10 Steps and Some Tricks To Set Up Neural Reinforcement Controllers

Martin Riedmiller[1]

Machine Learning Lab
Computer Science Department
Albert-Ludwigs Universitaet Freiburg
`riedmiller@informatik.uni-freiburg.de`,
WWW home page: `http://ml.informatik.uni-freiburg.de`

Abstract. Keywords: Neural reinforcement learning, fitted Q, batch reinforcement learning, learning control

1 Overview

The paper discusses the steps necessary to set up a for successfully solving typical (real world) control tasks. The major intention is to provide a code of practice containing crucial steps necessary to transform control task specifications into the specification and parameterization of a reinforcement learning task. Thereby, we do not necessarily claim that the way we propose is the only one (this would require a lot of empirical work, which is beyond the scope of the paper). But, wherever possible we try to provide insights why we do it the one way or the other. In that spirit, this paper is mainly intended to be a subjective report on how we tackle problems by in practice. It is not meant as a general review article and therefore, many related and alternative methods are not mentioned here.

When faced with a real world system, typically a very large number of ways exist to formulate it as a learning problem. This is somewhat different from the situation usually found in reinforcement learning papers, where all the main settings (like state description, actions,) are usually given. In the following we therefore carefully distinguish between the (real world) control problem (which is given) and the learning problem (which we have to design). Of course, ideally, when the learning task is solved, the resulting policy should fulfill the original controller task. The goal of this paper is to show how we can use the degrees of freedom in the modelling of the learning task to successfully solve the original control task. Our procedure of setting up a neural reinforcement learning system worked well for a large range of real, realistic or benchmark-style control applications, e.g. [HR03,Rie05b,HR07,RMD07,GR07,RGHL09,KR09,GLR11].

2 The Reinforcement Learning Framework

2.1 Learning in Markovian Decision Processes

The approach for learning controllers followed here tackles control problems as discrete-time (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 $\pi^* : S \rightarrow A$, that minimizes the expected cumulated costs for each state. In particular, we allow S to be continuous, assume A to be finite, and p to be unknown to our learning system (model-free approach). Decisions are taken in regular time steps with a constant $\triangle_t$.

The underlying learning principle is based on [Wat89], a model-free variant of the idea from . The basic idea is to iteratively learn a value function, Q, that maps state-action pairs to their expected optimal path costs. In Q-learning, the update rule is given by

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

Here, 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 factor. 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 (see e.g. [BT96]). Then, in the limit, the optimal Q-function, Q^*, is reached. The optimal policy π^* can then be derived by greedily evaluating the optimal Q-function:

$$\pi^*(s) \in \arg\min_{a \in A} Q^*(s, a)$$

For a detailed introduction to reinforcement learning the reader is referred to the excellent textbooks [BT96,SB98].

2.2 Q-learning with function approximation

When dealing with large or even continuous state spaces, a tabular representation of the Q-function comes to its limits or is simply infeasible. A standard way to tackle this, is the use of function approximation to represent the Q-function. We focus on neural networks in the following, but other approximation schemes (like e.g. Gaussian processes [DRP09], CMACs [Sut96,TR07], . . .) are being used as well.

One big advantage of using neural networks is their capability to generalize to unseen situations - a fact particularly useful in large or continuous state spaces, where one can not expect to experience all situations during training. However, this positive feature has also a negative impact: when the standard Q-learning rule is applied to a certain state transition, it will also influence the value at other inputs in an unforeseeable manner.

To work against this effect, we developed a neural Q-learning framework, that is based on updating batches of transitions instead of single transition updates as in the original Q-learning rule. This approach has turned out to be an instance of the family of algorithms [EPG05], and was named ' (NFQ)' accordingly [Rie05a].

The basic idea underlying NFQ is simple but decisive: the update of the value function is performed considering the complete set of transition experiences instead of single transitions. Transitions are collected in triples of the form (s, a, s') by interacting with the environment. 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 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 multilayer 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 of each pattern is computed as suggested by the Q-learning rule: it is the sum of the transition costs $c(s^l, a^l)$ plus the expected minimal path costs for the successor state s'^l. The latter is 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 particular, NFQ uses the algorithm for fast supervised learning. Rprop adapts its search step size based on the signs of the partial derivatives and has proven to be very fast and yet robust with respect to the choice of its parameters. For a detailed description of Rprop see [RB93]. The training of the pattern set is executed either for a predefined number of epochs (=complete sweeps through the pattern set), or until the error is below a certain predefined limit. Although simple and straight-forward, training for a fixed number of epochs works well and therefore is our standard choice. For a more detailed discussion about NFQ, please refer to [Rie05a].

3 Characteristics of the control task

In this work, we consider control scenarios of the following type. The controller has to control a technical system or process such that finally a desired target situation is achieved. The current situation of the system is measured by sensors. Thus, the control goal is usually defined by making one or more sensor values equal to externally given reference values within some tolerance bounds. To do so, the controller has to apply an appropriate sequence of control actions. The control system is realized as a , that acts at discrete time steps. At every time-step, the sensor values are measured, and the controller computes a control action, which is then applied to the process.

Different types of control tasks exist within this framework. An important characterization is if the control task has a defined termination or not. In the first case, control terminates immediately, once a certain goal criterion has been achieved. A typical example would be to drive a mobile robot to a certain target location. The task is terminated, once the target location is reached.

