Approximate Model-Assisted Neural Fitted Q-Iteration

Thomas Lampe and Martin Riedmiller

Abstract— **In this work, we propose an extension to the Neural Fitted Q-Iteration algorithm that utilizes a learned model to generate virtual trajectories which are used for updating the Q-function. Compared to standard NFQ, this combination has the potential to greatly reduce the amount of system interaction required to learn a good policy. At the same time, the approach still maintains the generalization ability of Q-learning. We provide a general formulation for approximate model-assisted fitted Q-learning, and examine the advantages of its neural implementation regarding interaction time and robustness. Its capabilities are illustrated with first results on a benchmark cart-pole regulation task, on which our method turns out to provide more general policies using much less interaction time.**

I. Introduction

Q-LEARNING is a powerful and general approach to reinforcement learning that allows a system to acquire arbitrary skills purely from success and failure, with little to no prior knowledge of the true system dynamics. Among different approaches to Q-learning, batch methods such as Fitted Q-Iteration [1] and its neural version Neural Fitted Q-Iteration (NFQ) [2] have proven useful for stable and robust learning. However, in practice such methods often suffer from limitations regarding the currently attainable speed of learning, which prevents their application to many interesting but complex problems. One such limitation lies in the nature of iteration itself: When a terminal state is observed, it will not influence the Q-values of other states immediately, since its value needs to be iterated through the function first. Thus, when learning in batch mode, there is a minimum number of episodes of interacting with the system that will be needed to acquire a suitable policy. In general, this number corresponds to the minimum number of steps that a trajectory from starting state to goal requires. Since interaction with the system can be costly, be it in terms of energy or wear of hardware, it would be desirable to be able to further speed up learning.

An obvious solution to the problem lies in simply performing multiple updates of the Q-function after each episode in order to propagate new experiences from terminal to starting state immediately. However, given the inevitable sparseness of data in the beginning of the learning process, such multi-updates on the same data set are likely to cause overfitting when using an approximator to represent the Q-function – which is a condition for practically any data-efficient algorithm and moderately complex task.

In principle, it is possible to apply a regularization mechanism with which to limit the approximator's complexity and thus reduce overfitting. For instance, Farahmand et al. [3] have used regularized least squares for regularized fitted Q-learning. When using neural networks, weight decay can likewise be used. However, doing so carries in turn some risk of reducing the complexity of the function in places where it actually *should* be complex. Specifically, sharp peaks in the value function might disappear even when they are desirable, as can commonly occur in tasks where a goal state is located between failure states. This would not be a problem if enough data was available, but with the goal being to allow learning with little interaction, there may be only single samples in the peak regions which consequently get regularized away.

To resolve such issues, the alternative approach that will be used in the following involves learning a model from the observed data. By performing virtual trajectories on this model, additional data can then be generated. Assuming that the system's policy is updated after every rollout through the model, this achieves a greater data spread than simply training repeatedly on the same set of transitions. In doing so, we avoid the need for Q-function regularization, and can instead simply fit the approximator to the larger data set.

This approach still uses techniques of model-free Q-learning and augments them with a model, rather than using it directly for learning a value function in a model-based manner. As such, data observed during real interaction can easily be incorporated with that generated by the model, which is desirable given that the real data is more reliable than the virtual. It is also possible in principle to switch from an almost purely model-based learning process in the beginning to purely model-free learning once enough data is available. The method thus constitutes an intermediate case between model-based and model-free learning. For the lack of an established term, we will refer to it as *model-assisted learning* in the following, or, since the model is to be learned as well, more specifically as *approximate* model-assisted learning.

Such bootstrapping from a learned model to boost the speed of Q-learning has previously been used in online learning by the Dyna family of algorithms [4], [5]. However, if the Q-function needs to be updated online while interacting with the system, the number of virtual rollouts is strictly limited by the system's cycle rate. As such, the Dyna algorithm uses the model only to predict the current trajectory several steps into the future, rather than generating an amount of data suitable for boosting early learning. In addition, the amount of time required for updating the respective approximator

The authors are with the Machine Learning Lab, Department of Computer Science, Faculty of Engineering, Albert-Ludwigs-University, Freiburg, Germany (email: {tlampe, riedmiller}@informatik.uni-freiburg.de).

This work was partly supported by the BrainLinks-BrainTools Cluster of Excellence funded by the German Research Foundation (DFG, grant number EXC 1086), and partly by DFG grant "Mobile Manipulation in natürlichen Umgebungen - TidyUpRobot" (grant number RI 923/7-1).

generally precludes the execution of a large number of updates, and thus prevents solving the goal state propagation issue described above.

