroinformatics Group,
University of Onsabrück, 49078 Osnabrück

Abstract. This paper introduces NFQ, an algorithm for efficient and effective training of a Q-value function represented by a multi-layer perceptron. Based on the principle of storing and reusing transition experiences, a model-free, neural network based Reinforcement Learning algorithm is proposed. The method is evaluated on three benchmark problems. It is shown empirically, that reasonably few interactions with the plant are needed to generate control policies of high quality.

Learn Q
MLP representation
Store EXP

1 Introduction

When addressing interesting Reinforcement Learning (RL) problems in real world applications, one sooner or later faces the problem of an appropriate method to represent the value function. Neural networks, in particular multi-layer perceptrons, offer an interesting perspective due to their ability to approximate nonlinear functions. Although a lot of successful applications exist [Tes92, Lin92, Rie00], also a lot of problems have been reported [BM95]. Many of these problems arise, since the representation mechanism in a multi-layer perceptron is not local, but global: A weight change induced by an update in a certain part of the state space might influence the values in arbitrary other regions - and therefore destroy the effort done so far in other regions. This leads to typically very long learning times or even to the final failure of learning at all. On the other hand, a global representation scheme can in principle have a very positive effect: by assigning similar values to related areas, it can exploit generalisation effects and therefore accelerate learning considerably.

function approximator is needed for real-world

"global" representation
- one change influences other values
+
long / failed learning
- OR -
faster learning (generalization)

Therefore the question is: how can we exploit the positive properties of a global approximation realized in a multi-layer perceptron while avoiding the negative ones? One key access to this question is that we need to constrain the malificious influence of a new update of the value function in a multi-layer perceptron. The principle idea that underlies our approach is simple: we have to make sure, that at the same time we make an update at a new datapoint, we also offer previous knowledge explicitly. Here, we implement this idea by storing all previous experiences in terms of state-action transitions in memory. This data is then reused every time the neural Q-function is updated.

Avoid ⊖ ?
- offer prev. knowledge when learning new. "
⇒ store exp. and reuse

J. Gama et al. (Eds.): ECML 2005, LNAI 3720, pp. 317–328, 2005.

The algorithm proposed belongs to the family of fitted value iteration algorithms [Gor95]. They can be seen as a special form of the 'experience replay' technique [Lin92], where value iteration is performed on all transition experiences seen so far. Recently, several algorithms have been introduced in this spirit of batch or off-line Reinforcement Learning, e.g. LSPI [LP03]. Our method is a special realisation of the 'Fitted Q Iteration', recently proposed by Ernst et.al [EPG05]. Whereas Ernst et.al examined tree based regression methods, we propose the use of multilayer-perceptrons with an enhanced weight update method. Our method is therfore called 'Neural Fitted Q Iteration' (NFQ). In particular, we want to stress the following important properties of NFQ:

- the method is model-free. The only information required from the plant are transition triples of the form (state, action, successor state).
- learning of successful policies is possible with relatively few training examples (data efficiency). This enables the learning algorithm to directly learn from real world interactions.
- although requiring much less knowledge about the plant than analytical controllers, the method is able to find control policies, that are able to compare well to analytically designed controllers (see cart-pole regulator benchmark).

Ref. to exp. replay

Properties of NFQ
(1) model-free
(2) data efficient
(3) comparable results

2 Main Idea

2.1 Markovian Decision Processes

The control problems considered in this paper 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 $c : S \times A \to \mathbf{R}$. The goal is to find an optimal policy $\pi^* : S \to A$, that minimizes the expected cumulated costs for each state. In particular, we allow S to be continuous, assume A to be finite for our learning system, and p to be unknown to our learning system (model-free approach). Decisions are taken in regular time steps with a constant cycle time.

(cycle = step)

MDPs
S, A, p(), c, π
finite unknown

2.2 Classical Q-Learning

In classical Q-learning, the update rule is given by

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

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. [SB98]). It can be shown, that under mild assumptions Q-learning converges for finite state and action spaces, as long as every state action pair is updated infinitely often. Then, in the limit, the optimal Q-function is reached.