The second case is more challenging: the control task does not end, if a target condition is reached once. Instead, the controller has to actively keep the system

```
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 $P = \{(\text{input}^l, \text{target}^l), l = 1, \ldots, \#D\}$ where:
            $\text{input}^l = s^l, a^l$,
            $\text{target}^l = c(s^l, a^l) + \gamma \min_b Q_k(s'^l, b)$
        Rprop_training($P$) $\rightarrow Q_{k+1}$
        k:= k+1
    } WHILE ($k < k_{max}$)
```

Fig. 1. Main loop of NFQ . k counts the number of iterations, k_{max} denotes the maximum number of iterations. *init_MLP*() returns a multilayer perceptron with randomly initialized weights. *Rprop_training*(P) takes pattern set P and returns a mulitlayer perceptron that has been trained on P using Rprop as the supervised training method.

in a set of goal states, that all fulfill a certain success criterion. Typically, this criterion is given by the sensor values being equal to their target reference values within a small tolerance band. This is a very common scenario in the control of technical systems. A typical example would be to achieve and hold a certain temperature in a room. From a control perspective, this latter scenario is much more challenging (since it contains the first one as a special case).

4 Modeling the learning task

This section discusses how to model a given (real-world) control task appropriately within the neural reinforcement learning framework. We discuss alternatives and 'tricks' while always trying to stay as close as possible to the framework proposed by the theory of dynamic programming.

4.1 State information

The underlying reinforcement learning framework crucially depends on the assumption of the Markov property of state transitions: the successor state is a (probabilistic) function of the current state and action. As a consequence, state information provided to the learning system must be 'rich' enough — 'rich' in the sense that the observed state transition does not depend on additional historical information. In a real application, we can not necessarily expect to get the complete state information out of the values provided by the sensors. In classical control theory, the concept of an observer is known to deduce state information out of the sequence of observed sensor information. However, this requires the availability of a system model, which we assume not to have in our learning framework. A standard way to tackle this problem is to provide historical sensor and action information from previous time steps. Since we are learning anyhow, we do not rely on a particular semantic interpretability of the state information. This allows for example to provide more information than necessary or redundant information, to be on the safe side. As a tradeoff, state information should be kept as small and concise as possible to support good generalization, which will generally lead to faster learning and better control performance. In technical dynamical systems, we often use temporal differences of sensor values as an approximation to physically meaningful values like velocity or acceleration.

Like in supervised learning, using meaningful features instead of raw sensor values to enforce generalization is often helpful. However, also like in supervised learning, the design of good features typically requires deep insight into the application (here: knowledge about system behavior which in the strong sense we assume not to have). A current research direction is to autonomously learn meaningful state representations directly out of high-dimensional raw sensor data (like e.g. cameras) [LR10a,LR10b,RLV12,BSWR12].

Summary:

– state information must be designed out of sensor information and must be rich enough to support Markov property
– redundant information is not a problem, but it is preferable to keep the input dimensionality as low as possible
– state representation can be transformed into features to enforce generalization
– state information does not necessarily have a human understandable meaning

4.2 Actions

The original control task often allows the application of (quasi) continuous control values, typically in a certain range between a minimum and a maximum value. While in principle methods exist to learn continuous control actions (e.g. [HR11,Rie97]), we will here focus on providing a discrete set of control to our learning system. This is the most common framework in reinforcement learning and corresponds to the framework as presented in section 2.

This means, for setting up the learning system, one has to explicitly choose a discrete set of actions within the range of potential control signals. One potential choice is a two action set, consisting out of the minimum and maximum control action ('bang-bang'-control). In classical control theory, such a two-value control policy is the basis of time-optimal controllers. Of course, oscillating back and forth between two extreme control signals, e.g. to keep the system close to a desired sensor output, often is not acceptable when it comes to the control of real hardware. Therefore it is often advisable, to add additional actions to the learning set, e.g. a neutral action that does not put additional energy into the system.

The search space of available policies increases exponentially with the number of actions. Therefore, from the perspective of learning time, one should try to keep the number of available actions limited. However, there is of course a trade-off: a smaller number of actions leads to a coarser control behavior and the learning controller might not be able to fulfill the required accuracy in control.

There is a close interplay with the length of the control interval $\triangle_t$: a larger control interval might require a larger action set to achieve the same level of controllability and vice-versa: the smaller the control interval, the coarser the action set may be, since more frequent interaction (and thus correction) is possible. The dynamic output element framework exploits this close relationship between temporal and structural aspects of the action set to enable more flexible control policies [Rie97].

Trivially, a least requirement to the action set is that a policy must exist, that transfers the system to the goal state and - in case of the non-terminal state framework (see below, section 4.4) - keeps it within the goal area.

Summary:

– action set should be kept small to allow fast learning
– tradeoff: more actions can enhance quality or accuracy of control policy
– actions must allow to reach goal states and to keep the system within goal area in the non-terminal goal state setting.

4.3 Choice of control interval $\triangle_t$

The control interval $\triangle_t$ denotes the time between two control interventions of the learning system. In classical control theory, controller design is often assuming that interaction happens in continuous time (like e.g. in classical PID-control). Therefore one aims to make the control interval $\triangle_t$ as small as possible to approximate the assumed continuous time scenario - otherwise, the controller will not work as expected. This is no necessary requirement in the learning framework proposed here. Instead, the controller learns to adapt its behavior to the control interval given - therefore also larger control intervals can be managed. This additional degree of freedom is a big advantage, since for example a slower controller might be realized on less expensive hardware.

As a general tradeoff, learning gets easier, if the control interval is larger, since there are less decision points. On the other hand, the smaller the control interval, the more accurately the system can be controlled. If absolutely no prior knowledge of the system behavior is available, then $\triangle_t$ must be chosen empirically. A potential strategy to determine $\triangle_t$ is to start with a relatively large time step, which helps to learn faster, and then to refine it until the desired accuracy is achieved.

As already discussed in section 4.2 there is a close interplay between the available action set, the control interval $\triangle_t$, and the potential accuracy in control.

Summary:

– the larger the control interval $\triangle_t$, the fewer decisions have to be taken to reach the goal, therefore learning is generally faster
– tradeoff: a smaller control interval $\triangle_t$ potentially allows more accurate control and better control quality

4.4 The terminal goal state and the non-terminal goal state setting

As already discussed in section 3, control tasks can either terminate once a goal criterion is met, or (virtually) continue forever. In the latter case, the control goal is not only to reach a state fulfilling certain success criteria, but to actively keep the system in a set of goal states, that all fulfill these success criteria. A typical success criterion for a state could be, for example, that all sensor values correspond to their target values within some tolerance bound. In the following we discuss, how these two control scenarios can be modeled in our learning framework.

