Ensembles of Neural Networks for Robust Reinforcement Learning

Alexander Hans
Neuroinformatics and Cognitive Robotics Lab
Ilmenau University of Technology
Ilmenau, Germany
Email: alexander.hans.ext@siemens.com

Steffen Udluft
Intelligent Systems and Control
Siemens AG, Corporate Technology
Munich, Germany
Email: steffen.udluft@siemens.com

Abstract—**Reinforcement learning algorithms that employ neural networks as function approximators have proven to be powerful tools for solving optimal control problems. However, their training and the validation of final policies can be cumbersome as neural networks can suffer from problems like local minima or overfitting. When using iterative methods, such as neural fitted Q-iteration, the problem becomes even more pronounced since the network has to be trained multiple times and the training process in one iteration builds on the network trained in the previous iteration. Therefore errors can accumulate. In this paper we propose to use ensembles of networks to make the learning process more robust and produce near-optimal policies more reliably. We name various ways of combining single networks to an ensemble that results in a final ensemble policy and show the potential of the approach using a benchmark application. Our experiments indicate that majority voting is superior to Q-averaging and using heterogeneous ensembles (different network topologies) is advisable.**

I. INTRODUCTION

In reinforcement learning (RL) [1] one is concerned with an agent interacting with an environment. The agent has the ability to observe the environment's current state and can influence it by carrying out actions. Each action causes a transition to a new state. Along with the transition the agent receives a reward, giving it a hint as to how useful that transition was. However, this reward is not necessarily only dependent on the last action. Instead, it usually is delayed and therefore the result from a series of actions.

Often an RL problem is formulated as a Markov decision process (MDP), consisting of a state space S, an action space A, a transition probability distribution $P_T : S \times A \times S \mapsto [0, 1]$, and a reward function $R : S \times A \times S \mapsto \mathbb{R}$. When all components of the MDP are known, an optimal policy can be determined, e.g., using dynamic programming. Otherwise, observations from the MDP must be sampled by interacting with it (exploration).

One distinguishes an online and an offline setting in RL. While in the online setting the interaction with the MDP and the determination of the optimal policy are performed by one agent, potentially using every observation immediately to update the policy, in the offline setting one is given a set of observations generated by an arbitrary exploration strategy and has to try to derive an optimal policy from that set. In this paper we deal with the offline setting (also known as batch-mode RL).

To determine the optimal policy, many RL methods calculate a so-called Q-function as an intermediate step. The Q-function gives the expected long-term reward of a given state-action pair for a fixed policy. The aim is then to utilize the Q-function to derive the optimal policy, e.g., by dynamic programming. If the state and action spaces are discrete and the number of states sufficiently small, the Q-function can be stored and calculated in a tabular way. If, however, one deals with large discrete or continuous state spaces, it is inevitable to resort to function approximation, for two reasons: first to overcome the storage problem, second to achieve data-efficiency (i.e., requiring only few observations to derive a near-optimal policy) by generalizing to afore unobserved states-action pairs.

The class of fitted Q-iteration (FQI) methods [2] has proven particularly data-efficient. Implementations of FQI used, for instance, tree-based regression [3], linear architectures on features [4], and neural networks [5]. In FQI one formulates the problem of learning the Q-function as a set of regression tasks. In each iteration an approximator is first learned and then used to determine the targets for the next iteration. This is repeated until the desired precision is reached.

However, function approximators like neural networks suffer from a number of problems and can fail at finding the correct mapping from input to target data. Dietterich [6] gives three sources of failure: 1) The statistical problem: when only few training patterns are available, a solution that nicely fits both training and validation sets can still be away from the real function. 2) The computational problem: many algorithms try to optimize a non-convex error function that exhibits local minima; by starting from different points in the parameter space and randomly selecting patterns for training, even given the same training data different instances of the same algorithm can arrive at different solutions. 3) The representational problem: the function approximator might be unable to represent the actual function. In supervised learning ensembles of learners have been successfully used to tackle all of those three problems, both for classification

and regression [6].

We believe that RL can also benefit from ensembles. However, so far only few contributions exist that employ ensembles for RL. Wiering and van Hasselt used ensembles of quite different algorithms trained with the same data for discrete MDPs in an online setting [7]. Each algorithm determines its own policy, the final policy is determined from the individual policies, e.g., using majority voting. Ernst et al. used ensembles of regression trees in an FQI approach [3]. Instead of using only one regression tree to represent the Q-function in each iteration, they used an ensemble of trees (random forest) [8].

In this paper, we propose to use ensembles for neural fitted Q-iteration, an FQI method employing neural networks. We name various ways of combining single networks and give empirical results for the pole balancing problem.

