Improved Neural Fitted Q Iteration Applied to a Novel Computer Gaming and Learning Benchmark

Thomas Gabel, Christian Lutz, Martin Riedmiller
Machine Learning Lab, Department of Computer Science
University of Freiburg, 79110 Freiburg, Germany
Email: {tgabel|lutzc|riedmiller}@informatik.uni-freiburg.de

***Abstract*—Neural batch reinforcement learning (RL) algorithms have recently shown to be a powerful tool for model-free reinforcement learning problems. In this paper, we present a novel learning benchmark from the realm of computer games and apply a variant of a neural batch RL algorithm in the scope of this benchmark. Defining the learning problem and appropriately adjusting all relevant parameters is often a tedious task for the researcher who implements and investigates some learning approach. In RL, the suitable choice of the function c of immediate costs is crucial, and, when utilizing multi-layer perceptron neural networks for the purpose of value function approximation, the definition of c must be well aligned with the specific characteristics of this type of function approximator. Determining this alignment is especially tricky, when no a priori knowledge about the task and, hence, about optimal policies is available. To this end, we propose a simple, but effective dynamic scaling heuristic that can be seamlessly integrated into contemporary neural batch RL algorithms. We evaluate the effectiveness of this heuristic in the context of the well-known pole swing-up benchmark as well as in the context of the novel gaming benchmark we are suggesting.**

I. Introduction

Computer games have always been an attractive platform for evaluating machine learning algorithms. When having a look back at the two recent decades, one can distinguish two different main goals that have been pursued in the use of adaptive and learning agents within games.

On the one hand, there is the field of well-established games, like card or board games, in the scope of which learning approaches have been utilized for seeking policies that can keep up with the strength of human players. Tesauro's Backgammon playing program TD-Gammon is an early representative of this branch of learning for games [1]. More recent examples cover the games of chess [2] and checkers as well as the intricate game of Go [3], [4] where learning approaches are still far away from the human level of playing.

On the other hand, there is the growing area of research-oriented competitions that are defined to provide a kind of test-bed for artificial intelligence-based, machine learning and, particularly, for reinforcement learning (RL) tasks. The RL Competition held in conjunction with the NIPS conference series, as well as Google's recent AI Challenge represent contemporary examples for research-oriented competitions[1]. One of the most prominent examples are the robotic soccer competitions organized by the RoboCup federation [5]. Here, the overall goal ('win the game') is so complex that it is a natural approach to split the task into several levels of abstraction and different sub-goals. For instance, several research groups have dealt with the task of learning parts of an individual soccer-playing agent's behavior autonomously [6], [7] as well as learning cooperative multi-agent behaviors [8].

Our focus in the paper at hand is on the latter type of learning in games. Our main contributions are:

- We introduce a new, open-source gaming benchmark domain that features two appealing properties. The hurdle of entrance is kept low as the benchmark is based on rather simple and clearly defined interfaces for the designer of the learning agents. Besides, it is embedded into a framework that brings along the requisites that are necessary to develop adaptive and learning agents.
- We have performed an extensive case study applying neural batch mode reinforcement learning algorithms in the scope of the mentioned gaming benchmark. In this context, we have developed a novel scaling heuristic that reduces the implementation effort for the designer of the learning algorithm. We present this heuristic as well as the results we obtained in the scope of our case study.

In Section II we briefly recap the basics of batch-mode reinforcement learning and of the neural fitted Q iteration (NFQ) algorithm. Section III presents the mentioned dynamic scaling technique and argues why this heuristic is beneficial. In Section IV we introduce the Star Ships Learning Framework (SSLF) and highlight its character as a suitable benchmark for learning algorithms. Section V summarizes the results we obtained in using NFQ for learning policies that control the agents participating in the SSLF gaming framework.

II. Batch Reinforcement Learning

Batch reinforcement learning is a subfield of dynamic programming-based reinforcement learning that has grown considerably in importance during the recent years. Historically, the term 'batch RL' is used to describe a reinforcement learning setting, where the complete amount of learning experience – usually a set of transitions sampled from the system – is a priori given and fixed. The task of the learning system is then to derive a solution – usually an optimal policy – from this given batch of samples.

[1] http://www.rl-competition.org/ and http://ai-contest.com/

In contrast to classic online reinforcement learning where updates to the value function are made after each transition, batch reinforcement learning methods store and reuse their experience. That experience is a set of transition tuples consisting each of a state, an action taken in that state, the immediate reward received (or immediate costs interchangeably), as well as the successor state entered. After having collected a batch, i.e. a larger number of transition tuples, the computational update to the value function is performed. Various batch RL algorithms have been proposed in the literature [9]–[11] and batch RL has recently been successfully applied to various challenging real-world applications [12], [13].

Figure 1 sketches the general batch reinforcement learning framework. It basically consists of three main steps that are interconnected by two loops. The step of sampling experience (outer loop) realizes interaction with the environment and creates a set of transition tuples (data). The second step utilizes dynamic programming methods to generate a set of training patterns which are, subsequently and iteratively (inner loop), employed by some batch mode supervised learning algorithm that outputs an approximated function represented by the training patterns. Note that most of the practical realizations of batch reinforcement learning algorithms work in alternating (also called 'growing') batch mode. This means that the outer loop can be iterated an arbitrary number of times, involving corresponding data samplings and successive training phases realized by the inner loop.

The individual components shown in Figure 1 may be implemented differently, e.g. data may be sampled using a greedy or an exploring policy and for the task of fitting a function approximator to the training patterns, in principle, any of a dozen supervised learning algorithms can be utilized.