For control tasks, mainly two learning scenarios are most appropriate: the and the non-terminal goal state framework. From the learning perspective, the terminal goal state setting is the simpler one. The task is to transfer the controlled system from an initial state to a terminal goal state by an appropriate sequence of actions. Once the goal state is reached, the episode is stopped, and the target Q-value of the last state action pair is set to the transition costs plus final costs of the terminal state.

This can be implemented by computing the target Q values as follows

$$Q(s,u) = \begin{cases} c(s,u) + \gamma \cdot \mathit{terminal\_costs}(s') & , \quad \text{if } s' \in X^+ \\ c(s,u) + \gamma \cdot \min_b Q(s',b) & , \quad \text{else} \end{cases} \tag{1}$$

where $c(s,u)$ denotes the of a transition (see below). Here, X^+ denotes the set of all terminal goal states, that fulfill the success criteria and by being reached, terminate the control task. For each terminal goal state, $\mathit{terminal\_costs}()$ assigns the corresponding terminal costs.

On the other side, the non terminal goal state framework is particularly tailored to control tasks, where the controller also has to actively keep the system in a set of goal states. Here, X^+ again denotes the set of all goal states, that fulfill the success criteria, but in contrast to the above framework, the control task is not terminated, when one of those states is reached.

This results in the following rule for the update of the Q-values

$$Q(s,u) = \begin{cases} 0 + \gamma \cdot \min_b Q(s',b) & , \quad \text{if } s' \in X^+ \\ c(s,u) + \gamma \cdot \min_b Q(s',b) & , \quad \text{else} \end{cases} \tag{2}$$

where as before $c(s,u)$ denotes the immediate costs of a transition outside the goal region (see below).

This seemingly slight modification has two important consequences: the episode is not stopped, once a state in the goal area is reached, and secondly no 'grounding' of the Q values to a terminal value occurs. This has a nasty effect to the value function, when a multilayer perceptron is used to approximate it. Due to interpolation effects and the lack of grounding, the value function tends to steadily increase. We will discuss this problem in further detail in section 5.3.

Since learning in the terminal goal state framework is usually easier, it sometimes makes sense to model a per-se non-terminal control problem as a terminal state learning problem. The general idea is to consider goal states with a low change-rate as pseudo terminal states. Then, the terminal goal state framework according to equation 1 can be applied. During learning, the task is always stopped, when one of these pseudo terminal states is reached. In the application phase, the policy learned by this procedure is then applied without stopping. The idea behind this is, that whenever the system drifts away from its goal region during application, then the controller immediately brings it back to its goal region.

However, this method is only an approximation to the actually desired behavior. It moreover requires to heuristically define what a 'low change-rate' means within the particular setting. So for non terminal control tasks we recommend to use the non terminal goal state learning framework whenever possible. We only wanted to mention this possibility, because sometimes a control task might be too difficult to learn within the non-terminal goal state framework. Then, an approximation by a terminal goal state problem might constitute a practical way to make it work.

Summary:

– in general, terminal goal states make learning easier
– for control applications, often the nonterminal goal state framework is appropriate, since finding a control policy that also stabilizes the system within the target region is required.

4.5 Choice of X^+

The set X^+ comprises all states, which fulfill the goal criteria as defined by the control tasks. One typical way to define X^+ is to denote ranges for the values of each state variable. For the state variables that we want to control, we typically define a range around their specified target value, i.e. target value $\pm\delta$, where $\delta > 0$ denotes the allowed tolerance. For other state variables, the allowed ranges might be infinitely large, denoting that we do not care what value they have for judging membership to X^+.

Again, there is a tradeoff: the smaller we choose X^+, the more accurate the successfully learned final controller will be. The larger X^+, the easier it will be to learn, but we also have to accept less accurate controllers as a result.

An important requirement from the perspective of the learning framework is that X^+ is large enough, so that a policy exists, that X^+ can be reached from every starting state. Therefore, the choice of X^+ is highly related to the choice of the action set and the choice of the control interval $\triangle_t$.

In the undiscounted ($\gamma = 1$) nonterminal goal state case, an additional requirement applies for X^+. It must be chosen such that a policy exists, that keeps the system permanently within X^+. This policy need not to be known in advance. Again, this requirement implies the interplay between the choice of X^+, the control interval $\triangle_t$ and the available action set.

Summary:

– the larger X^+, the easier it is to learn
– the smaller X^+, the more accurate the learned controller will be

4.6 Choice of X^-

In many control problems, constraints on the state variables exist, that must not be violated by a successful control policy. The definition of the set of undesired states, X^- constitutes a way to model this requirement within the proposed learning framework. In a typical setting, a state is within X^- whenever a constraint of the original control problem is violated. Whenever a state within X^- is encountered, the control episode is stopped. Below, we show the resulting computation for the target of the Q-value as an extension of the equation for the non-terminal goal state framework (equation 2). The application within the terminal goal state framework is straightforward.

$$Q(s,u) = \begin{cases} 0 + \gamma \cdot \min_b Q(s',b) & , \quad \text{if } s' \in X^+ \\ c(s,u) + \gamma \cdot \mathit{terminal\_costs}(s') & , \quad \text{if } s' \in X^- \\ c(s,u) + \gamma \cdot \min_b Q(s',b) & , \quad \text{else} \end{cases} \tag{3}$$

The terminal costs for a state within X^- should ideally be larger than the path costs for any successful policy. When using a multilayer perceptron with a sigmoid output function, we typically use a value close to 1 as terminal costs of a state within X^-.