Typically, the update is performed on-line in a sample-by-sample manner, that is, every time a new transition is made, the value function is updated.

Q-Learning

α

γ

Convergence guarantee

Update:
- online
- sample update

2.3 Q-Learning for Neural Networks

In principle, the above Q-learning rule can be directly implemented in a neural network. Since no direct assignment of Q-values like in a table based representation can be made, instead, an error function is introduced, that aims to measure the difference between the current Q-value and the new value that should be assigned. For example, a squared-error measure like the following can be used: $error = (Q(s,a) - (c(s,a) + \gamma \min_b Q(s',b)))^2$. At this point, common gradient descent techniques (like the 'backpropagation' learning rule) can be applied to adjust the weights of a neural network in order to minimize the error. Like above, this update rule is typically applied after each new sample.

The problem with this on-line update rule is, that typically, several ten thousands of episodes have to be done until an optimal or near optimal policy has been found [Rie00]. One reason for this is, that if weights are adjusted for one certain state action pair, then unpredictable changes also occur at other places in the state-action space. Although in principle this could also have a positive effect (generalisation) in many cases, in our experiences this seems to be the main reason for unreliable and slow learning.

3 Neural Fitted Q Iteration (NFQ)

3.1 Basic Idea

The basic idea underlying NFQ is the following: Instead of updating the neural value function on-line (which leads to the problems described in the previous section), the update is performed off-line considering an entire set of transition experiences. Experiences are collected in triples of the form (s, a, s') by interacting with the (real or simulated) system[1]. Here, s is the original state, a is the chosen action and s' is the resulting state. The set of experiences is called the sample set $\mathcal{D}$.

The consideration of the entire training information instead of on-line samples, 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. Here we use Rprop [RB93], a supervised learning method for batch learning, which is known to be very fast and very insensitive with respect to the choice of its learning parameters. The latter fact has the advantage, that we do not have to care about tuning the parameters for the supervised learning part of the overall (RL) learning problem.

[1] Note that often experiences are collected in four-tuples with the additional entry denoting the immediate costs or reward from the environment. Since we take an engineering view of the learning problem, we think of the immediate costs as something being specified by the designer of the learning system rather than something that occurs naturally in the environment and can only be observed. Therefore, costs come in at a later point and also potentially can be changed without collecting further experiences. However, the basic working of the algorithm is not touched by this.

NN + Q-Learning

Error Function
- diff. between current Q : new Q

Problem ⊖
global weight change
⊕
unpredictable changes
⊕
unreliable / slow learning

NFQ

Off-line update

Allow using advanced supervised learning techniques
ex) Rprop
- fast & robust

3.2 The NFQ -Algorithm

NFQ is an instance of the Fitted Q Iteration family of algorithms [EPG05], where the regression algorithm is realized by a multi-layer perceptron. The algorithm is displayed in figure 1. 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.

NFQ_main() {
input: a set of transition samples D; output: 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:
$input^l = s^l, u^l,$
$target^l = c(s^l, u^l, s'^l) + \gamma \, min_b Q_k(s'^l, b)$
Rprop_training(P) $\rightarrow Q_{k+1}$
k:= k+1
} WHILE ($k < N$)

Fig. 1. Main loop of NFQ

Since at this point, training the Q-function can be done as batch learning of a fixed pattern set, we can use more advanced supervised learning techniques, that converge more quickly and more reliably than ordinary gradient descent techniques. In our implementation, we use the Rprop algorithm for fast supervised learning [RB93]. The training of the pattern set is repeated for several epochs (=complete sweeps through the pattern set), until the pattern set is learned succesfully.

3.3 Sample Setting of Costs

Here, we will give an example setting of the immediate cost structure, which can be used in many typical reinforcement learning settings. We find it useful to use a more or less standardized procedure to setup the learning problem, but we want to stress that NFQ is by no means tailored this type of cost function, but works with arbitrary cost structures.