In contrast, when applying model-assisted learning in a fitted batch learning setting, the Q-function approximator is updated only between interaction episodes. Consequently, it is possible to perform an arbitrary number of rollouts and boost the data set used for learning to any desired size. In addition, as many updates of the Q-function can be performed as are needed to propagate the values of terminal states through the entire function.

The basic principle behind using model-assisted learning to boost batch learning is not tied to any specific implementation of the Q-function and the model. In the following, we focus on the neural variant for both, thus gaining an approximate model-assisted version of NFQ, which we dub *Approximate Model-Assisted Neural Fitted Q-Iteration (AMA-NFQ)*. By choosing neural networks as approximators, we gain a compact representation of arbitrarily complex functions, while the resulting policy remains fast to evaluate.

To sum up, we introduce a general fitted approach to approximate model-assisted learning, and provide a neural implementation for it. In the following we will first describe the method in detail in Section II. Afterward, we will examine its performance in a classical cart-pole benchmark in Section III with regards to the amount of interaction needed and the quality of the attained policy. Using this setting, the approach will be compared to its classical model-free variant.

II. Methods

A. Markov Decision Processes

Before describing the specifics of the proposed method, we briefly introduce the notation used in the remainder of this work. In reinforcement learning, a problem is usually formulated as a Markov Decision Problem (MDP), which can be formalized as a tuple $\{S, A, P, c\}$ [6]. The MDP consists of a set of states S that the system can be in, a set of actions A that are available to the agent, a state transition probability function $P(s'|s,a)$, and a transition cost function $c : s \times a \times s' \to \mathbb{R}$. Our aim is to learn a Q-function $Q : s \times a \to \mathbb{R}$ that estimates the expected cost-to-go when choosing action a in state s, without assuming prior knowledge about P. This Q-function can then be used as a control policy $\pi : s \to a$ by simply choosing whichever action minimizes the expected costs in a given state, i.e. by $\pi(s) = \operatorname{argmin}_a Q(s,a)$.

B. Approximate Model-Assisted Fitted Q-Learning

In principle, the method we propose is a straightforward extension of standard fitted Q-learning [1]. When training a controller in a batch fashion, one would normally perform a trajectory on the real system $\mathfrak{S}$ using the policy derived form the current Q-function. The trajectory is added to a memory B, and the Q-function is fitted to the new transition pool using some update function $\mathcal{C}$, as illustrated in Algorithm 1.

To achieve approximate model-assisted learning, we additionally learn a transition model $\mathfrak{M}$ that approximates P. $\mathfrak{M}$ is updated using an update function $\mathcal{U}$ and the collected real transitions B^R, as in Algorithm 2. At the end of each interaction episode, virtual trajectories B^V are generated from the model and used to update the Q-function.

Algorithm 1 Standard Fitted Q-Iteration

```
Require: Q update function $\mathcal{C}$
$Q \leftarrow$ initialization
$B \leftarrow \emptyset$
loop
    $T \leftarrow \mathfrak{S}(Q)$                      ▷ perform trajectory
    $B \leftarrow B \cup T$                        ▷ extend memory
    $Q \leftarrow \mathcal{C}(Q, B)$                   ▷ update Q-function
end loop
```

Algorithm 2 Approximate Model-Assisted Fitted Q-Iteration

```
Require: Q update function $\mathcal{C}$, model update function $\mathcal{U}$
$Q \leftarrow$ initialization
$B^R \leftarrow \emptyset$
loop
    $T \leftarrow \mathfrak{S}(Q)$                      ▷ perform real trajectory
    $B^R \leftarrow B^R \cup T$                    ▷ extend real memory
    $\mathfrak{M} \leftarrow \mathcal{U}(\mathfrak{M}, B^R)$               ▷ update model
    $B^V \leftarrow B^R$                          ▷ copy memory
    for $r$ rollouts do
        $T \leftarrow \mathfrak{M}(Q)$                  ▷ perform virtual trajectory
        $B^V \leftarrow B^V \cup T$                ▷ extend virtual memory
        $Q \leftarrow \mathcal{C}(Q, B^V)$             ▷ update Q-function
    end for
end loop
```

It is worth noting that by using a temporary copy B^V of the transition store B^R, we discard any virtual transitions after they have been used for the update, and replace them with new ones from the updated model after the next episode. This approach bears resemblance to the Dyna-2 algorithm [5], where separate Q-functions were learned on the real and simulated data, dubbed the long-term and short-term memories, respectively. Such a procedure was necessitated by the fact that past experiences are only stored implicitly in the Q-function when learning online. In contrast, batch learning allows a straightforward treatment of obsolete experiences by simply discarding them rather than splitting the Q-function into parts.