Summary:

– the learning framework allows the modeling of hard constraints on state variables

4.7 Choice of immediate and final costs

The choice of the immediate cost function $c(s,u)$ determines the course of the control trajectory until the target region is reached. It is not uncommon in reinforcement learning to make $c(s,u)$ a function of the distance to the target region. This has the advantage, that the immediate costs already contain a local hint to the goal, which may help learning considerably. From the perspective of the control task, however, one has to keep in mind, that the final controller optimizes the path costs to the goal. Optimizing the integrated distances to the goal might not always be the ideal realization of what is actually intended (imagine a situation, where a policy first makes a large error but then reaches the goal in a few time steps, instead of a policy that only makes small errors but for a long period of time). The situation gets even more difficult, if one has to design an immediate cost function that trades off between two or more sensor values, that all have to finally achieve a certain target value.

We therefore prefer a cost formulation, that has the advantage of a very simple and thus broadly applicable cost function:

$$c(s,u) = \begin{cases} 0 & , \quad \text{if } s' \in X^+ \\ c & , \quad \text{else} \end{cases} \tag{4}$$

where $c > 0$ is a constant value. A reasonable choice of c is that c multiplied by the estimated number of time steps of the optimal policy should be considerably below the maximum path costs that can be represented by the neural network (which, when using a the standard sigmoid output function, is 1)

The immediate cost function proposed above moreover has the advantage, that the learned optimal policy has a clear interpretation: it is the minimum time controller. As a side note: using this immediate cost function, one can also check the ability of a learning system to learn correct value functions: within this framework, learning can only be successful, if the learned value function is actually meaningful, since no hint towards the goal is provided by the immediate cost function.

The terminal cost function is simple as well: terminal costs are 0, if a terminal goal state is reached, and 1, if a constraint is violated. Of course, these values may depend on the potential range of the output values of the neural network.

Summary:

- costs should serve the purpose of meeting the specifications of the original control task as close as possible
- immediate costs may reflect local hints to the goal to help learning but this might not necessarily reflect the intention of the original control task

4.8 Discounting

In the above equations, γ with $0 \leq \gamma \leq 1$ denotes a discounting parameter. Using $\gamma < 1$ may make learning the value function easier, since the horizon of the future costs considered is reduced (consider e.g. the extreme case where $\gamma = 0$. Then only the immediate costs are relevant). On the other hand, choosing a discount rate also has an influence on the resulting optimal policies: if $\gamma < 1$, immediate costs that occur later in the sequence are weighted less. One has to make sure, that this is in accordance with the initial control task formulation. We therefore usually prefer a formulation with no discounting, i.e. $\gamma = 1$ and therefore have to make sure that for successful learning, additional assumptions are fulfilled (e.g. the existence of proper policies, which basically means that X^+ can be reached from every state with non-zero probability. For a detailed discussion see e.g. [Ber95]).

Summary:

- discounting requires less assumptions and therefore can make learning simpler and/ or more robust
- it must be checked, whether introducing a discounting rate for the sake of better learning still matches the intention of the original control task.

4.9 Choice of X^0

In a typical setup, at the start of each episode, an initial starting state is randomly drawn from the starting state set X^0. Ideally X^0 is chosen such that it covers the whole range of initial conditions that occur in the original control task.

In tasks, that in average require a large number of steps to reach the goal states, the probability of hitting the goal region by chance can be pretty low. Here, a method that we call the *'growing-competence'-heuristic* [Rie96] might help: First, start with initial states close to the goal area and then incrementally increase the set of starting states until it finally covers the complete original starting state area.

Summary:

- the set of initial starting states for learning should cover the intended working space of the original control problem
- if applicable, then starting with simple states first and then increasing the range might help to improve the learning process dramatically

4.10 Choice of the maximal episode length N

Control episodes might take infinitely long — this is inherently the case in the non-terminal goal state framework and can also occur in the terminal goal state setting, if the policy neither finds to the goal region nor crashes. Therefore, while learning, one typically stops the episode after some predefined number of time steps. This is called the maximal episode length N in the following. Theoretically, within the fitted Q learning framework, N is not a critical choice. It just denotes the number of transitions sampled in a row (this is different from learning methods that rely on complete trajectories). Actually, N might be as low as 2. Then, per episode only one transition sample is collected. From a practical perspective, however, it typically makes more sense to consider longer episodes — in particular, when the policy used to sample the transitions drives the system closer towards the goal region and therefore allows to collect more and more 'interesting' transitions.

A rough heuristic that we use is to make N double or three times as large as the expected average time a successful controller will need to reach the target region. If N is chosen too large, then a lot of useless information might be collected - consider for example very long episodes that just contain cycles of ever the same states.

Summary:

– theoretically, the choice of N is not critical
– practically, N can considerably influence learning behavior, since it influences the distribution of the collected transitions.

5 Tricks

5.1 Scaling the input values

Like in normal supervised learning, scaling the input values is an important preprocessing step. Various methods and according explanations are discussed in [LBOM98]. As a standard method, in all our learning experiments, we normalize the input values to have mean of 0 and a standard deviation of 1.

Summary:

– like in supervised learning, it is important that all input values have a similar level
– a simple scaling to mean 0 and standard deviation 1 works well in all our learning experiments so far

5.2 The X^{++}-Trick

If no explicit terminal state exists (which is the case in the nonterminal goal state framework), the output of the neural network tends to constantly increase from iteration to iteration. This is due to the choice of the transition costs, which are 0 (within the target region) or positive (outside the target region). Therefore, the target value of each state action pair is larger or at least equally large than the evaluation of its successor state. Amplified by the generalization property of the multilayer perceptron, this leads to the tendency to ever increase the output values of all state action pairs.

A simple but effective remedy against this effect is to actually fix the values of some state action pairs to 0. We call the set of such states, for which we assume this to be true, X^{++}. This heuristic is in accordance with a correct working of the value iteration scheme, as long as 0 is the expected optimal path costs for the respective state action pairs in X^{++}. Of course, usually state action pairs for which this is true, cannot be assumed to be known a priori. Therefore, in order to apply this trick, one has to rely on heuristics. One reasonable choice of X^{++} are states, that lie in the center of X^{+}, the region of zero transition costs. The reasoning behind this is the following: if one starts at a state x at the center of X^{+}, then a good control policy has a very high chance of keeping the system within X^{+} forever — which justifies to assign zero path costs to that starting state.