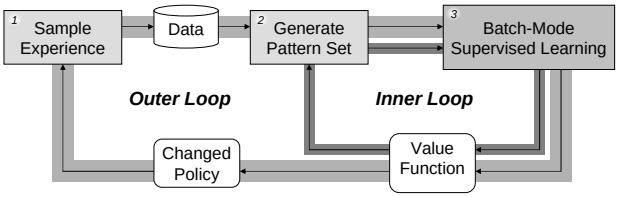

Fig. 1. The Batch-Mode Reinforcement Learning Framework: In growing batch mode, the stages of interaction with the environment (outer loop) and value function learning (inner loop) are interweaved.

A. Fitted Q Iteration

Perhaps the most popular algorithm in batch RL is 'Fitted Q Iteration' (FQI, [14]). As characteristic for batch RL approaches, FQI (when embedded into the inner loop of Figure 1) computes an approximation of the optimal policy from a finite set of four-tuples $\mathbb{T} = \{(s_i, a_i, c_i, s'_i)|i = 1, \ldots, |\mathbb{T}|\}$. These experience units may be collected in any arbitrary manner and they are made up of states s_i, the respective actions a_i taken, the immediate costs c_i incurred, as well as the successor states s'_i entered. The basic algorithm takes $\mathbb{T}$, as well as a regression algorithm as input, and after having initialized $\tilde{Q}$ and a counter k to zero, repeatedly processes the following three steps until some stop criterion becomes true:

1) increment k
2) build up a training set $\mathbb{F}$ for the regression algorithm according to:
$$\mathbb{F} := \{(in_i, out_i)|i = 1, \ldots, |\mathbb{T}|\}$$
where $in_i = (s_i, a_i)$ and $out_i = c_i + \gamma \min_{b \in A} \tilde{Q}^{k-1}(s'_i, b)$
3) use the regression algorithm and the training set $\mathbb{F}$ to induce an approximation $\tilde{Q}^k : S \times A \to \mathbb{R}$

Subsequently, we consider neural fitted Q iteration (NFQ, [15]), a realization of fitted Q iteration where multi-layer perceptron (MLP) neural networks are used to represent the Q function and an enhanced network weight update rule is employed (step 3). NFQ is an effective and efficient RL method for training a Q value function that requires reasonably few interactions with the environment to generate policies of high quality. We will also discuss a heuristic extension of NFQ to be used in the scope of this work in Section III.

B. Neural Fitted Q Iteration

In online Q learning in conjunction with a table-based value function representation, updates to the value function can be made easily on the fly. By contrast, when faced with very large or continuous state-action spaces and when using neural networks as value function approximators, as we intend to do, no direct value assignment for singular state-action pairs can be realized. Instead, an error function must be introduced, which measures the deviations between (a) state-action values approximated by the function approximator (we denote the approximation realized by a neural network as $\tilde{Q}$) and (b) those that are defined through the Bellman equation (cf. the out_i values in the algorithm sketch of FQI in Section II-A). For example, given a single transition tuple (s, a, c, s') and a current value function estimate $\tilde{Q}$ represented by an MLP, the squared Bellman error

$$\left(\tilde{Q}(s, a) - (c + \gamma \in_{b \in A} \tilde{Q}(s', b))\right)^2$$

may be employed, which can be minimized using gradient descent techniques, like the backpropagation algorithm, on the neural network's connection weights.

A drawback of this type of online update is its wasteful utilization of data as each transition tuple is used only once. Thus, typically thousands of episodes are necessary until a policy of sufficing quality is obtained [16]. This disadvantage can be tributed, at least in part, to the global approximation character of MLPs: Weight updates for a certain state-action pair (s, a) may cause unforeseen changes of the value function in very different regions of the joint state-action space.

Of course, the last-mentioned effect can be beneficial in terms of yielding generalization. However, recent studies on the neural fitted Q iteration (NFQ) algorithm have shown, that more stable and reliable learning results can be achieved when learning in batch-mode, while not sacrificing the excellent generalization capabilities of neural networks [15].

As indicated, NFQ is an instance of the class of fitted Q iteration algorithms we described above, where the supervised batch-mode regression algorithm, i.e. the 'fitting' part,

is realized by a multi-layer perceptron neural network. Let $\mathbb{T} = \{(s_i, a_i, c_i, s'_i) | i = 1, \ldots, |\mathbb{T}|\}$ be the set of transition and let $\mathcal{Q}$ denote the space of Q value functions over $S \times A$ representable by MLPs. Then, for each pattern i, NFQ calculates target values out_i as specified in the context of FQI (cf. Section II-A), thereby utilizing the recent Q function estimate Q^{k-1}, and computes the next (kth) iterate of the state-action value function Q^k by minimizing the batch error

$$\sum_{i=1}^{|\mathbb{T}|} (Q^k(s_i, a_i) - out_i)^2. \quad (1)$$

The minimization of this expression is achieved by adapting the connection weights of the neural network representing Q^k,

$$Q^k \leftarrow \arg\min_{f \in \mathcal{Q}} \sum_{i=1}^{|\mathbb{T}|} (f(s_i, a_i) - out_i)^2.$$

As actual neural network training procedure, we utilize the Rprop algorithm [17] which naturally builds on batches of training data and, hence, can be integrated easily into a batch RL algorithm. An important merit of using Rprop, when compared against, e.g. backpropagation, is its insensitivity with respect to learning parameters which rids us from tuning parameters for the supervised part of an FQI algorithm.

III. Neural Fitted Q Iteration with Dynamic Scaling

A practical concern during neural network training is the distribution of the target values (desired outputs out_i). Clearly, in a supervised learning scenario this issue is not critical because the concrete target values are given. In reinforcement learning, however, they – the optimal state-action values – are generally not known beforehand. This point is especially striking when utilizing an iterative batch-mode RL approach, like the neural fitted Q iteration algorithm we are focusing on.