C. Neural Fitted Q-Iteration

The general approach of model-assisted fitted Q-iteration as described above does not assume the use of any specific type for the Q-update function $\mathcal{C}$. Here, we use a multilayer perceptron to represent the Q-function. It is updated using the same update function as in the Neural Fitted Q-Iteration (NFQ) algorithm [2], which is depicted in Algorithm 3. After each episode, a new, randomly initialized network is prepared. Training targets are generated by adding the observed transition costs $c(s_i, a_i, s'_i)$ to the costs-to-go estimated by the previous Q-function. The network is then fitted to the resulting training patterns using the Rprop variant of backpropagation [7], which uses a gradient-independent momentum term for fast convergence.

Algorithm 3 NFQ Update Function $\mathcal{C}^{NFQ}$

```
Require: current Q-function $Q_k$, transition samples $B$
$P^{in} \leftarrow \emptyset, P^{out} \leftarrow \emptyset$
for $i < |B|$ do                                   ▷ generate pattern set
    $P_i^{in} \leftarrow (s_i, a_i)$
    $P_i^{out} \leftarrow c(s_i, a_i, s_i') + \gamma \min_b Q_k(s_i, b)$
end for
$Q_{k+1} \leftarrow init\_weights$
for $e$ epochs do
    $Q_{k+1} \leftarrow Rprop(Q_{k+1}, P^{in}, P^{out})$   ▷ Rprop training
end for
return $Q_{k+1}$
```

In keeping with the neural architecture, we also use a feed-forward neural network to approximate the system model. Instead of directly learning the transition function P directly, we train the net to predict the difference Δs between states. The net is updated using Rprop gradient descent in the same manner as the Q-function.

III. Experiments

To illustrate the advantages of the AMA-NFQ approach over its classical variant, we examined a cart-pole system on which we attempted to solve a balancing and regulation task.

A. Cart-Pole Regulator

We have adopted the system dynamics introduced by Barto et al. [8], [6], which have since become a standard benchmark for reinforcement learning algorithms. The four-dimensional system state is described by the linear position p and velocity Δp of the cart, as well as the angular position θ and velocity $\Delta\theta$ of the pole attached to it. Cart positions are restricted to the interval $[-2.4; 2.4]$. The available actions are defined by the linear force to be applied to the cart, with $10N$ and $-10N$ being available, and can be given at time intervals of $0.02s$.

The task to be optimized here was a combined balancing and regulation problem. In addition to preventing the pole from falling down, the agent was required to keep the cart within a narrow target zone. Specifically, the goal was to bring the system into a target area S^+, which required keeping the pole in the interval $\theta^+ = [-0.05; 0.05]$ (in rad), and the cart in $p^+ = [-0.05; 0.05]$ (in m), as depicted in Fig. 1. If the system left the working area S^{work} defined by $\theta^{work} = [-0.7; 0.7]$ and $p^{work} = [-2.4; 2.4]$, the agent was considered to have failed (i.e. entered S^-), and the episode was aborted.

Initial states could vary from the upright position, and for training were sampled randomly from the interval $[-0.5; 0.5]$ (in m) for the cart position p^{train}, and $[-0.5; 0.5]$ (in rad) for the pole angle θ^{train}. The initial velocities of both cart and pole were always 0. A training episode lasted for 200 cycles or until the system left the working area. For testing, 100 initial states were sampled uniformly from $p^{test} \in [-1; 1]$ and $\theta^{test} \in [-0.5; 0.5]$. Episodes could last for 500 cycles during testing.

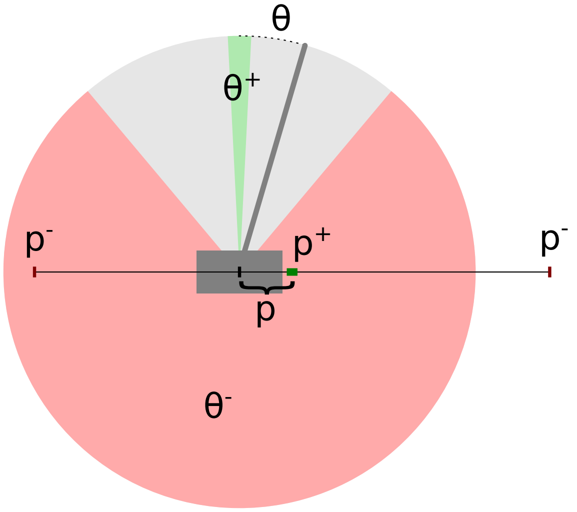

Fig. 1. Illustration of the cart-pole system, with goal (green) and failure (red) areas indicated to scale.

No exploration strategy was used; both during training and testing, policies were performed greedily. Costs were assigned purely based on time, with $c = 0.01$ for each transition not in the goal region and $c = 1$ for failure; no reward shaping was utilized, thus yielding the cost function:

$$c(p', \theta') = \begin{cases} 0 & \text{if } p' \in p^+ \text{ and } \theta' \in \theta^+ \\ 1 & \text{if } p' \in p^- \text{ or } \theta' \in \theta^- \\ 0.01 & \text{otherwise} \end{cases}$$

B. Results

As a first comparison, we examine how much of an increase in performance the use of a model brings over the standard version of NFQ.

