Neural Reinforcement Learning to Swing-up and Balance a Real Pole

Martin Riedmiller
Neuroinformatics Group
University of Osnabrueck
49069 Osnabrueck
`martin.riedmiller@uos.de`

Abstract

This paper proposes a neural network based reinforcement learning controller that is able to learn control policies in a highly data efficient manner. This allows to apply reinforcement learning directly to real plants - neither a transition model nor a simulation model of the plant is needed for training. The only training information provided to the controller are transition experiences collected from interactions with the real plant. By storing these transition experiences explicitly, they can be reconsidered for updating the neural Q-function in every training step. This results in a stable learning process of a neural Q-value function. The algorithm is applied to learn the highly nonlinear and noisy task of swinging-up and balancing a real inverted pendulum. The amount of real time interaction needed to learn a highly effective policy from scratch was less than 14 minutes.

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 [10, 5, 7], also a lot of problems have been reported [1]. 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 failure of learning. 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.

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.

The algorithm proposed belongs to the family of fitted value iteration algorithms [3]. They can be seen as a special form of the 'experience replay' technique [5], 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 [4]. Our method is a special realisation of the 'Fitted Q Iteration', recently proposed by Ernst et.al [2]. 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 handcrafted controllers.

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.

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. [9]). 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.

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: $error = (Q(s, a)-(c(s, a)+\min_b Q(s', b)))^2$. 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 [7]. 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 have a positive effect (generalisation), from our experience this seems to be one of the main reasons 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 [8], a supervised learning method

[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.

```
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:
            $input^l = s^l, u^l$,
            $target^l = c(s^l, u^l, s'^l) + \gamma\, min_b Q_k(s'^l, b)$
        Rprop_training($P$) $\rightarrow Q_{k+1}$
        k:= k+1
    } While ($k < N$)
```

Figure 1. Main loop of NFQ .

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.

3.2. The NFQ -algorithm

NFQ is an instance of the Fitted Q Iteration family of algorithms proposed by Ernst et al. [2], 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.

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 [8]. 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 to this type of cost function, but works with arbitrary cost structures.

In the following, we denote the set of goal states $\mathcal{X}^+$, the set of forbidden states are denoted by $\mathcal{X}^-$. $\mathcal{X}^+$ therefore denotes the region, where the system should finally be controlled to (and in case of a regulator problem, should be kept in), and $\mathcal{X}^-$ 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:

$$target^l = \begin{cases} c(s^l, u^l, s'^l), \text{ if } s'^l \in \mathcal{X}^+ \\ R^-, \text{ if } s'^l \in \mathcal{X}^- \\ c(s^l, u^l, s'^l) + \gamma\, min_b Q_k(s'^l, b), \text{ else} \end{cases} \tag{1}$$

Setting $c(s^l, u^l, s'^l)$ to a positive constant value c_{trans} (e.g $c_{trans} = 0.01$) 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. R^- 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 4.6), 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 $s'^l \in \mathcal{X}^+$ 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 (e.g. shaping information like distance to the goal, etc.) is required.

3.4. Variants

Several variants can be applied to the basic algorithm. In particular, for the experiments in section 4 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*-

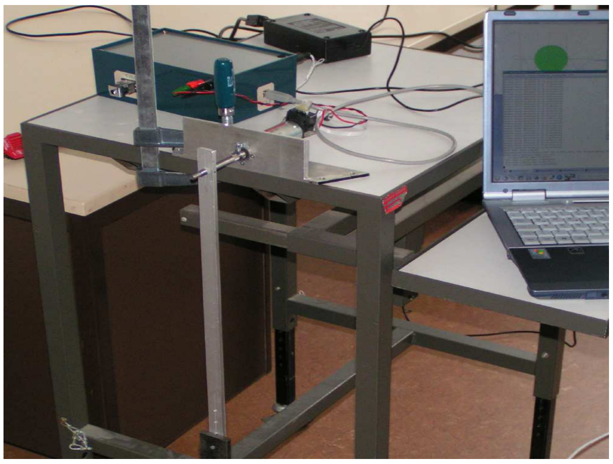
Figure 2. Hardware used. The pole is mounted on the axis of a DC motor. The pole angle is sensed by an encoder attached to the motor. The DC motor is controlled by PWM signals.