II. NEURAL FITTED Q-ITERATION

Neural fitted Q-iteration (NFQ) [5] is an instance of the fitted Q-iteration (FQI) [2] approach. FQI does value iteration based on samples of the MDP. When dealing with a discrete MDP, value iteration means repeatedly applying an update rule based on the Bellman optimality equation:

$$Q^{k+1}(s,a) := \sum_{s' \in S} P(s'|s,a) \left[R(s,a,s') + \gamma \max_{a' \in A} Q^k(s',a') \right],$$

where γ is the discount factor. Starting with an arbitrarily initialized Q^0, e.g., $Q^0 := 0$, with $k \to \infty$, Q^k converges to Q^*, the Q-function of the optimal policy $\pi^*(s) := \arg\max_a Q^*(s,a)$. If the state space is large or continuous, one has to resort to function approximation to represent the Q-function. FQI then means repeatedly training a new function approximator for the Q-function and using the learned approximator to determine the targets for the next iteration (for the first iteration the Q-function is assumed to be constantly zero and only the rewards are learned). Figure 1 summarizes the FQI algorithm. Note that the sum over successor states and transition probabilities are not considered explicitly here. Instead, they are implicitly given by the set of observations $((s,a,r,s')$ tuples).

The main characteristic of NFQ is using a neural network (multi-layer perceptron) as function approximator. The generalization capabilities of neural networks are excellent, therefore it is possible to produce near-optimal policies with only few observations of the MDP [5]. Another powerful approach is neural rewards regression (NRR) [9]. NRR uses a special architecture with shared weights and can learn the Q-function in a single step without the need for iteration.

However, for both, NFQ and NRR, one has to choose a suitable network topology (e.g., number of hidden layers, number of neurons in each layer) and learning algorithm. Moreover, one can influence the learning process, e.g., either learn using a constant learning rate or start with a large learning rate and reduce it later.

Figure 2 shows the NFQ implementation used in this paper. In each iteration the targets are scaled to lie within $[-1, 1]$, before using the output of a network, the scaling is reversed. In the first iteration only the rewards are learned by assuming $Q^0 = 0$.

- $\hat{Q}^0 := 0$
- $i := 0$
- while the desired precision is not reached
  - $\text{input}_j := (s_j, a_j)$
  - $\text{target}_j := r_j + \gamma \max_a \hat{Q}^i(s_j', a)$
  - $\hat{Q}^{i+1} := \text{train}(\text{input}, \text{target})$
  - $i := i + 1$
- return $\hat{Q}^i$

Figure 1. The basic FQI algorithm. $\hat{Q}^k$ is a function approximator, $\hat{Q}^k(s,a)$ gives the value of $\hat{Q}^k$ evaluated with input (s,a). input and target are arrays containing the training samples based on the $j = 1 \ldots N$ observations of the MDP.

- $\text{input}_j := (s_j, a_j)$
- $\text{target}_j := r_j$
- $\text{target} := \text{scale}(\text{target})$
- $\text{net}_Q^0 := \text{train\_net}(\text{input}, \text{target})$
- $i := 0$
- while $i < M$
  - $\text{target}_j := r_j + \gamma \max_a \text{unscale}(\text{net}_Q^i(s_j', a))$
  - $\text{target} := \text{scale}(\text{target})$
  - $\text{net}_Q^{i+1} := \text{train\_net}(\text{input}, \text{target})$
  - $i := i + 1$
- return net_Q^i

Figure 2. Our NFQ implementation. Again, input and target are arrays containing the training samples based on the $j = 1 \ldots N$ observations of the MDP. scale() scales the targets to lie within $[-1, 1]$, unscale() reverses the scaling (it therefore needs to know the scale factor of the previous scale() operation). train_net() trains a neural network for the given inputs and targets. $\text{net}_Q^i(s,a)$ denotes evaluating the network net_Q^i using inputs (s,a).

III. INSTABILITY OF THE LEARNING PROCESS

There have been a number of reports on problems with the learning process with FQI in general and NFQ in particular [10]–[13]. When using function approximation for RL, a phenomenon called *chattering* can occur [10]. The space of possible Q-functions for an MDP representable by a function approximator contains so-called *greedy regions*. In such a region the policy resulting from greedy exploitation of the respective Q-function, i.e., following $\pi(s) = \arg\max_a Q(s,a)$, does not change. Each greedy region contains a *greedy point*, during the process of learning the Q-function represented by the function approximator moves to that point. If the greedy point lies within the greedy region, no problems arise. However, if the greedy point lies on the border of another or even outside the greedy region, an