When using standard NFQ, a single update of the Q-function was performed after each episode. For AMA-NFQ, 10 rollouts were performed, each of them followed by an update. In both cases, the Q-function was represented by a feed-forward multi-layer neural network with two fully connected hidden layers of 20 units each, a structure that has proven useful for Q-learning in many previous applications [9]. The model was represented by a multi-layer percepton as well, using a structure of a single hidden layer with 20 units. Again, the structure was chosen arbitrarily based on past experiences in order to avoid having to tailor parameters to the task, i.e. providing prior knowledge. For all nets, each unit used a sigmoidal activation function, and weights between them were randomly initialized in the interval $[-0.5; 0.5]$. During each application of the update function $\mathcal{C}^{NFQ}$, $e = 300$ epochs of Rprop gradient descent were performed. While likely not enough for perfect fitting of the net to the data, this amount has often turned out to be

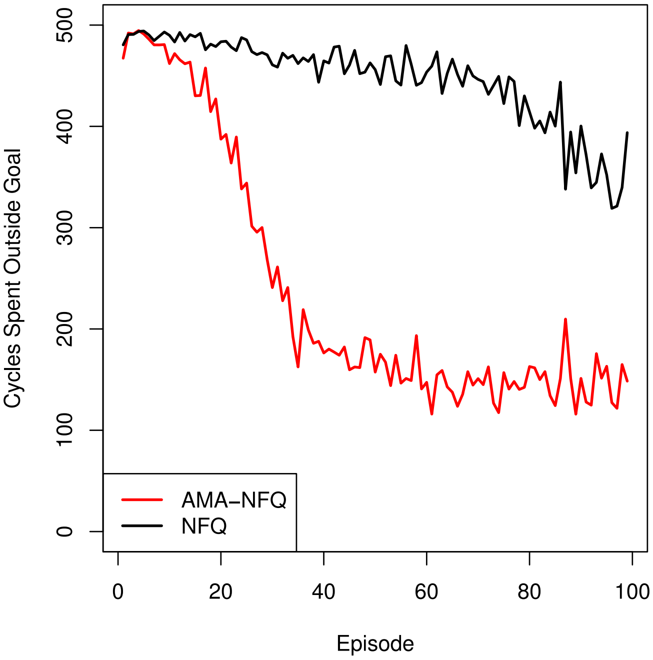

Fig. 2. Average number of steps spent outside target during testing. The episode length of 500 steps provides the upper limit.

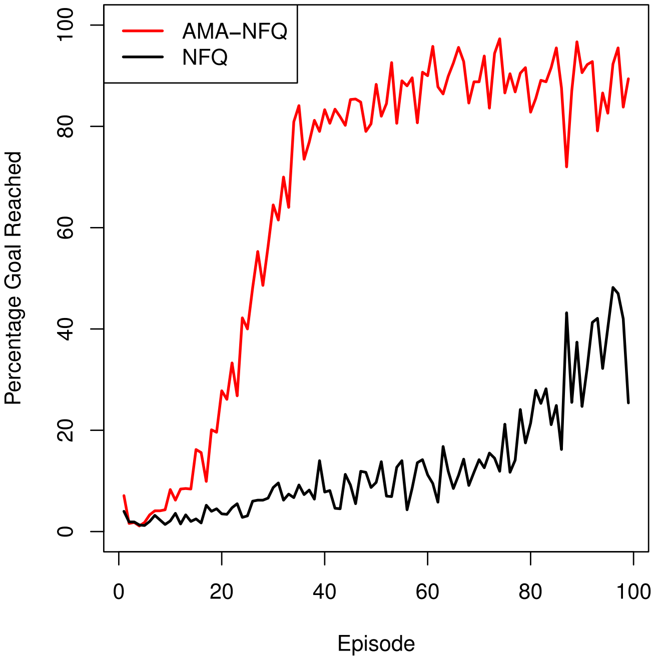

Fig. 3. Average percentage of successful test trials during testing.

sufficient to learn the relative costs between different states in the past.

The system was trained for 100 episodes. We compared the accumulated transition costs, i.e. the number of steps that the system spent outside the target range, with results averaged over 10 runs of the experiment and each policy evaluated from 100 test positions. Since the reward function was specified in such a way as to optimize for minimum time, this measure corresponds to the actual policy performance. As Fig. 2 shows, the number of steps decreased considerably faster for AMA-NFQ than for NFQ.