In the following, we denote the set of goal states $\mathcal{S}^+$, the set of forbidden states are denoted by $\mathcal{S}^-$. $\mathcal{S}^+$ therefore denotes the region, where the system

should finally be controlled to (and in case of a regulator problem, should be kept in), and $\mathcal{S}^-$ denotes regions in state space, that must be avoided by a correct control policy.

Within this setting, the generation of training patterns is modified as follows:

$$target^l = \begin{cases} c(s^l, u^l, s'^l) & , \text{ if } s'^l \in \mathcal{S}^+ \\ C^- & , \text{ if } s'^l \in \mathcal{S}^- \\ c(s^l, u^l, s'^l) + \gamma \, min_b Q_k(s'^l, b) & , \text{ else (standard case)} \end{cases} \quad (1)$$

Setting $c(s^l, u^l, s'^l)$ to a positive constant value c_{trans} means to aim for a minimum-time controller. In technical process control, this is often desirable, and therefore we choose this setting in the following. C^- is set to 1.0, since this is the maximum output value of the multi-layer perceptron that we use. In regulator problems (see section 4), reaching a goal state does not terminate the episode. Therefore, the first line in the above equation must not be applied. Instead, only line 2 and 3 are executed and $c(s^l, u^l, s'^l) = 0$, if $s'^l \in \mathcal{S}^+$ and $c(s^l, u^l, s'^l) = c_{trans}$, otherwise.

Note that due to its purity, this setting is widely applicable and no prior knowledge about the environment (like for example the distance to the goal) is incorporated.

3.4 Variants

Several variants can be applied to the basic algorithm. In particular, for the experiments in section 5.2 and 5.3 we used a version, where we incrementally add transitions to the experience set. This is especially useful in situations, where a reasonable set of experiences can not be collected by controlling the system with purely random actions. Instead, training samples are collected by greedily exploiting the current Q_k function and added to the sample set D.

Another heuristic that we found helpful, is to add 'artificial' training patterns from the goal region, which have a known target value of 0. This technique 'clamps' the neural value function to zero in the goal region, and we therefore call it the *hint-to-goal*-heuristic. Note that no additional prior knowledge is required to generate the patterns, since the goal region is already known in the task specification.

4 Benchmarking

The following gives a short overview of the intention of the benchmarks done in the empirical section.

4.1 Types of Tasks

In control problems, three basic types of task specification might be distinguished (there might be more, but for our purposes, this categorisation is sufficient):

- avoidance control task - keep the system somewhere within the 'valid' region of state space. Pole balancing is typically defined as such a problem, where the task is to avoid that the pole crashes or the cart hits the boundary of the track.
- reaching a goal - the system has to reach a certain area in state space. As soon as it gets there, the task is immediately finished. Mountaincar is typically defined as getting the cart to a certain position up the hill.
- regulator problem - the system has to reach a certain region in state space and has to be actively kept there by the controller. This corresponds to the problems typically tackled with methods of classical control theory.

① Avoid crash

② Reach goal

③ Reach & stay (≈ ① + ②)

The problem types show different levels of difficulty, even when the underlying plant to be controlled is the same. In the following, we consider three benchmark problems, where each belongs to one of the above categories.

4.2 Evaluating Learning Performance

Each learning experiment consists of a number of episodes. An episode is a sequence of control cycles, that starts with an initial state and ends if the current state fulfills some termination condition (e.g. the system reached its goal state or a failure occured) or some maximum number of cycles has been reached.

Episode

(cycles = steps)

Learning time in principle can be measured in many different ways: number of episodes needed, number of cycles needed, number of updates performed, absolute computation time, etc.

Since we are interested in methods that can directly learn on real systems, our preferred measure of learning effort is the number of cycles needed to achieve a certain performance. This number is directly related to the amount of interaction with the plant to be controlled. By multiplying the number of cycles with the length of the control interval, we get the absolute real time that we would have to spend on a real system to achieve a certain performance.