Assume you are implementing a learning approach where you want the state-action value function be represented by an MLP and where you have no prior knowledge about the learning task. Thus, you do neither know about typical episode length nor the length of an optimal episode given some starting state. Moreover, you have no clue about the range of state-action values that the optimal Q function might adopt, i.e.

$$Q^\star_{\min} := \min_{(s,a) \in S \times A} Q^\star(s, a) \text{ and } Q^\star_{\max} := \max_{(s,a) \in S \times A} Q^\star(s, a)$$

are unknown a priori.

This lack of knowledge also affects any single iteration k of a fitted Q iteration method (k denotes the iteration counter), because the distribution as well as the minimum and maximum

$$\begin{aligned} out^k_{\min} &:= \min\{out_i | i = 1, \ldots, |\mathbb{T}|\} \\ out^k_{\max} &:= \max\{out_i | i = 1, \ldots, |\mathbb{T}|\} \end{aligned}$$

of the out_i values (cf. step 2 in the pseudo code in Section II-A) are unknown beforehand, and they are varying from iteration to iteration[2].

When intending to train an MLP, it must be kept in mind, that networks of this type can cover an output range from the interval $I = (0, 1)$ only. Thus, it is the task of the designer of the learning algorithm to ensure that in each iteration k of the NFQ algorithm as well as for each target value out_i it always holds $out_i \in I$. Consequently, the algorithm's designer must be cautious, when defining the cost function such that target values below zero or above one can never occur. This, however, is hard (and tedious) to accomplish when having no knowledge about an optimal policy and about possible minimal or maximal Q values that might occur. Typically, the designer does even not know of how many steps poor, good, or optimal episodes will consist. The straightforward approach of using a pre-defined scaling factor for the out_i values is flawed for the same reason; minimal and maximal state-action values are not known beforehand. Besides, when the algorithm's designer is over-cautious in the way he/she scales the output values and, hence, squeezes them into a small fraction of I (for example such that $out^k_{\min} = 0.2$ and $out^k_{\max} = 0.3$), then the performance of the MLP as function approximator might be affected as well, as not the full range I is exploited. For the reasons mentioned we suggest a dynamic linear scaling mechanism that, in each iteration k, newly determines the way that all output values out_i are to be scaled.

A. Dynamic Scaling Heuristic

Different sigmoidal activation functions of neural networks have different limit values (e.g. $I = [0, 1]$ or $[-1, 1]$). We therefore allow for working with a customizable interval $I' = [net_{\min}, net_{\max}] \subset I$. The task now is to find a mapping from $x \in [out_{\min}, out_{\max}]$ to $y \in I'$ and vice versa as sketched in Figure 2. This is done by the bijective linear function

$$f_{scale} : x \mapsto y = Ax + B$$

with gradient

$$A = \frac{net_{\max} - net_{\min}}{out_{\max} - out_{\min}}$$

and axis intercept

$$B = -\frac{out_{\min} \cdot (net_{\max} - net_{\min})}{out_{\max} - out_{\min}} + net_{\min}.$$

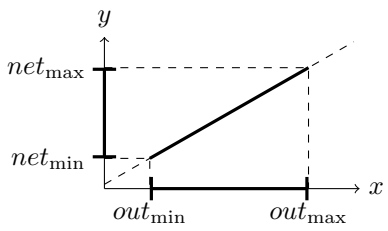

Fig. 2. Mapping $f_{scale} : [out_{\min}, out_{\max}] \rightarrow [net_{\min}, net_{\max}]$

[2]Ideally, in the limit ($k \rightarrow \infty$) it should hold $Q^\star_{\min} = out^k_{\min}$ and $Q^\star_{\max} = out^k_{\max}$, given that $\mathbb{F}$ comprises the best/worst state-action pairs for the task at hand.

The main point is, that the parameters A and B are recalculated (and stored for later use) in each NFQ iteration. The pseudo code in Figure 3 shows where the scaling heuristic is applied (red color). After having randomly initialized the weights of the MLP, we assign initial values to the scaling parameters A and B (lines 1–3). If no a priori knowledge about the task at hand is available, then these initial values can be safely set to $A = 1$ and $B = 0$ such that, initially, actually no scaling is applied at all. Lines 6–13 correspond to the outer batch RL loop and, thus, to interaction between the agent and the environment. New state transitions are collected and added to the existing ones (growing batch) in interaction with the real system or simulation.

Starting in line 16, we provide NFQ-related pseudo code, enriched by the dynamic scaling heuristic we are suggesting. It is important to note that our scaling technique does not just map Q values to some value within $I = (0, 1)$, but that the dynamic mapping between Q values and neural net outputs is anchored in the learning algorithm itself. As a matter of fact, for the purpose of calculating new target values from the current net values (line 20), the inverse back-scaling function $f^{-1}_{scale}(y) = \frac{y-B}{A}$ is used. When all target values are computed, A and B are adapted according to the new minimal and maximal values (lines 22–25) and the target values are scaled by the new scaling function (lines 26–28).

From the perspective of a learning system's designer, the utilization of the proposed scaling heuristic features a number of benefits:

- The cost/reward function can be taken 'as is' (e.g. step costs of 1.0) and does not have to be scaled manually.
- The heuristic will find the 'right' interval, i.e. the true range of the Q value function by itself.
- The whole capacity of the neural net output can be utilized.
- The computing time for scaling operations does not carry weight in comparison to the effort for the supervised fitting of the function approximator.
- There is no drawback in the quality of found solutions compared to an algorithm with a manually (and appropriately) tuned cost function, as we will show empirically.