In addition to performance, we also examined the robustness of the learned policies. To this end, the average number of test runs that reached a terminal state were compared. As evident from Fig. 3, AMA-NFQ achieved a much higher success rate, managing to reach the goal from approximately 80% of all initial positions in the end, compared to only 30% for standard NFQ.

Minimum-time optimization can also be used in situations where any robust policy is sufficient, rather than a globally optimal one, as well as when the main aim is to generate a sample set for fixed-batch learning. In a task like the simulated cart-pole, where the performance of each policy can be evaluated easily, there is then no need to wait for convergence of the learning process. Instead, once a desirable first policy occurs, it can be chosen as final. Therefore, it is worthwhile to not only look at the average performance, but also at the number of episodes needed until the first occurrence of an admissible policy. The first policy to reach a terminal state from all test positions occurred much earlier for AMA-NFQ than for NFQ, as Fig. 4 shows. AMA-NFQ managed to produce such a policy during every single run, requiring between 19 and 50 episodes to do so. In contrast, standard NFQ could did not yield any policy capable of reaching the goal from all initial positions within the 100-episode time frame. Its best policy across all 10 runs managed a success rate of 98%, though fully robust policies were acquired eventually in all but one run, where the top performance did not exceed 67%. The average reliability of NFQ reached that of AMA-NFQ eventually, as shown in Fig. 5, with results again averaged across 10 runs.

The cause for the earlier model acquisition becomes apparent when examining the model's prediction error. Fig. 6 shows its development over time, computed across the working range S^{work}, as well as across velocities $\Delta\theta$ and Δp within the interval $[-0.5; 0.5]$. The system rapidly learned a faithful representation by around episode 10 as Fig. 7 illustrates, and it was not long after that fully robust policies appear in Fig. 4. Particularly in the region around the target state was the model error reduced quickly, whereas it remained higher at the more sparsely sampled edges of the system's working range, as illustrated in Fig. 8

While the results for AMA-NFQ were consistently better than for classical NFQ, there was a large difference in the amount of time required by each method to reach these results. Whereas a 100-episode run of NFQ could be completed within under two hours on a commodity system (without parallelization or GPU usage), AMA-NFQ required more than a day for the same number of episodes.

IV. Discussion

In summary, we have proposed AMA-NFQ, a new approximate model-assisted extension to Neural Fitted Q-Iteration. A general formulation for other types of approximate model-assisted fitted Q-learning, using arbitrary approximators on

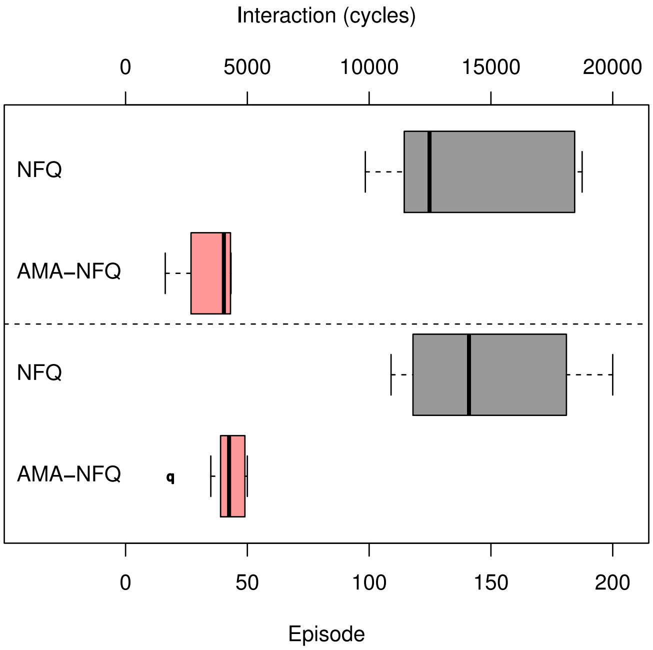

Fig. 4. Number of episodes (bottom) and cycles (top) until generating the first policy that reaches the goal from all initial test states, for AMA-NFQ (red) and NFQ (black). Unsuccessful policy for NFQ not shown.