Speed of learning measured by # of "cycles"

We also give the number of episodes that is needed to learn a task. Although this is not as expressive as the number of cycles (since this figure drastically depends on the maximum allowed length of a training episode), it is a commonly used measure and gives at least a rough intuition about the learning effort.

4.3 Evaluating Controller Performance

Controller performance is evaluated with respect to some cost-measure, that evaluates the average performance over a certain amount of control episodes. In principle this cost measure can be chosen arbitrary. Due to its practical relevance, we use the average time to the goal as a performance measure for the controller. In the regulator problem case, we measure the overall time outside the target region. This takes the fact into account, that a controlled system might leave the target region again. Note that the learning controller might have an internal goal formulation that differs from the performance measure (i.e. by using discounting or shaping rewards).

Avg time to goal

-OR-

Time outside S^t

Controller could view reward differently (discount, rew. shaping)

Another important aspect when evaluating controller performance is to specify the 'working region' of the controller, that means the set of starting states, for which the controller should work. We distinguish between the following types of working regions:

- always start from a single starting state
- start from one of a finite set of starting states
- start from an arbitrary random state within a starting region

In the following experiments, we use the third case, which is the most general and (typically) the most challenging one.

5 Empirical Results

All experiments are done using CLS^2 (Closed Loop System Simulator)[2], a software system designed to benchmark (not only) RL controllers on a wide variety of plants.

5.1 The Pole Balancing Task

The task is to balance a pole at the upright position by applying appropriate forces to the system. System equations and parameters are the same as in [LP03]. Three actions are available, left force (-50 N), right force (+50 N) and no force. Uniform noise in $[-10, 10]$ is added to the system. Cycle length is 0.1 s. The state space is continuous and consists of the angle and the angular velocity. An episode was counted as a failure, if the angle of the pole exceeded $\pm\pi/2$ respectively.

Learning System Setup. For comparison, we choose the same cost structure as in [LP03]: Immediate costs of 0 arise, if the angle remains within $[-\pi/2, \pi/2]$ ('$\mathcal{S}^+$'), if the angle gets outside this region, the episode is stopped and costs of $+1$ are given. A discount factor of $\gamma = 0.95$ is used. Transition samples were generated by starting the pole in an upright position and then applying random control signals until failure. The average length of a training episode was about 6 cycles.

NFQ uses a multilayer-perceptron with 3 inputs (2 for the state, 1 for the action), two hidden layers with 5 neurons each and 1 output. For all neurons, sigmoidal activation functions with outputs between 0 and 1 were used.

Results. Lagoudakis and Parr reported very good results both for their LSPI approach and Q-learning with experience replay using a linear function approximator with reasonably selected basis functions [LP03]. The learned controllers were tested on 1000 test episodes with a maximum length of 300 seconds each. LSPI reached an average balancing time of 285 seconds after 1000 training episodes. This means, that most but not all of the training trials generated totally successful policies. For Q-learning with experience replay they report a balancing time of 'about 300' seconds after 750 episodes of training [LP03].