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. Swing-Up and Balancing of a Real Pole

All experiments are done using CLS^2 ('clsquare')[2], a software system designed to benchmark (not only) RL controllers on a wide variety of simulated and real plants.

4.1. System Description

The system consists of a pole mounted on the axis of a DC motor (see figure 2). The control signal is given as a PWM signal to the motor. As input information, the controller gets the pole angle, which is sensed by an encoder attached to the motor, and the angular velocity, which is approximated by the difference of two consecutive angular values. The control interval used is $\triangle_t = 0.1s$.

4.2. Swing-up task

The swing-up task is to bring the pole from a hang-down into an upright position, i.e. the target value for the pole angle is $0 \pm 0.3rad$. The pole always starts from a hang-down position, with a pole angle of $+\pi$. Note that in the hang-down position, there is a discontinuity in the angular values: moving the pole a bit to the left or to the right makes the angular value jump from a value of $+\pi$ to a value of $-\pi$ respectively. This additional difficulty could for example be resolved by switching to another representation of the sensor signal. However, according to our philosophy to incorporate as few hand-crafted features as possible into the controller setup, we expect our learning system to be able to cope with this additional difficulty.

The maximum allowed PWM signals are not sufficient to bring the pole up directly. Instead, a succesful control policy has to move the pole back and forth several times, in order to bring more and more kinetic energy into the system such that the pole finally can be brought upright.

The performance measure for a controller is the average time needed to get the pole to the top ($\theta = 0$) from the initial hang-down position ($\theta = \pi$).

As a reference for controller performance, we generated a hand-crafted policy, that always changes the control signal sign, whenever the angular velocity of the pole gets below a certain value. This turns out to be a pretty effective policy. To get the pole up, it required about 3.4 seconds.

The swingup task is a challenging control task for RL, in some sense similar to the popular *mountain car* benchmark, a simulated system often used to compare RL algorithms.

4.3. Learning system setup

Input to the learning controller is the raw and continuous valued state information of angle and angular velocity. While the angle can be directly measured, the velocity is approximated by the difference of consecutive angle measurements. Two actions are available to the learning controller, a positive PWM signal of $+30$ that turns the motor right and a negative PWM signal of -30 that turns the motor left. The Q-value function is represented by a multi-layer perceptron with 3 inputs (two state variables and one control signal), 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 $[-0.5, 0.5]$. The *hint-to-goal* heuristic was used. For the specification of the learning task, the (standard) framework of the immediate cost function from section 3.3 was used with $c_{trans} = 0.01$. No further training information like e.g. shaping was provided to the learning system. For training the neural Q-function, 1000 epochs of batch learning were performed using the Rprop learning algorithm with default parameters [8].

[2]freely available at clss.sf.net

4.4. Acquisition of transition experiences

An important question when directly learning by interaction with the real systems is, how can we collect an 'informative' set of transition experiences (i.e. the sample set $\mathcal{D}$)? In a real system, we typically can not set and experience arbitrary transitions, like we could do in a simulated plant.

In the following experiments, we follow the strategy of 'greedy acquisition', always starting from one initial state: the natural position for the pole is to hang down, therefore this initial state can always be taken by the plant without human interaction. Therefore, all the episodes for training are started from the hang down position. Starting from this inital position, the system is then controlled by greedily exploiting the current Q-value function. Each such episode delievers a set of new potential entries to the sample set $\mathcal{D}$. For reasons of efficiency of learning time, we disregarded multiple entries of the same transition.

It is clear, that at the beginning, when the neural Q-value function is randomly initialized, no successful episodes can be expected. But in the course of learning, more and more succesful episodes are observed. Each trainining episode had a maximum length of 50 cycles (corresponding to 5s). If the pole reached its target, the episode was stopped. This procedure was done for a maximum of 300 episodes. In fact, it turned out that much less episodes were enough to learn successful policies.

4.5. Results

The learning experiments were repeated for 10 times. The following reports the average figures. The first successful episode that brought the pole up to the target region was already observed after only 28 episodes in average (meaning $28 * 50 * 0.1 = 140$ seconds of interaction with the real plant). In average, the first successful policies needed 3.2 seconds to bring up the pole from the hang-down position.