If X^{++} is chosen too large, then for some states within X^{++} the assumption of optimal path costs of 0 may be violated. As a consequence, the resulting policy most likely will not fulfill the expected property of reliably keeping the system within X^+. On the other hand, if X^{++} is too small, the chance, that a state actually falls into X^{++} is very low and therefore the heuristic becomes ineffective. A remedy against this, is to actually force the learning system to face states in X^{++}. One possibility to do that, is to enforce starting episodes close to X^{++}, another possibility is to introduce artificial state transitions, which is discussed in the context of the in the next section 5.3.

Summary:

– the X^{++} heuristic is a method to prevent the value function to steadily increase
– if applied carefully, it is in perfect accordance with the value iteration scheme

5.3 Artificial training transitions

In a certain sense, the learning process can be interpreted as spreading its knowledge of the optimal value function from the goal states to the rest of the state space. Therefore, it is crucially required to actually have a reasonable number of state action pairs that lead to a goal state within the overall transition sample set. An obvious recipe would be to try to enforce the occurrence of such goal states, e.g. by starting episodes close to the goal area. However, this is not possible for all systems because they do not allow to set arbitrary initial states.

An unconventional method to cope with the situation is to add artificial state transitions to the sample set. Then, the pattern set used for training consists of actually collected transitions, as well as additionally added artificial transitions. This method was first introduced as part of the *hint-to-goal* heuristic in our first NFQ paper [Rie05a] and has meanwhile also been successfully applied by other researchers using other function approximation schemes (e.g. Gaussian processes, [DRP09]). The idea of the *hint-to-goal* heuristic is to introduce artificial state transitions, that start in X^{++} and end in X^{++}. Those states have — by definition of X^{++} — terminal costs of 0. As a consequence, the value function is 'clamped' to zero at these input patterns. Supported by the generalization ability of the function approximation, also the neighboring states will tend to have a low and thus attractive value.

If for the artificially introduced state action pairs the optimal path costs are actually zero, the hint-to-goal heuristic will not negatively interfere with the correct working of the value iteration process. An obvious choice therefore is a state action pair, where the state is well embedded in X^+ such that the assumption of optimal path costs of 0 is most likely fulfilled. We usually generate such an artificial state action pair by combining such a state with every action in the action set.

The number of artificial patterns should be chosen such that a 'reasonable balance' between experience of success and regular state transitions exists (as a rule of thumb, something between 1:100 and 1:10). This is of course a number, that has to be determined empirically. We are currently working on methods that automatically find such a balance, but this is ongoing work and beyond the scope of this paper.

Summary:

- the *hint-to-goal* heuristic might help to establish a goal region in the value function, if real experiences of success are difficult to achieve during regular learning

5.4 Growing Batch

The fitted Q iteration framework originally works with a fixed set of transitions. No particular assumption is made, how these transitions are collected. In the extreme case, these experiences are randomly sampled all over the working space of the controller in advance. In a practical setting, however, this is not always feasible. One reason is, that arbitrary sampling all over the working space is not realizable, since initial states can not be set arbitrarily. Another reason is, that to sample transitions equally over the working space might just be infeasible due to the huge amount of data required to cover the complete space.

Therefore it is desirable to concentrate on regions of the state space that are relevant for the final controller. One method to realize this is the *growing batch* method. The idea is, that one starts with an empty transition set. After the first episode, the value function is updated and the new episode is controlled by exploiting the new value function. Different variants exist, e.g. the value function can only be updated after n episodes, or the number k_{max} of NFQ iterations between two episodes can be varied. In most of our experiments so far we successfully used this growing batch procedure with the choice of $n = k_{max} = 1$.

Summary:

- the *growing batch* method aims at collecting more and more relevant transitions when the performance of the policy increases.

5.5 Training the neural Q-function

To represent the value function, we use a neural multilayer perceptron. Although it is often believed that setting up such kind of networks is a black art and its parameters are hard to find, we found that this is not particularly critical in the proposed neural fitted Q framework. One crucial point however is to use a powerful training algorithm to train the weights. The *Rprop* learning algorithm combines the advantage of fast learning and uncritical parameter choice [RB93]. We always use Rprop with its standard parameters. Also we found, that the number of epochs (sweeps through the training set) is not particularly critical. We therefore always train for 300 epochs and get good results. One can also think of ways to monitor the training error and find some stopping criterion to make this more flexible (e.g. to adapt to different network sizes, to different pattern set sizes, etc.), but for the applications we had so far, we found this a minor issue for learning success.

Surprisingly, the same robustness is observed for the choice of the neural network size and structure. In our experience, a multilayer perceptron with 2 hidden layers and 20 neurons per layer works well over a wide range of applications. We use the tanh activation function for the hidden neurons and the standard sigmoid function at the output neuron. The latter restricts the output range of estimated path costs between 0 and 1 and the choice of the immediate costs and terminal costs have to be done accordingly. This means, in a typical setting, terminal goal costs are 0, terminal failure costs are 1 and immediate costs are usually set to a small value, e.g. $c = 0.01$. The latter is done with the consideration, that the expected maximum episode length times the transition costs should be well below 1 to distinguish successful trajectories from failures.

As a general impression, the success of learning depends much more on the proper setting of other parameters of the learning framework. The neural network and its training procedure work very robustly over a wide range of choices.

Summary:

– choice of multilayer perceptron is rather uncritical
– important to have a powerful learning algorithm to adjust the weights
– advantage, if the supervised learning algorithm is not particularly dependent on the choice of its parameters.