[2] available at clss.sf.net

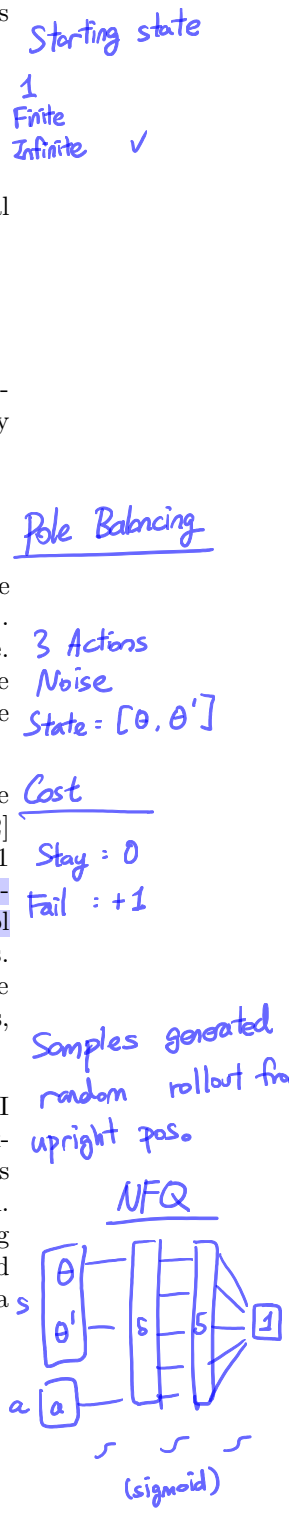


Table 1. Results of NFQ on the pole balancing benchmark. Left column reports the number of random episodes that were used for training. The length of each episode was about 6 cycles. Altogether, 50 repetitions of the experiment were done. For each experiment, a new set of random episodes was produced. Using 200 or more training episodes (about 1200 cycles, corresponding to 2 minutes real time), all experiments generated successful policies, i.e. the controller balanced the pole for all the test cases for the maximum time 300 s.

# random episodes	successful learning trials
50	23/50 (46%)
100	44/50 (88 %)
150	48/50 (96 %)
200	50/50 (100 %)
300	50/50 (100 %)
400	50/50 (100 %)

fully learned

Results of the NFQ method are shown in table 1. The experiments were repeated for 50 times. Each experiment had a different set of training samples and a different initialisation of the neural network weights. With only 50 training episodes (corresponding to about 300 transition samples), NFQ was able to find totally successful policies (policies that balanced the pole for the full 300 seconds for all the test episodes) in 23 out of 50 experiments. Using more training episodes, the result improves. Using only 200 training episodes, a successful policy could be found reliably in all of the 50 experiments. This is a remarkable result with respect to training data efficiency and gives some hint to the benefit of generalisation ability of a multilayer-perceptron.

5.2 The Mountain Car Benchmark

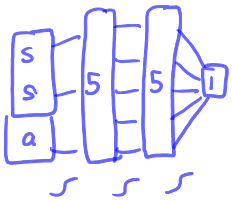
Mountain Car

The mountain car benchmark is about accelerating a car up to the top of the hill, where for many situations the acceleration of the car is too weak to directly go to the top, but instead the car has to move to the other direction to get enough energy [SB98]. The control interval is $\triangle_t = 0.05s$. Actions are restricted within the interval $[-4, 4]$. The road ends at -1m, i.e. the position must fulfill the constraint $position > -1m$. The task is to reach the top, which means that then, the position must be larger or equal to $0.7m$. For testing performance, 1000 starting states are drawn randomly from the interval $(-1, 0.7)$. The initial velocity of the cart is set to 0. Performance is measured by the average number of cycles to the goal.

actions
constraint
goal
perf. test

Learning System Setup. Two actions are provided to the learning controller, -4 and +4. For training, initial starting positions are drawn randomly from $(-1, 0.7)$, the initial velocity of the car was always set to zero. Training trajectories had a maximum length of 50 cycles. An episode was stopped, if the system entered $\mathcal{S}^-$ (failure by constraint violation) or entered $\mathcal{S}^+$ (success). Each training trajectory was generated by a controller, that greedily exploited the current

2 Actions
Training
Max-length
Increment exp.



Q-value function. The Q-value function was represented by a multi-layer perceptron with 3 input neurons (2 state variables and 1 action), 2 layers of 5 hidden neurons each and 1 output neuron, all equiped with sigmoidal activation functions. The weights of the network were randomly initialized within $[-0.5, 0.5]$. After each episode, one iteration of the inner NFQ loop was performed. The hint-to-goal heuristic was used with a factor of 100. For each transition, costs of $c_{trans} = 0.01$ were given.

Results. Results of the NFQ approach on the mountain car benchmark are shown in table 2. The results are averaged over 20 experiments. Experiments differ in the randomly drawn starting states for training and the randomly initialized neural Q-function. Each trial was stopped after 500 training episodes. All 20 experiments produced a successful policy, i.e. a policy that was able to reach the goal state for all of the 1000 randomly drawn starting positions.