1 init_net() $\rightarrow \tilde{Q}$
2 init_net_minmax_values() // e.g. $net_{\min} := 0$, $net_{\max} := 1$
3 init_scaling_params() // e.g. default values $A = 1, B = 0$
4 **repeat**
5 // collect state transitions:
6 **repeat**
7 set_startsituation()
8 **repeat**
9 chose_best_action() // exploitation: $a = \arg\min_b \tilde{Q}(s, b)$
10 **or** chose_random_action() // with exploration prob. ϵ
11 $\mathbb{T} := \mathbb{T} \cup \{(s, a, c, s')\}$
12 **until** $s' \in \mathcal{S}^+$ or $s' \in \mathcal{S}^-$
13 **until** *'enough transitions collected'*
14 reinit_net()
15 reinit_scaling_params() // optional
// NFQ-Iterations:
16 **for** $k := 1$ **to** N **do**
17 // compute target values (training set $\mathbb{F}$):
18 **for** $i := 1$ **to** $|\mathbb{T}|$ **do**
19 $in_i := (s_i, a_i)$
20 $out_i := \begin{cases} C^+ & \text{if } s'_i \in \mathcal{S}^+ \\ C^- & \text{if } s'_i \in \mathcal{S}^- \\ c(s_i, a_i, s'_i) + \gamma \min_b Q^{k-1}(s'_i, b) & \text{else} \end{cases}$
// $Q^{k-1}(s'_i, b) = f^{-1}_{scale}(\text{get\_current\_net\_value}(s'_i, b))$
21 **end for**
// apply scaling:
22 $out_{\min} := \min_{i \in \{1,\ldots,|\mathbb{T}|\}} out_i$
23 $out_{\max} := \max_{i \in \{1,\ldots,|\mathbb{T}|\}} out_i$
24 $A := \frac{net_{\max} - net_{\min}}{out_{\max} - out_{\min}}$
25 $B := -\frac{out_{\min} \cdot (net_{\max} - net_{\min})}{out_{\max} - out_{\min}} + net_{\min}$
26 **for** $i := 1$ **to** $|\mathbb{T}|$ **do**
27 $net_i := f_{scale}(out_i)$
28 **end for**
// train new MLP using $\mathbb{F} = \{(in_i, net_i) | i = 1, \ldots, |\mathbb{T}|\}$:
29 reinit_net()
30 Rprop_training() $\rightarrow Q^k$
31 **end for**
32 $Q^N \rightarrow \tilde{Q}$
33 **until** *'behavior successfully learned'*

Fig. 3. Batch-Mode RL Using Neural Network Value Function Approximation: This pseudo code corresponds to both, the outer and inner loop of the batch-mode RL framework (cf. Figure 1), where in the inner loop we employ an extended NFQ variant using the proposed dynamic scaling heuristic.

IV. SSLF: A New Benchmark for RL Algorithms

The Star Ships Learning Framework (SSLF) is an open-source framework[3] that provides basic routines and methods in order to interact with the space shooter game Star Ships [18]. It provides the necessary interfaces for controlling the program's star ships and the program's logic from outside the Star Ships program and, thus, commends itself as benchmark.

A. The Game Star Ships

Star Ships is an action game in which two players take control over two star ships fighting against one another. It is played in a two-dimensional toroidal universe where both players must try – by use of their offensive weapons (phasers as well as photon torpedos) – to weaken the opponent's shields successively until they, finally, collapse and the opponent ship can be destroyed with the next hit. The game play and story behind the game is leant to Star Trek Enterprise (TNG). Of course, each ship may also be controlled not by a human player, but by an intelligent (software) agent or even a learning agent. To facilitate the latter, the SSLF provides the necessary infrastructure in order to allow an intelligent agent – which is entirely decoupled from the Star Ships program, i.e. runs as a stand-alone software – to control one of the ships.

The external agent is informed about the current state of the environment, is expected to think about a good action to take, and of course, to finally execute that action. It can thus realize a sense-think-act loop and, moreover, has the option of improving its strategy by learning.

B. The Framework and its Interfaces

Essentially, the SSLF allows you to create your own agents that play within Star Ships. They interact with the arcade game by reading state information from file (written there periodically by the game engine) and, in turn, writing action information to file (read from there periodically by the game engine). Figure 4 provides a graphical overview of the SSLF.

[3] http://sourceforge.net/projects/sslf/

The state space is high-dimensional and contains several continuous dimensions. The current state is comprised by the following entries.

- time: current time step (integer)
- game state: -1 (game is going on), 0 (game ended, ships collided), 1 (game ended, ship 1 lost), 2 (game ended, ship 2 lost)
- s^1_x, s^1_y, s^1_{vx}, s^1_{vy}: x/y position/velocity of ship 1 (real)
- s^2_x, s^2_y, s^2_{vx}, s^2_{vy}: x/y position/velocity of ship 2 (real)
- s^1_α, s^2_α: angle of ship 1/2 (integer, discretized to 15 degrees, i.e. within [0,15,30,...,345])
- $s^1_{\#t}$, $s^2_{\#t}$: number of alive torpedos shot by ship 1/2 (can take values from within [0...5])
- s^1_p, s^2_p: ship 1/2 is currently hit by a phaser (in [0,1])
- s^1_t, s^2_t: ship 1/2 is currently hit by as many torpedos (can take values from within [0,...,10])
- $\#t$: number of currently active torpedos overall
- t^i_x, t^i_y, t^i_{vx}, t^i_{vy}: x/y position/velocity of the ith torpedo (real)
- t^i_{age}, t^i_{from}: torpedo i is as many time steps old (in [0,...,100]) / descends from that ship (in [1,2])

The following seven actions are available to each ship in each time step: no action (0), fire phaser (2), fire torpedo (2), turn left (0), turn right (0), accelerate by a single impulse (1), warp jump (10), where the costs in terms of consumed energy (initially each ship has 400 energy units) are given in brackets.