5.6 Exploration

In reinforcement learning, — the deviation from a greedy exploitation of the current value function — is important to explore the state space. Various suggestions for good exploration strategies have been proposed, e.g. considering a safe control behavior in the learning phase [HSSU08]. From our experience with NFQ, a simple ϵ-greedy exploration scheme is often sufficient. This means that in every time step, with a certain probability (e.g. 0.1), the action is chosen randomly instead of greedily exploiting the value function.

In many application cases, we also observe good results even with no explicit exploration at all. This is due to the fact, that the learning process itself — the randomly initialized neural value function, the growing experience, the randomly distributed starting states — already bears a fair amount of randomness. To learn without explicit exploration is also of practical interest. When always acting greedily, the performance achieved in a training episode is already the performance, that the final greedy controller will show. This reduces the effort of additional testing and therefore is particularly interesting for real world tasks.

Summary:

– a simple ϵ greedy exploration scheme is often sufficient

– if the starting states are well distributed in the working space, then in conjunction with the *growing batch* method, even an always greedy exploitation of the current value function works in many cases

5.7 Delays

In practical systems delays play a crucial role. Delays may occur both on the sensor side - i.e. a sensor value is available only n time steps later, or on the actor side - a control action has an effect only some time steps later. Simply neglecting these effects typically leads to bad control behavior or even failures. Various methods exist, e.g. to use prediction or filter methods to synchronize the information available to the controller with the current world situation. One simple but effective method is to augment state information with historical information about previous actions applied to the system [WNLL07,RHLL08].

Summary:

– in practice, actuator and sensor delays may often be not neglectable
– a simple remedy is to add historical information about previous action values to the current state information

6 Experiments

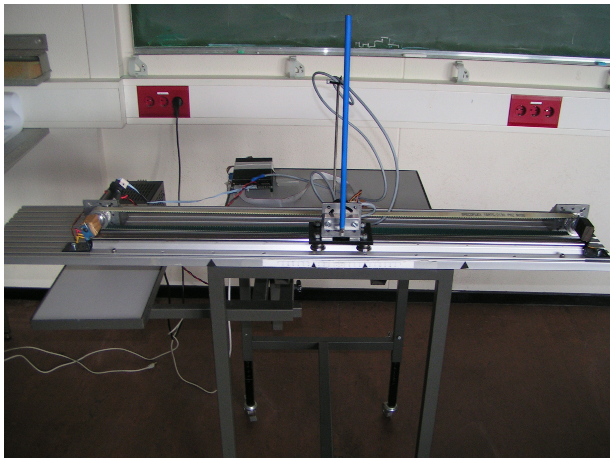

Fig. 2. The real cart pole system.

6.1 The control task

The control task tackled in the following is to control a real cart pole system. While is a well known benchmark [SB98], this real world task is characterized by additional challenging features:

- the initial states can not be set to arbitrary values. We moreover assume, that no human intervention is allowed, in particular, the system can initially only be started with the pole hanging down
- the control task is to balance the pole upright with high accuracy and with the cart at a given target position (here: in the middle of the track). The controller therefore not only needs to learn to swing-up the pole from the downright position, but also to do it in such a sophisticated manner, that finally it can be balanced at the requested position.
- one cannot directly control the force applied to the cart, but only the voltage given to the DC motor driving the car. This introduces additional dynamical effects into the system.
- due to communication effects, the sensor information is delayed
- there is considerable noise in both actuation and sensor values.
- there is a discontinuity (jump) in sensor values from $-\pi$ to $+\pi$ when the pole is in the downright position.
- there is a hard constraint: the position of the cart may not be less than -0.25m and more than 0.25m, since the track is bounded.
- the final controller should be able to work from arbitrary initial start states, not only from one position.

The range of control inputs is (quasi) continuous from -12 volt to 12 volt. Sensor information provided by the system is the position of the cart and the pole; no velocity information can be measured. The target values for the sensor values should be reached as fast as possible. The minimum control interval allowed by the hardware is $\triangle_t = 0.01s$.

Since on the real system we can only perform a limited number of experiments, we also report some results on a reliable simulation of the real system (section 6.5). The input and output interfaces are exactly the same for both simulated and real plant. The simulation model was derived by parameterizing a physical model of the plant using real data. The accurate match between real and simulated system behavior allowed us to do keep all modelling decisions and learning parameter settings the same for both the simulated and the real system. Therefore in the following, we only describe the real system setup. An implementation of the simulated plant is available within our open-source learning framework CLSquare available at

`http://ml.informatik.uni-freiburg.de/research/clsquare.`

6.2 Modeling as a learning task

State description State information provided to the learning system consists of sensor values of position (p_t), angle (α_t), the normalized temporal difference of these measurements $\frac{p_t - p_{t-1}}{\triangle_t}$ and $\frac{\alpha_t - \alpha_{t-1}}{\triangle_t}$. The angle is zero, when the pole is upright. The angular value has a discontinuity (a jump from $-\pi$ to $+\pi$) when the pole is hanging down. Nothing particularly is done to resolve this discontinuity. Instead, we expect the learning algorithm to be able to deal with that. To cope with the sensor delay, additionally the value of the previous control action a_{t-1} is added to the state information.

Actions The action set available for the learning system consists of the 'standard' choice of minimal and maximal control signal plus the 'neutral' action 0V. Thus $A = \{-12V, 0V, +12V\}$.

Control interval $\triangle_t$ As defined by the hardware, the minimum length of the control interval is 0.01s. After some initial experiments, we found that a control interval of $\triangle_t = 0.05s$ is still sufficient for an acceptable control quality while at the same time allowing a fast and successful learning process.

Non-terminal goal state framework For the cart-pole task, control must be continued once pole angle and cart position reached their target values to actively keep the system within the goal states. This means, that the correct formulation of the learning problem is the non-terminal goal state setting. As a consequence, every episode is only interrupted, if the system state entered the failure set X^- or if the maximum number of steps per episode, N is reached.