To generate a successful policy, only about 71 episodes or 2777 cycles were needed in average over all experiments. This corresponds to less than 2 and a half minutes of training in real time. In the best case, a successful policy could be found in only 356 cycles, but even in worst case, only 10054 cycles were needed, which corresponds to about 8 and a half minutes in real time and therefore still is a very realistic number for an assumed interaction with a real system. Finding a fast policy to the goal can be done in about 296 episodes or about 11000 cycles respectively, corresponding to about 9 minutes in real time. Again, this is a very reasonable number for direct interaction with a real system.

Table 2. Results of NFQ on the mountain car benchmark. The upper part reports on the training effort to reach a succesful policy. A policy is successful, if all test situations are controlled to the goal state. The table shows the figures for the average (best/ worst) number of episodes, the average (best/ worst) number of cycles and the corresponding time for interacting with a real system. The lower part reports on the learning effort to reach an optimized policy. In average over all training trials, the average best costs are 28.7. This value is slightly better than the performance achieved with a fine granulated Q-table (29.0).

Mountain Car				
First successful policy				
	episodes	cycles	interaction time	costs
average	70.95	2777.0	2m19s	41.05
best	10	356		
worst	243	10054		
Best policy found (within 500 episodes)				
	episodes	cycles	interaction time	costs
average	296.6	10922.8	9m06s	28.7
best	101	3660		
worst	444	16081		

The best policies found needed only an average of 28.7 cycles to reach the goal. This figure compares well to a table-based Q-learning approach, which yielded an average of 29.0 cycles to reach the goal. This means that we can expect the NFQ controllers to be pretty close to the optimum. As a side remark (not meant as a true comparison): to get to this result, table based Q-learning required 300,000 episodes (with a maximum length of 300 cycles), and the Q-table had a resolution of $250 \times 250 \times 2$ entries.

5.3 The Cartpole Regulator Benchmark

System dynamics of the cartpole system are described in [SB98]. The control interval is $\triangle_t = 0.02s$. Actions are restricted within the interval $[-10, 10]$. The position is restricted by the constraint $-2.4 \leq pos \leq 2.4$. For testing performance, 1000 starting states are drawn randomly. Results on the cartpole system are typically reported with respect to maximum balancing time. Here, we report results on a more difficult task that comprises balancing, namely cartpole regulation. The task is to move the cart to a certain position *and keep it there* while preventing the pole from falling. The target position of the cart is the middle of the track, with a tolerance of $\pm 0.05m$. As a further complication, we allow initial starting states deviating a lot from the 'all-zero' position: for testing performance, initial pole angles are randomly drawn from $[-0.3, 0.3]$ (in rad), positions are drawn from $[-1., 1.]$ (in m), initial velocities are set to 0.

This more complicated formulation of the cartpole benchmark is closer to realistic control tasks and the resulting controllers can be compared to control policies derived by classical controller design methods.

Learning System Setup. Two actions are available to the learning controller, -10N and +10N. For training, initial starting positions for the cart are drawn randomly from $[-2.3, 2.3]$, initial pole angles are drawn from $[-0.3, 0.3]$ (in rad), cart velocity and angular velocity are initially set to zero. Training episodes had a maximum length of 100 cycles. Each training episode was generated by a controller, that greedily exploited the current Q-value function. The Q-value function was represented by a multi-layer perceptron with 5 inputs, 2 hidden layers with 5 neurons each, and one output neuron, all equiped with sigmoidal activation functions. The weights of the network were randomly initialized within $[-0.5, 0.5]$. After each episode, one loop of the NFQ algorithm was performed. The hint-to-goal heuristic was used with a factor of 100. For each transition, costs of $c_{trans} = 0.01$ were given.

Results. Results for the cart-pole benchmark are shown in table 3. Performance is tested on 1000 testing episodes starting from randomly drawn initial states and having a maximum length of 3000 cycles. In the cartpole regulator benchmark, a controller is successful, if at the end of the episode, the pole is still upright and the cart is at its target position 0 within $\pm 0.05m$ tolerance. Note that all the controllers that solve the regulator problem also solve the balancing problem. Typically, the balancing problem is solved much earlier than the regulator problem (figures not shown here).