The simulation is time discrete with alternating action stages (for ship 1 and 2, respectively) lasting 50ms each. Movements are deterministic, but hits by phasers as well as torpedos cause stochastic damage to the ships (damage in terms of reducing the available energy by some random value from [11,16]). The SSLF does not provide access to the system model, i.e. a learning agent would either have to infer the model or to use a model-free learning approach. However, it allows for setting arbitrary states which is a fundamental requirement when pursuing a learning approach.

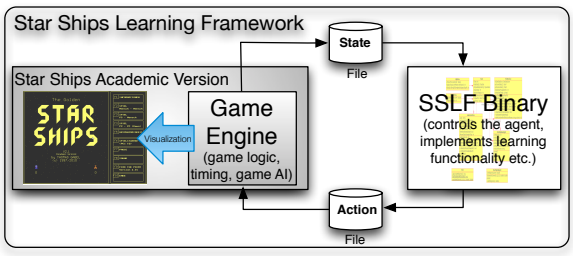

Fig. 4. Components of the Star Ships Learning Framework

The SSLF comprises a base class from which own agents can be derived easily. It also brings along one such example hand-coded agent (called 'AI agent' subsequently), whose playing strength is already superior to a well-trained human player. This agent is provided in order to extend it, i.e. it should be interpreted as a base line AI player that is to be enhanced by certain clever, machine-learned sub-behaviors. In the remainder of this paper we report on our experiences in doing exactly this: We identified two particular behaviors (an offensive and a defensive one) and made the AI agent learn them autonomously using the NFQ algorithm with dynamic scaling presented in the preceding sections.

V. Empirical Evaluation

In this section, we evaluate the use of neural batch-mode reinforcement learning with the dynamic scaling heuristic we proposed. We start off with a proof of concept by illustrating the basic functioning of NFQ with dynamic scaling and by presenting some results we obtained for the well-known pole swing-up benchmark.

In the second part of this empirical evaluation, we focus on the SSLF benchmark introduced in Section IV. Throughout all experiments in the context of the Star Ships Learning Framework, our dynamic scaling heuristic is employed. We finish this evaluation by a comparison of the playing performance of a hand-coded SSLF agent playing against an agent whose strategy incorporates the behaviors we learned using neural batch-mode RL with dynamic scaling.

A video illustrating and explaining the learning process as well as the empirical results achieved, when embedding the learning results into SSLF's base AI agent, is available as supplementary material to this paper[4].

A. Proof of Concept: The Pole Swing-Up Benchmark

In this well-known benchmark a pole is fixed at one ending and a mass (precisely, a mass point) is mounted at its opposite end. We use the established setting where the pole has a length of $0.5\,\mathrm{m}$ and the mass mounted onto it has a mass of $2\,\mathrm{kg}$. The task is to swing up the pole out of the idle position using small torques. To make the task sufficiently challenging we decided on two available actions $-1\,\mathrm{Nm}$ and $+1\,\mathrm{Nm}$. The state space comprises two dimensions, an angle out of $[-\pi, \pi]$ (0 denoting the upright position) and an angular velocity.

The criterion for optimality is the number of time steps needed to swing up. Hence, each action has constant costs c. In the goal region (angle within $[-0.2, 0.2]$) no costs arise ($C^+ = 0$). The problem now is that the interval $[Q^\star_{\min}, Q^\star_{\max}]$, in particular $Q^\star_{\max}$, is unknown a priori as we do not know an optimal solution yet. We expect $Q^\star_{\max}$ to correspond to the time steps an optimal solution would need to swing up the pole. The common approach would be to try different values for c, to fit the value function to the output range [0,1]. For the purpose of comparison, we included this (tedious) methodology in our experiments. Moreover, we applied our dynamic scaling heuristic.

1) Experimental Setup: We employ a fixed pattern set $\mathbb{T}$ containing 1862 transitions that has been generated as follows: starting from each angle in $[-3.1, -3, -2.9, \ldots, 3.1]$ with velocity 0, two sequences are run executing always the same action (-1 and $+1$, respectively) up to 15 time steps or until reaching the goal. With this set a wide area of the state-action space is covered by $\mathbb{T}$. A basic NFQ algorithm with

[4]http://bit.ly/dBxy3L or http://www.youtube.com/watch?v=rgYwP5slMH4

$N = 250$ iterations is executed for different definitions of the cost function ranging from $c = 0.1$ to 0.001. We set up our extended algorithm with dynamic scaling (denoted as DYN) with $[net_{\min}, net_{\max}] = [0, 1]$. A and B are initialized with default values 1 and 0. The cost function is arbitrarily set to $c = 1$. In all experiments an MLP with 3 inputs, one hidden layer with 9 neurons, and one output is used. The discount factor is set to $\gamma = 1.0$.

2) Results: As shown in Figures 5 and 6, if the direct costs are 'over-valued' ($c = 0.1$ and $c = 0.05$), the value function does not fit into the output range of the MLP, which results in unsuitable solutions. By contrast, the dynamic scaling heuristic is able to determine the boundaries of the resulting value function automatically and scales the values exactly to the output range of the MLP. The results are therefore nearly identical to those of having chosen manually a 'suitable' cost function (e.g. $c = 0.02$ and $c = 0.05$).

It should be noted that NFQ with dynamic scaling needs some more iterations to adjust the upper bound $out_{\max}$. This results in a slightly flatter slope of the learning curve (within the inner batch RL loop) and must be tributed to the fact that $out_{\max}$ starts from 1 and is incremented by $c = 1$ in each iteration whereas the optimal solution needs $Q^{\star}_{\max} = 76$ time steps. As a consequence, the share of improper policies is higher during the early NFQ iterations.