Choice of X^+ A state is in X^+, if the following two conditions are fulfilled: the cart position is at most 0.1m away from the target position (here: middle of the track) and the pole angle deviates from 0 rad by maximally ± 0.15 rad. The rest of the state entries is not considered for judging membership to X^+.

Choice of X^- A state is in X^-, if the cart position is less than -0.25m or more than 0.25m. This corresponds to the physical boundaries of the track. The rest of the state entries is not considered for judging membership to X^-.

Immediate and final cost functions As immediate costs we use the standard minimum-time formulation with constant transition costs of 0.01. Thus,

$$c(s,u) = \begin{cases} 0 & , \quad \text{if } |p_t| \leq 0.1m \text{ and } |\alpha_t| \leq 0.15\text{rad} \\ 0.01 & , \quad \text{else} \end{cases} \tag{5}$$

When a state from X^- is observed, the episode is stopped and final costs of +1 are assigned.

Episode length Empirically, a good episode length was found to be $N = 200$.

6.3 Applied Tricks

Scaling We applied our standard input scaling procedure as described in 5.1.

Choice of X^{++} and artificial transitions X^{++} contains only one state, namely if all state variables are exactly 0. This corresponds to the center of X^{+}. Of course, this state will most likely not occur by chance in the learning process. Therefore, this definition makes only sense in conjunction with adding artificial transitions in the spirit of the *hint-to-goal* heuristic. Here, we used 3 different artificial patterns, namely state (0,0,0,0,0) combined with all 3 actions. These transitions were repeated 100 times in the training pattern set, in order to establish some balance between the (huge) number of normal transitions and those special transitions. The target values for those artificial patterns is 0.

Growing batch Learning was implemented as a 'growing batch' process. This means, that after every episode, one NFQ iteration (new calculation of Q-target values, supervised learning of the neural Q function) was performed. Then the next episode was controlled by ϵ-greedy exploitation of this new Q function.

Training the neural Q function The neural Q function is represented by a multilayer perceptron with 6 input neurons, two hidden layers with 20 neurons each and one output neuron. Hidden neurons use the tanh activation function, the output neuron uses the standard sigmoid function. Rprop with standard parameters was used for weight updates. In every NFQ iteration step, the network weights of the learning network were randomly initialized between -0.5 and 0.5 before training. The network was trained for 300 epochs per NFQ iteration.

Exploration No explicit exploration scheme was used for the experiments done here, i.e. the current Q function was always exploited greedily to determine the action. This has the advantage, that the application performance can already be determined during training.

6.4 Measuring quality

The quality of a learning control approach has two important aspects: the quality of the learning process and the quality of the resulting controller. The quality of the learning process is measured by the learning effort needed, usually measured in the number of transitions (or the number of episodes) needed, the quality of the achieved solution with respect to the used cost function, and the reliability of the results over a number of learning trials.

The quality of the resulting controller is measured with respect to the specification of the original control task. Relevant criteria are for example accuracy, robustness, working area, and performance measures like e.g. the time outside the tolerated error zone. For a detailed discussion of different criteria also see [HR11].

Here, we first report results achieved in a realistic simulation. This allows us to conduct a series of 10 experiments with different seeds of the random generator in reasonable time. For learning on the real system, we used exactly the same setup and parameters. The only difference we made was, that the controller was allowed to learn for a maximum of 500 episodes on the simulated cart-pole and - due to time restrictions - for a maximum of 300 episodes on the real cart-pole system.

6.5 Results on the simulated cart pole

For the simulated system, all 10 runs delivered a successful controller. 'Successful' means, that for a test set of 100 random initial starting situations, the controller was able to swing up the pole and then steadily keep the system within the desired tolerance. A test run lasted 20s. In average over 10 runs, the best controller was found after an average training of 392 episodes with a standard deviation of 80. The average time needed by the best controllers was 3.23s with a standard deviation of 0.16s.

setup	successful trials	best controller at episode	time outside of X^+
Default	10/10	392 (80.7)	3.23s (0.16s)

Table 1. Results on the simulated cart-pole for the standard setup, averaged over 10 trials. Shown are the average number of episode to train the best controller and its control performance, measured in time outside the target region. The number in brackets shows the respective standard deviation.

6.6 Results on the real cart pole

The evaluation on the real cart-pole system was slightly different, due to the effort it takes to do experiments with the real device. However, the overall picture of the learning behavior on the simulated and real system was consistent.

We performed three learning trials with different initializations of the random generator. Each learning trial lasted 300 episodes. Besides the reduced number of maximum episodes, the setup of the learning system was exactly the same as for the simulated system. In all 3 trials performed, successful controllers were learned within less than 300 episodes of training. In particular, the controllers are very robust with respect to varying initial states or to disturbance from outside.

A video documenting learning and final controller performance is available at

`http://www.youtube.com/watch?v=Lt-KLtkDlh8`

7 Conclusion

This paper discusses many of the basic modeling and methodological tricks to set up a reinforcement learning task. These insights should help to successfully handle a wide range of interesting control problems. The proposed method builds on neural fitted Q iteration (NFQ), a method that considers the complete batch of collected transitions to update the Q function. While the paper is written from the perspective of using a neural network, it should also give useful insights when using other kinds of function approximation schemes.

Current and future work is aiming to further improve the method in several directions. One big direction is to improve NFQ with respect to resulting controller quality (e.g. accuracy, continuous actions, interpretation of control policies, increasing complexity of control tasks, etc). Some steps in this direction have already been made and are discussed in [HR11]. Another area of ongoing and future research is to further improve NFQ with respect to robustness and autonomy of the learning process. A third area is to improve efficiency with respect to the data required for learning. Beyond that, distributed reinforcement learning algorithms that cooperatively control a complex system in a multi-agent setting is a vital research area. Distributed learning systems that are based on the neural learning framework presented here have been successfully applied in typical multi-agent scenarios like distributed job-shop scheduling [RG11,GR08]).