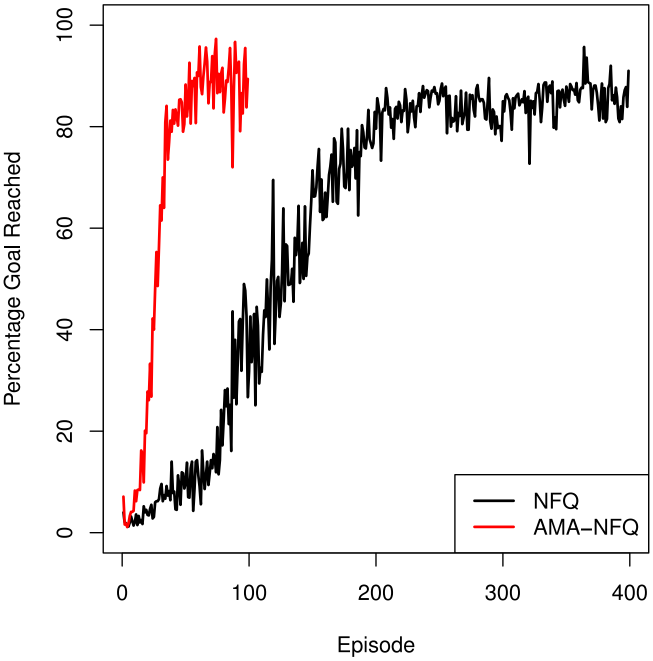

Fig. 5. Average percentage of successful test trials during extended testing for NFQ, with AMA-NFQ performance for the first 100 episodes shown in gray.

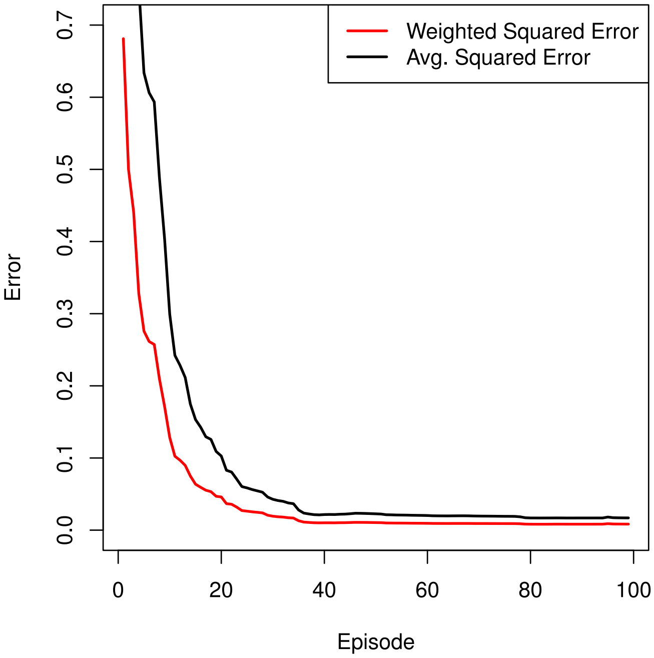

Fig. 6. Development of the model prediction error over time, averaged over 10 experimental runs. For the normalized metric, errors were scaled by the range of their respective inputs.

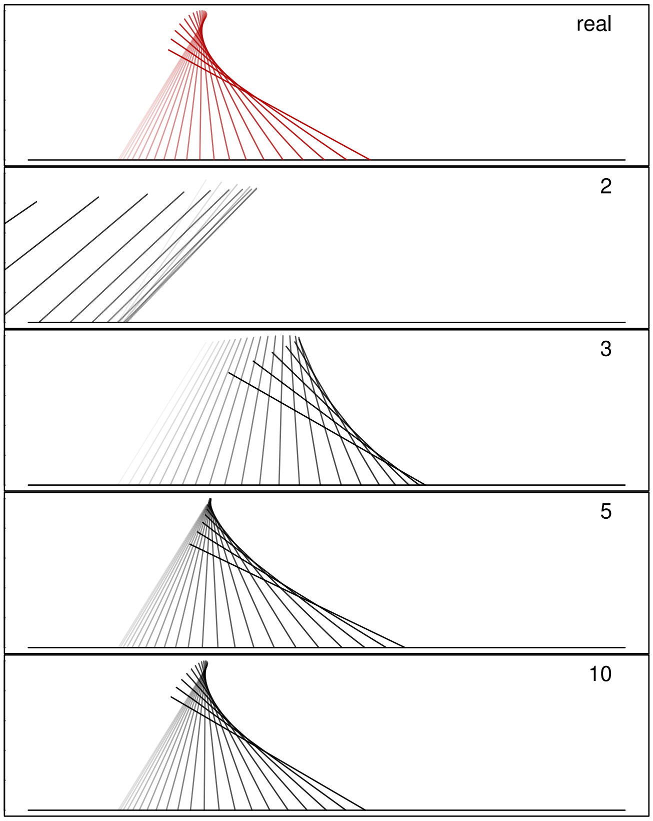

Fig. 7. Example trajectories generated from initial state $\theta = 0.3, p = -0.7$ using constant action $a = 10N$. Top: true trajectory. Second row to bottom: trajectories from different models after 2, 3, 5 and 10 episodes. X-axis shows interval $[1; 1]$, y-axis $[0; 1]$.