To this end, however, the learning process could be easily sped up by choosing better initial scaling parameters (e.g. $A = 0.01$ and $B = 0$ as visualized in Figure 5 as $\text{DYN}_{0.01,0}$). In general, for a fixed data set, this is no option as it would mean to tune parameters again (which is exactly what we wanted to omit by introducing the dynamic scaling technique). But, if we consider a growing batch approach, we may assume that the interval of the value function will not change significantly from one pass of the outer loop to the next one. This information could be used by keeping the parameters A and B of the previous pass as initial values of the current pass (which is actually done by omitting line 15 in Figure 3).

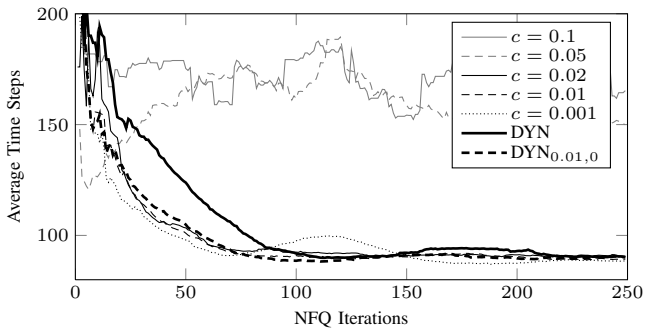

Fig. 5. We considered a fixed transition data set ($|\mathbb{T}| = 1862$) and different definitions of the function c of direct costs as well as the dynamic scaling heuristic (DYN) which is, by definition, entirely independent of the exact numerical values of the direct costs. For each configuration, the entire experiment was repeated 10 times, forming averages. After every NFQ iteration the current policy was evaluated. This graph shows the time steps needed to swing up, averaged over the previous 50 NFQ iterations. Improper policies (not reaching the goal within 400 steps) were filtered out.

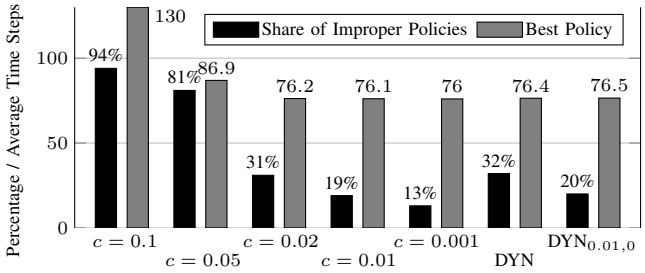

Fig. 6. This diagram depicts the share of improper policies that were filtered out in Figure 5 as well as the number of time steps that were needed to swing up the pole out of the idle state for the best solution found within 250 iterations. The values are averaged over all 10 repetitions of the experiment.

B. Learning Partial Behaviors

Next, we turn to the use of RL approaches in the scope of the proposed gaming benchmark and, thus, investigate several learning problems in the Star Ships Learning Framework.

As the state space is large (cf. Section IV-B) and the overall task (winning the game) is too complex we had to identify relevant subtasks for an agent playing this game. We focused on (1) evading and (2) firing torpedos, as the exposure to torpedos is – beside close combat using phaser attack which is beyond the scope of this paper – decisive for the outcome of a match. Both subtasks were further divided and learned in different levels of complexity.

1) Torpedo Evasion: The aim here is to evade a torpedo that crosses the trajectory of the own ship consuming a minimum of energy. The underlying MDP consists of the following components:

- Our state space S has four dimensions: the distance between the own ship and the torpedo, the orientation towards the torpedo, and the relative velocity (x and y) assuming a ship-centered coordinate system with the x-axis running through the torpedo.
- Offensive actions are not relevant for this task. We also want to optimize the learned behavior in terms of energy usage. We therefore surrender the expensive warp jump (ten units of energy). Thus, the set A of possible actions is restricted to rotations and impulses only.
- The transition model p is deterministic in this case, since the opponent is not considered. However, it is unknown to our learning agent. Hence, a model-free learning approach must be taken.
- The cost function c corresponds to the energy usage of the respective action. Thus an impulse costs 1 unit. To bring in a time aspect (the agent should also evade within the fewest time steps) costs of 0.25 are assigned to rotations (which do not consume energy).
  If the ship is hit by the torpedo, final costs of 13.5 arise[5]. Else, i.e. if the agent has evaded successfully the sequence ends with no final costs.
- A discount factor $\gamma = 0.95$ is introduced.

[5]expected/average loss of energy

A set of 5000 training start situations was generated as follows: If the ship does not execute any action (impulse), it will be hit by the torpedo. Both the velocity of the ship and the torpedo as well as the orientation are randomized such that different angles of striking occur. There is a minimum of 6 time steps until collision.

The behavior is learned using the algorithm described in section III-A. An MLP with 5 input neurons (4 for the state, 1 for the action), one hidden layer with 10 neurons and one output was used. The learning process is subdivided in training units (TU), which correspond to cycles of the outer loop (cf. Figure 1). In each TU 500 transitions (about 100 episodes depending on the current performance) are collected and added to $\mathbb{T}$. Actions are selected exploiting the current function approximator and choosing random actions with a probability of $\epsilon = 10\%$. Afterwards a new MLP is trained within $N = 20$ NFQ iterations. Both the newly added and already existing transitions are used (growing batch).

The results of this process are shown in Figure 7. After about 10 training units, the agent has learned to evade the torpedo in 86% of the start situations. Furthermore, the learned value function provides information about whether it is still possible to evade. A high value indicates that there is probably no way to evade using only rotations and impulses. This information could be used when embedding this sub-behavior manually into an entire strategy.