Table 3. Results of NFQ on the cart-pole regulator benchmark. Training time was restricted to 500 episodes per trial. For an interpretation of the figures, see explanation at table for the mountain car benchmark.

Cart Pole Regulator				
First successful policy				
	episodes	cycles	interaction time	costs
average	197.3	14439.8	4m49s	319.1
best	75	4016		
worst	309	24132		
Best policy found (within 500 episodes)				
	episodes	cycles	interaction time	costs
average	354.0	28821.1	9m 36s	132.9
best	119	8044		
worst	489	43234		

Again, training is done very efficiently. Although the control problem is challenging, a moderate amount of sample transitions - an average of 14439.8 cycles to find a successful policy and an average of 28821.1 cycles to find the best controller - are sufficient. This corresponds to an average real time of 5 minutes (or 10 minutes respectively for the best controller) that would be needed to do the collection of transition samples on a corresponding real system.

To have a better feeling for the control performance of the learned controller, we analytically designed a linear controller for the cartpole regulator benchmark. We used a pole assignment method where we placed the poles of the closed loop system such that it was stable. Additionally, we tried to find parameters that produced control actions within the interval $[-10, +10]$ according to the above specification. The control law used was $u = -R\,x$, where $R = (30.61, 7.77, 0.45, 1.72)$ and x is the state vector. For the linear controller, the average number of cycles outside the goal region was 402.1 over the 1000 test starting positions. The neural controllers that were learned had an average cost of 132.9, which means that they are about 3 times as fast as the linear controller. This is an even more remarkable result, if one considers, that no prior knowledge about plant behaviour was available to develop the neural policy.

Baseline:
Linear Controller

fast? it could have gone outside after reaching zero...

6 Conclusion

The paper proposes NFQ, a memory based method to train Q-value functions based on multi-layer perceptrons. By storing and reusing all transition experiences, the neural learning process can be made very data efficient and reliable. Additionally, by allowing for batch supervised learning in the core of adaptation, advanced supervised learning techniques can be applied that provide reliable and quick convergence of the supervised learning part of the problem. NFQ allows to exploit the positive effects of generalisation in multi-layer perceptrons while avoiding their negative effects of disturbing previously learned experiences.

The exploitation of generalisation leads to highly data efficient learning. This is shown in the three benchmarks performed. The amount of training experience required for learning successful policies is considerably low. The corresponding time for acquisition of the training data on a hypothetic real plant lies in the range of a few minutes for all three benchmarks performed.

For all three benchmarks, the same neural network structure was successfully used. Of course, this does not mean, that we have found the one neural network that solves all control problems, but it is a positive hint with respect to the robustness of NFQ with respect to the choice of the underlying neural network. Robustness against the parametrisation of a method is of special importance for practical applications, since the search for sensitive parameters can be a resource consuming issue.

Future Work

Robustness of NFQ?

Hyperparameter sensitivity?

References

[BM95] Boyan and Moore. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in Neural Information Processing Systems 7*. Morgan Kaufmann, 1995.

[EPG05] D. Ernst and and L. Wehenkel P. Geurts. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556, 2005.

[Gor95] G. J. Gordon. Stable function approximation in dynamic programming. In A. Prieditis and S. Russell, editors, *Proceedings of the ICML*, San Francisco, CA, 1995.

[Lin92] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

[LP03] M. Lagoudakis and R. Parr. Least-squares policy iteration. *Journal of Machine Learning Research*, 4:1107–1149, 2003.

[RB93] 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.

[Rie00] M. Riedmiller. Concepts and facilities of a neural reinforcement learning control architecture for technical process control. *Journal of Neural Computing and Application*, 8:323–338, 2000.

[SB98] R. S. Sutton and A. G. Barto. *Reinforcement Learning*. MIT Press, Cambridge, MA, 1998.

[Tes92] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, (8):257–277, 1992.