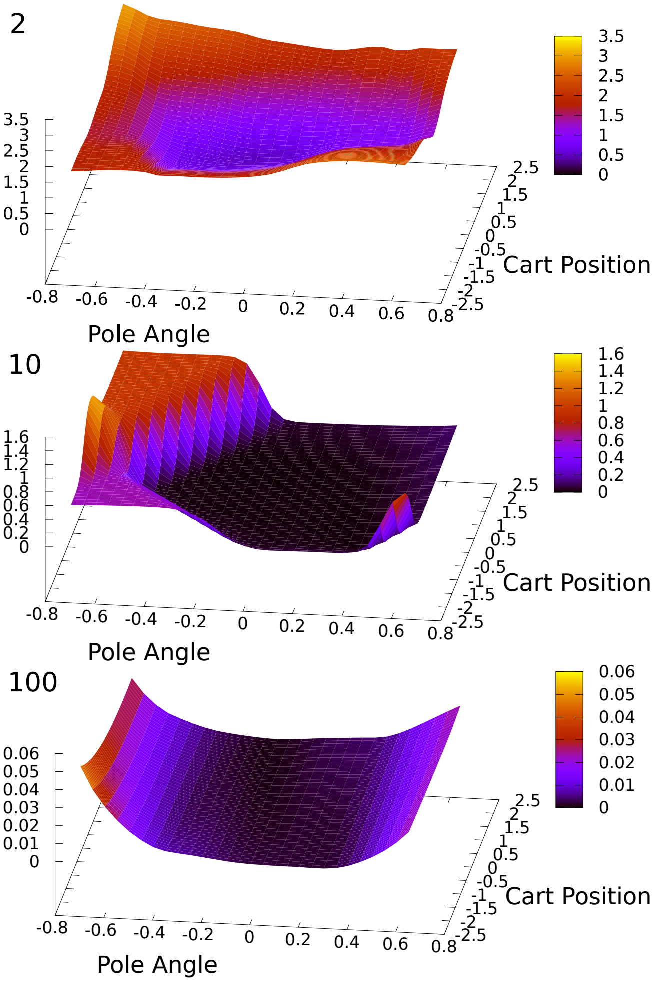

Fig. 8. Prediction error across the working area for three sample models after (from top to bottom) 2, 10 and 100 episodes of training. Values are averaged across $[-0.5; 0.5]$ for velocities and across both actions. Also note the differences in magnitude of the error.

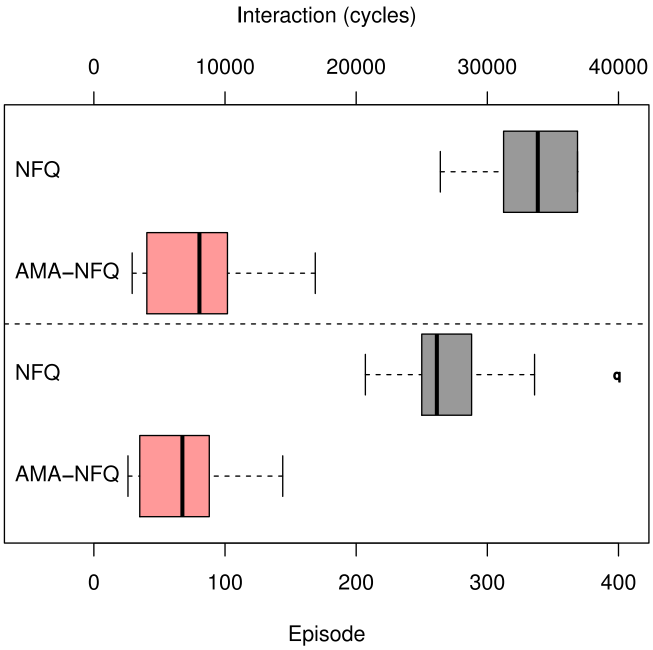

Fig. 9. Number of episodes (bottom) and cycles (top) until generating the first policy that reaches the goal from all initial test states, for AMA-NFQ (red) and NFQ (black) in a two-link manipulator task.

which to fit Q-function and model, was also provided. The AMA-NFQ algorithm was shown to perform consistently and significantly better than model-free NFQ on a cart-pole regulator benchmark, both with respect to performance and to robustness of the learned policies. Within only 100 interaction episodes, our method was able to reliably generate policies that could reach the task's goal from any test position, a feat that model-free NFQ failed to repeat even once. This was made possible by the model's ability to learn an accurate representation of the actual system within a shorter time than required for learning the Q-function directly.

Thus far we have only applied the method to a single benchmark, the cart-pole regulator. It remains to be seen how well it scales up to either more complex tasks or to higher-dimensional action spaces. For the latter case at least, preliminary results indicate that it retains its advantage over classical NFQ. For instance, when applied to the two-link manipulator benchmark [10], which requires control of two degrees of freedom of a simulated robotic arm, the differences in the time required to generate a fully reliable policy illustrated in Fig. 9 mirror those seen earlier in Fig. 2 for the cart-pole regulator. In fact, these results neccessitated the use of the hint-to-goal heuristic for NFQ [2] to allow learning, while no such heuristic was needed for AMA-NFQ.