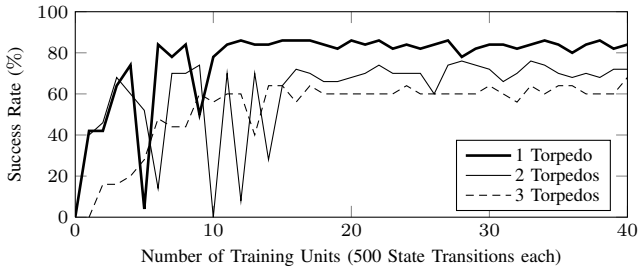

Fig. 7. Learning Processes of the Torpedo Evasion Behaviors: After each training unit the current neural network was evaluated on an independent set of 50 start situations. 'Success' means the agent was able to evade all 1, 2 or 3 torpedos.

As the own ship might be threatened by more than one torpedo at the same time one has to think about how to deal with such a case. For example, considering only the nearest torpedo might result in a maneuver that routes the ship directly towards another torpedo. We therefore decided to learn separate sub-behaviors for the evasion of 2 and 3 torpedos in a similar way. Start situations are generated with 2 and 3 torpedos, each starting from a different (random) direction. They might hit the ship staggered by few time steps. An episode ends, if the agent succeeds in evading *all* torpedos or if it is hit by one of them. The state space thus rises to 8 and 12 dimensions and the number of hidden neurons of the MLP is increased to 15 and 20, respectively. This way we made the learning task far more challenging. Nevertheless, the agent succeeds in achieving noticeably high success rates (cf. Figure 7).

2) Torpedo Attack: The goal in this subtask is to orient the own ship and fire a torpedo directly into the course of the opponent ship. As a proof of concept we start with both ships standing still and the opponent being passive. The state space thus comprises only two dimensions, the distance and the orientation towards the opponent. Actions are restricted to left and right rotations (at costs of 0.5 for enforcing time optimality) and to firing off a torpedo (2 energy units), which terminates the current sequence from the agent's perspective (in practice we have to wait some time steps to observe the outcome). Additional costs (reward) of -13.5 arise, if the opponent is hit by the torpedo. This time we use three MLPs (one for each action) with the topology 2–7–1 to approximate the value function. The results are shown in Figure 8.

Interestingly, the success rate (share of torpedos that hit the opponent) does not grow beyond 60% in this artificially simplified scenario with stationary opponent ships. This is due to the fact that the rotations are discretized in steps of $15°$. As a consequence, there are many situations in which the stationary opponent cannot be reached by the torpedo because the direct orientation towards the opponent lies disadvantageous between two steps of the discretization – one might say the opponent is hiding in a 'blind spot'.

This changes, however, when both ships are provided with a random, but constant velocity as in the following experiment. Now the state space is extended by two dimensions for the relative velocity (v_x,v_y) between the ships. The agent successfully learns in which direction it has to fire off the torpedo to actually hit the constantly flying opponent in remarkably many situations (cf. Figure 8).

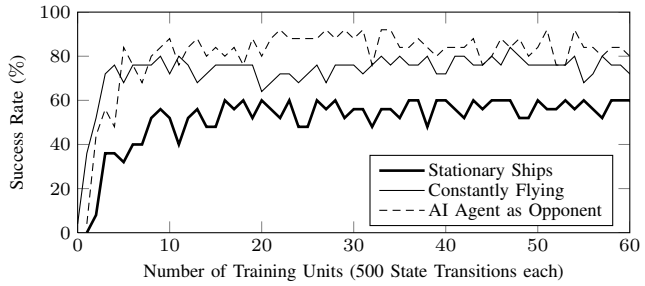

Fig. 8. Learning Processes of the Torpedo Attack Behaviors: After each training unit the current neural network was evaluated on an independent set of 50 start situations. 'Success' means the opponent ship was hit by the torpedo.

As a next step we want to provide the opponent with some kind of intelligence. The first choice is the hand-coded AI player (dubbed 'AI agent') that is already delivered with the SSLF. To avoid disturbing the learning process we manually disable the offensive actions (phaser and torpedo) and the unpredictable warp jump of the opponent by replacing them with 'no action'. The state space is expanded by the orientation of the opponent towards the own ship to enable the learning agent to predict the action of the opponent. In fact, the learning agent is able to adjust itself to the strategy of the opponent and hit the enemy ship in up to 9 out of 10 situations.

C. Evaluation of the Hybrid Agent

As mentioned, the SSLF already brings along a hand-coded AI player which, as a matter of fact, is already hard to beat for a human player. In what follows, we dedicatedly improve the capabilities of this AI agent by embedding the two RL-optimized sub-behaviors described in the previous sections.

First, the AI agent should be enhanced by an intelligent sub-behavior for torpedo evasion. Whenever the ship is in danger of one or more torpedos, the learned evasion for the corresponding number of torpedos takes control. In the very rare case of more than three torpedos threatening the ship only the first three are considered. If the minimum state-action value is greater than 10 (indicating a low probability for evading in a conventional way) a warp jump is executed. As shown in Figure 9 the AI agent (AIA) is considerably improved by employing the learned evasion although this sub-behavior is only activated 9.4% of the time.

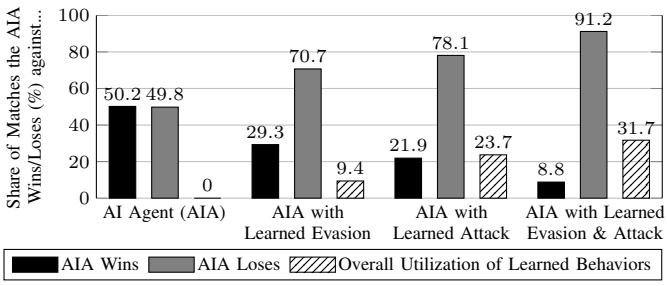

Fig. 9. Playing against itself, the AI agent (AIA) wins on average one out of two matches. By contrast, when both learned sub-behaviors (evasion and attack) are integrated, this agent wins 10 out of 11 matches against the AIA. The share of actions chosen by the learned sub-behaviors is also shown.