8 Acknowledgment

The author wants to especially thank Roland Hafner from Cognit GmbH for the important initial ignition for writing this article.

References

[Ber95] D. P. Bertsekas. *Dynamic Programming and Optimal Control, Vol. I and II.* Athena Scientific, Belmont, Massachusetts, 1995.

[BSWR12] Manuel Blum, Jost Tobias Springenberg, Jan Wlfing, and Martin Riedmiller. A Learned Feature Descriptor for Object Recognition in RGB-D Data. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, St. Paul, Minnesota, USA, 2012.

[BT96] D. P. Bertsekas and J. N. Tsitsiklis. *Neuro Dynamic Programming.* Athena Scientific, Belmont, Massachusetts, 1996.

[DRP09] Marc P. Deisenroth, Carl E. Rasmussen, and Jan Peters. Gaussian Process Dynamic Programming. *Neurocomputing*, 72(7–9):1508–1524, March 2009.

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

[GLR11] T. Gabel, C. Lutz, and M. Riedmiller. Improved Neural Fitted Q Iteration Applied to a Novel Computer Gaming and Learning Benchmark. In *In Proceedings of the IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 2011)*, Paris France, April 2011. IEEE Press.

[GR07] T. Gabel and M. Riedmiller. On Experiences in a Complex and Competitive Gaming Domain: Reinforcement Learning Meets RoboCup. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, Honolulu, USA, 2007.

[GR08] T. Gabel and M. Riedmiller. Adaptive Reactive Job-Shop Scheduling with Reinforcement Learning Agents. *International Journal of Information Technology and Intelligent Computing*, 24(4), 2008.

[HR03] Roland Hafner and Martin 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.

[HR07] Roland Hafner and Martin 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.

[HR11] Roland Hafner and Martin Riedmiller. Reinforcement learning in feedback control. *Machine Learning*, 27(1):55–74, 2011. 10.1007/s10994-011-5235-x.

[HSSU08] Alexander Hans, Daniel Schneegaß, Anton Maximilian Schäfer, and Steffen Udluft. Safe exploration for reinforcement learning. In *ESANN*, pages 143–148, 2008.

[KR09] T. Kietzmann and M. Riedmiller. The Neuro Slot Car Racer: Reinforcement Learning in a Real World Setting. In *Proceedings of the Int. Conference on Machine Learning Applications (ICMLA09)*, Miami, Florida, Dec 2009. Springer.

[LBOM98] Y. LeCun, L. Bottou, G. Orr, and K.-R. Müller. Efficient backprop. In G. Orr and K.-R. Müller, editors, *Neural Networks: Tricks of the trade*, pages 5 –50. Springer, 1998.

[LR10a] Sascha Lange and Martin Riedmiller. Deep auto-encoder neural networks in reinforcement learning. In *International Joint Conference on Neural Networks (IJCNN 2010)*, Barcelona, Spain, 2010.

[LR10b] Sascha Lange and Martin Riedmiller. Deep learning of visual control policies. In *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN 2010)*, Brugge, Belgium, 2010.

[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.

[RG11] Martin Riedmiller and Thomas Gabel. Distributed Policy Search Reinforcement Learning for Job-Shop Scheduling Tasks. *TPRS International Journal of Production Research*, 50(1), 2012. Available online from May, 2011.

[RGHL09] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange. Reinforcement Learning for Robot Soccer. *Autonomous Robots*, 27(1):55–74, 2009.

[RHLL08] Martin Riedmiller, Roland Hafner, Sascha Lange, and Martin Lauer. Learning to Dribble on a Real Robot by Success and Failure. In *Proceedings of the 2008 International Conference on Robotics and Automation (ICRA 2008)*, Pasadena CA, 2008. Springer. video presentation.

[Rie96] M. Riedmiller. Learning to control dynamic systems. In Robert Trappl, editor, *Proceedings of the 13th. European Meeting on Cybernetics and Systems Research - 1996 (EMCSR '96)*, Vienna, 1996.

[Rie97] M. Riedmiller. Generating continuous control signals for reinforcement controllers using dynamic output elements. In *European Symposium on Artificial Neural Networks, ESANN'97*, Bruges, 1997.

[Rie05a] M. Riedmiller. Neural Fitted Q Iteration - First experiences with a data efficient neural Reinforcement Learning Method. In *Lecture Notes in Computer Science: Proc. of the European Conference on Machine Learning, ECML 2005*, pages 317–328, Porto, Portugal, October 2005.

[Rie05b] M. Riedmiller. Neural reinforcement learning to swing-up and balance a real pole. In *Proc. of the Int. Conference on Systems, Man and Cybernetics, 2005*, Big Island, USA, October 2005.

[RLV12] Martin Riedmiller, Sascha Lange, and Arne Voigtlnder. Autonomous reinforcement learning on raw visual input data in a real world application. In *Proceedings of the International Joint Conference on Neural Networks*, Brisbane, Australia, 2012.

[RMD07] Martin Riedmiller, Mike Montemerlo, and Hendrik Dahlkamp. Learning to Drive in 20 Minutes. In *Proceedings of the FBIT 2007 conference.*, Jeju, Korea, 2007. Springer. Best Paper Award.

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

[Sut96] R. S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*, pages 1038–1044, Cambridge, MA, 1996. MIT Press.

[TR07] S. Timmer and M. Riedmiller. Fitted Q Iteration with CMACs. In *Proceedings of the IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning (ADPRL 07)*, Honolulu, USA, 2007.

[Wat89] C. J. Watkins. *Learning from Delayed Rewards.* Phd thesis, Cambridge University, 1989.

[WNLL07] Thomas J. Walsh, Ali Nouri, Lihong Li, and Michael L. Littman. Planning and learning in environments with delayed feedback. In *Proceedings of the 18th European conference on Machine Learning*, ECML '07, pages 442–453, Berlin, Heidelberg, 2007. Springer-Verlag.