An application to complex problems currently seems to be limited by the high time requirements, which rise linearly with the number of rollouts to be performed and the number of steps in each rollout. Although the rollouts themselves may be cheap to compute, each is followed by an update (which, to reiterate, is needed to solve the propagation issue we set out to solve). It is worth noting that model-free NFQ is by no means incapable of learning policies of equivalent performance eventually, as Fig. 5 ilustrated, so we are effectively trading a moderate amount of interaction time for a large amount of computation time. We find this acceptable, given that the computation time can be improved through parallelization and larger computational power; it is therefore only a momentary issue. Also, in many real-world applications, interaction time is more expensive due to factors like wear or damage to physical systems. Even so, it should be worthwhile in future refinements to somewhat alleviate the speed issue by reducing the number of rollouts over time, which can easily be done due to the fact that at the core, we still use techniques of model-free Q-learning.

Of course, using the model for rollouts assumes that it provides an accurate representation of the real system, and manages to do so with less data than we would normally need to learn the Q-function. While this was the case here, it is by no means guaranteed. Therefore, we are often at risk of incurring a model bias in the learning process. To avoid such a bias, we intend to employ probabilistic models in the future, which can help taking the uncertainty of the model into account.

So far we have not compared the method with the simple approach of performing multiple inner-loop updates during classical NFQ. In fact, the use of a single starting state for all virtual rollouts of a given episode could potentially introduce a similar bias in our approach. It will therefore be worthwhile to compare both approaches in the future, and determine the increase in performance that the bias reduction resulting from randomized starting states would provide. Preliminary experiments suggest that the use of a model can result in superior performance when the model generalizes well, and equivalent performance at worst in case of overfitting.

Lastly, we have only examined a neural implementation of approximate model-assisted fitted Q-iteration thus far. While it is obviously possible to substitute both model and Q-function by other types of approximators, it remains to be seen if the same advantages are maintained independently of the fitting mechanism.

The current study of Approximate Model-Assisted Neural Fitted Q-Iteration provides only a first look at the approach, and there are multiple issues that remain to be examined. However, it should have become apparent that the method possesses the potential for reducing the amount of data needed for batch Q-learning, and thus for providing more data-efficient methods that are still capable of learning robust control policies in the little amount of training time afforded in real-world applications.

REFERENCES

[1] D. Ernst, P. Geurts, L. Wehenkel, and L. Littman, “Tree-based batch mode reinforcement learning,” *Journal of Machine Learning Research*, vol. 6, pp. 503–556, 2005.

[2] 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*, Porto, Portugal, October 2005, pp. 317–328.

[3] A. Farahmand, M. Ghavamzadeh, C. Szepesvári, and S. Mannor, “Regularized fitted Q-iteration: Application to planning,” in *Recent Advances in Reinforcement Learning*, ser. Lecture Notes in Computer Science, S. Girgin, M. Loth, R. Munos, P. Preux, and D. Ryabko, Eds. Springer Berlin Heidelberg, 2008, vol. 5323, pp. 55–68.

[4] R. S. Sutton, “Integrated architectures for learning, planning, and reacting based on approximating dynamic programming,” in *Proceedings of the Seventh International Conference on Machine Learning (ICML 1990)*. Morgan Kaufmann, 1990, pp. 216–224.

[5] D. Silver, R. S. Sutton, and M. Müller, “Sample-based learning and search with permanent and transient memories,” in *Proceedings of the 25th International Conference on Machine Learning (ICML 2008)*, 2008.

[6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: an Introduction*. MIT Press, 1998.

[7] M. Riedmiller and H. Braun, “RPROP: A fast and robust backpropagation learning strategy,” in *Fourth Australian Conference on Neural Networks*, M. Jabri, Ed., Melbourne, 1993, pp. 169 – 172.

[8] A. G. Barto, R. S. Sutton, and C. Anderson, “Neuronlike adaptive elements that can solve difficult learning control problems,” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-13, no. 5, 1983.

[9] M. Riedmiller, “10 steps and some tricks to set up neural reinforcement controllers.” in *Neural Networks: Tricks of the Trade (2nd ed.)*, 2012, pp. 735–757.

[10] L. Busoniu, B. De Schutter, and R. Babuska, “Decentralized reinforcement learning control of a robotic manipulator,” in *Proceedings of the 9th International Conference on Control, Automation, Robotics and Vision (ICARCV 2006)*, Singapore, 2006, pp. 1347–1352.