Next, we enhance the AI agent by the torpedo attack trained against itself. The learned attack is employed whenever the distance of the two ships is greater than the maximal phaser length (i.e. close combat is disregarded and left untouched). Otherwise the hand-coded behavior of the original AIA takes account which will turn towards the opponent and fire phasers with a high probability, if the distance is advantageous.

Finally, both learned sub-behaviors are integrated into the AIA. As above, the evasion is considered, if the agent is threatened by at least one torpedo. Else, the learned, RL-based torpedo attack or the hand-coded close-combat behavior of the AIA are applied. The performance of the resulting player is astonishing. It wins 10 out of 11 matches against the hand-coded AI player (cf. Figure 9) and thus makes a point for the strength of neural batch-mode reinforcement learning.

VI. Conclusion

In this paper, we have made two contributions. First, we have suggested a new dynamic scaling heuristic for neural batch-mode RL algorithms like NFQ. This heuristic rids the learning algorithm's designer from the tedious task of aligning the learning task's function of immediate cost with the properties of the neural value function approximator. We have shown empirically that this dynamic scaling mechanism yields superior performance when compared to a poorly-adjusted definition of direct costs. By contrast, if the learning algorithm's designer has done a good job and employed a suitable definition of the immediate cost function, then the quality of the resulting policies is equal to what our scaling heuristic achieves.

Our second contribution has been the introduction of the Star Ships Learning Framework, a novel gaming benchmark domain for reinforcement learning algorithms. We have successfully evaluated our dynamic scaling-based NFQ variant in the context of this benchmark. Besides, we envision that the SSLF may be conveniently utilized by lecturers as an attractive tool for teaching purposes and by other researchers and practitioners as a benchmark to compare various RL approaches against one another.

References

[1] G. Tesauro, "TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play," *Neural Comp.*, vol. 6, pp. 215–219, 1995.

[2] M. Campbell, A. Hoane, and F. Hsu, "Deep Blue," *Artificial Intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.

[3] B. Bouzy and T. Cazenave, "Computer Go: An AI Oriented Survey," *Artificial Intelligence*, vol. 132, no. 1, pp. 39–103, 2001.

[4] Y. Wang and S. Gelly, "Modifications of UCT and Sequence-Like Simulations for Monte-Carlo Go," in *Proceedings of the IEEE Symposium on Computational Intelligence in Games (CIG 2007)*. Honolulu, USA: IEEE Press, 2007, pp. 175–182.

[5] M. Veloso, T. Balch, and P. Stone, "RoboCup 2001: The Fifth Robotic Soccer World Championships," *AI Magazine*, no. 23, pp. 55–68, 2002.

[6] G. Kuhlmann and P. Stone, "Progress in Learning 3 vs. 2 Keepaway," in *RoboCup-2003: Robot Soccer World Cup VII*. Berlin: Springer, 2004.

[7] M. Riedmiller, T. Gabel, R. Hafner, and S. Lange, "Reinforcement Learning for Robot Soccer," *Autonomous Robots*, vol. 27, no. 1, pp. 55–74, 2009.

[8] M. Riedmiller and T. Gabel, "On Experiences in a Complex and Competitive Gaming Domain: Reinforcement Learning Meets RoboCup," in *Proceedings of the 3rd IEEE Symposium on Computational Intelligence and Games (CIG 2007)*. Honolulu, USA: IEEE Press, 2007, pp. 68–75.

[9] G. J. Gordon, "Stable Fitted Reinforcement Learning," in *Advances in Neural Information Processing Systems*, D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, Eds., vol. 8. The MIT Press, 1996, pp. 1052–1058.

[10] D. Ormoneit and S. Sen, "Kernel-Based Reinforcement Learning," *Machine Learning*, vol. 49, no. 2, pp. 161–178, 2002.

[11] M. Lagoudakis and R. Parr, "Model-Free Least-Squares Policy Iteration," in *Proceedings of Neural Information Processing Systems (NIPS2001)*, Vancouver, Canada, 2001, pp. 1547–1554.

[12] M. Riedmiller, M. Montemerlo, and H. Dahlkamp, "Learning to Drive a Real Car in 20 Minutes," in *Proceedings of Frontiers in the Convergence of Bioscience and Information Technologies (FBIT 2007)*. Jeju Island, South Korea: IEEE Computer Society, 2007, pp. 645–650.

[13] M. Deisenroth, J. Peters, and C. Rasmussen, "Approximate Dynamic Programming with Gaussian Processes," in *Proceedings of the 2008 American Control Conference (ACC 2008)*. Seattle, USA: IEEE Press, 2008, pp. 4480–4485.

[14] D. Ernst, P. Geurts, and L. Wehenkel, "Tree-Based Batch Mode Reinforcement Learning," *Journal of Machine Learning Research*, no. 6, pp. 504–556, 2005.

[15] M. Riedmiller, "Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method," in *Machine Learning: ECML 2005*. Porto, Portugal: Springer, 2005.

[16] ——, "Concepts and Facilities of a Neural Reinforcement Learning Control Architecture for Technical Process Control," *Neural Computation and Application Journal*, vol. 8, pp. 323–338, 1999.

[17] M. Riedmiller and H. Braun, "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm," in *Proceedings of ICNN 1993*, San Francisco, USA, 1993, pp. 586–591.

[18] T. Gabel, "Star Ships Learning Framework Manual," 2010, `http://sourceforge.net/projects/sslf/`.