For doing the total of 300 episodes that were run during the training phase, less than 14 minutes (7850 cycles $*0.1s/cycle$) of real time interaction with the plant were required. This stresses the claim, that the NFQ framework makes very efficient use of its training experience. Also, the controller performance was highly satisfiable: the best controller found needed only 2 seconds to swing-up the pole, which is significantly better than the already well-performing hand-crafted policy. The very effective behaviour of the controller is shown in figure 3.

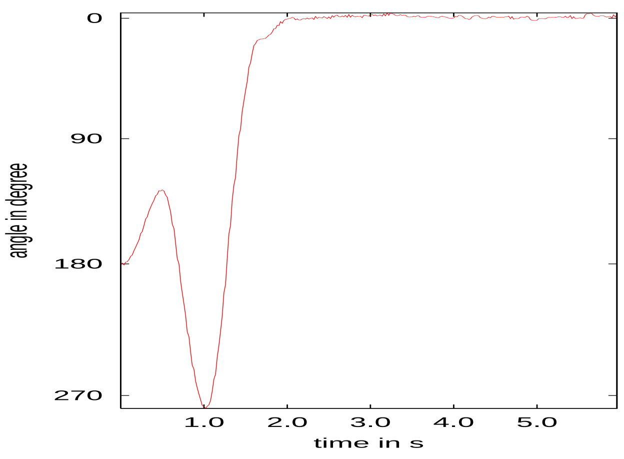

Figure 3. Swingup and balance task. Behaviour of controlled pole angle over time. The values plotted correspond to the inputs to the learning controller. After about 2 seconds, the controller has brought the pole upright and keeps it balanced.

4.6. Swing-up and balance

One big advantage of NFQ is, that once the transition experiences are collected, they can be reused to learn different tasks for the same plant without any further interaction with the plant. This is due to the fact, that within the NFQ approach, learning the controller itself is done completely offline, i.e. the core learning process requires only 'appropriate' transition samples - and does not necessarily rely on a direct interaction with the plant.

To demonstrate this capability, we applied this method to the more challenging swing-up and balance task of the real pole. This task requires to swing-up the pole like before, but additonally to avoid a turn-over and to appropriately balance the pole in the upright position. Balancing is especially difficult, since a PWM signal of 0 is not available to the learning controller - meaning that even with a perfect policy, the pole is permanently pushed away from its stable upright position.

Balancing the pole appropriately with the coarse bang-bang actions of ± 30 requires a shorter control interval than in the swing-up case, where a control interval of $\triangle_t = 0.1s$ was sufficient. Fortunately, data collection on a finer control interval is always possible for free, and was done in the swing-up experiment. Therefore, data for a finer control interval of $\delta_t = 0.025$ is available, and can be used to learn the more complicated task of swing-up and balance the pole.

To learn the new controller, only the learning goal has to be reformulated: we now face a regulator task. In particular, we choose $c(s, a) = 0$, if the angle of s is within $\pm.05$ rad and $c(s, a) = c_{trans} = 0.01$ for all other states. There is a subtle difference to the formulation in 3.3 by the fact that for $s \in \mathcal{X}^+$, no *final* reward is given, since in a regulator problem the episode is *not* terminated when the state reaches $\mathcal{X}^+$ (see [7] for a detailed discussion of this framework). Instead, the controller has to learn the much more difficult task to actively keep the system within the cost-free target region.

Since transition samples are reused from the swing-up task, no further interaction with the real plant is required. Starting with a randomly initialized neural network, the controller learned a perfect swingup and balance behaviour after about 100 iterations of the NFQ algorithm (see figure 3). The final controller needs about 2 seconds to bring up the pole and then kept the pole regulated perfectly in the upright position.

5. Further Results

More experiments using the new NFQ framework have recently been done on simulated plants. NFQ has been evaluated on a simulated pole problem [4], on the classical mountain car [6] and a challenging cart-pole regulator problem. In all benchmarks, NFQ performed amazingly well. The training experiences required to learn highly competitive policies could be collected in less than 10 minutes of corresponding real time interaction for all three benchmarks.The results will be published soon in a forthcoming paper.

6. Conclusion

The paper proposes NFQ , a memory based method to train neural 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 pole swing-up and balance task using a real system. The amount of training experience required for learning successful policies is considerably low, whereas the resulting policy is highly effective.

References

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

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

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

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

[6] A. Barto R. Sutton. *Reinforcement Learning.* MIT Press, Cambridge, Massachusetts, 1998.

[7] 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.

[8] 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.

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

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