oscillation can occur with the Q-function moving from one greedy region to another. For NFQ this problem has been observed as well [11] and also matches our experience. In [11] the authors suggest doing a policy selection by monitoring the current policy's quality and stopping the learning process once the quality declines. Their method works by calculating a sample of the optimal Q-function tabularly and comparing the ranking of actions of the tabular Q-function with the neural one. They conclude that the closer the match, the better the neural Q-function. While the approach is indeed able to stabilize the learning process and produce high-quality policies more reliably, its major drawbacks are the limitation to discrete state spaces and the necessity of having observed multiple actions in the same state. We believe that in addition to chattering also the general problems of function approximators named by Dieterich [6] contribute to the instability of RL with function approximation. However, for the pole balancing benchmark (see section V-A) even with approximately 60,000 observations (10,000 episodes), a number that here excludes the statistical problem, we could observe oscillations. While the vast majority of iterations produced successful policies, occasionally there was a policy that was unable to balance the pole for the required 3,000 steps. Another effect that contributes to the problems is the overestimation of Q-values [12]. When learning with noisy data, the output of a function approximator will also be affected by noise. Although the noise has a mean of zero, an FQI-like algorithm will systematically overestimate the Q-value as it selects the maximum Q-value over all actions when determining the targets for the next iteration. Thus the noise is maximized as well. This problem is also known as the "rising Q problem" [13].

To mitigate those problems, we want to use ensembles for more robust and reliable RL with function approximation. This also makes the algorithm less sensitive to the possible choices of parameters.

IV. Ensembles in NFQ

Ensembles have been used in supervised learning to improve the performance of a single learner by combining several ones. For classification problems a possible way to combine the single learners is (weighted) majority voting. If their errors are not strongly correlated, in expectation the performance of the ensemble will in general be better than that of any single learner. To provide the necessary diversity among learners (and therefore hopefully not strongly correlated errors), one can use ensemble methods like bagging [14] or boosting [15]. In bagging the training set is split into separate (possibly overlapping) partitions, each learner is trained on a different partition. Boosting goes one step further by training one learner after another and giving so far misclassified examples a higher weight. This way learners trained in the beginning cover the general "easy" training examples and later trained learners become "experts" for

certain cases. The final decision is made by a weighted majority voting, where the weight of each single learner is dependent on its performance on the complete dataset. In the case of neural networks even simply training the network multiple times on the same training set leads to some diversity because of the random initialization of the network's weights and the random selection of patterns during learning. Other possibilities of introducing diversity into an ensemble of neural networks include varying the network's topology (number of hidden layers, number of neurons per layer, randomly sparse initialization of weight matrices [16]), the learning algorithm, the learning rate, and regularization techniques like weight decay or early stopping.

To combine the networks of an ensemble to a final policy, one can think of various possibilities. In particular, the aggregation method can be varied.

Combination of final policies: It is possible to let each instance of the algorithm run for itself until a final policy or Q-function is determined and then combine those to obtain the final policy. This method was used in [7]. It makes no assumptions about the algorithm and is therefore also suitable for combining algorithms that use a different notion of a Q-function (e.g., actor-critic algorithms) or no Q-function at all (e.g., the recurrent control neural network [17]). An easy solution for combining policies is (weighted) majority voting. In [7] a number of additional methods for combining policies are proposed.

Combination of final Q-functions: When combining learners that use a Q-function, one can combine the single Q-functions to an ensemble Q-function and base the final policy on that. This can be achieved by, e.g., (weighted) averaging or median selection.

Selection of the "most agreeable" policy: Instead of combining several policies to one, one could as well select the presumably best policy of the ensemble. That policy could be the "most agreeable" one, i.e., the one that is most often among the majority [18]. This could be useful for situations where there are two equally good ways to navigate the MDP. One policy could come up with one way, another policy with the other. Mixing them might produce an inferior policy.

Ensemble representation of Q-function: Instead of letting each instance run for itself, the ensemble can already be used to generate new targets in each iteration. To do so, after having trained all learners in an iteration their combined output is used to generate new targets for the next iteration. E.g., the single outputs can be combined as a (weighted) average. This is similar to the tree-based approach of [3].

Common ensemble policy: Similar to the ensemble representation of the Q-function, one can in each iteration generate a common policy from the ensemble. For determining the targets for the next iteration each learner uses its own current Q-function, but instead of maximizing it for

the successor state, the *Q*-value of the action selected by the ensemble policy is used. For obtaining the policy the same methods as for combining final policies can be used, e.g., (weighted) majority voting.

V. EXPERIMENTS

To demonstrate the usefulness of ensembles in combination with NFQ, we conducted experiments using the well-known pole balancing problem.

A. Pole Balancing

In the pole balancing benchmark a pole attached to a cart must be kept in upright position by applying forces to the cart. Starting from an upright position, in each time step the agent can choose to apply -50 N, 0 N, or $+50$ N to the cart. The actions are corrupted by uniformly distributed noise $n \in [-10, 10]$ N. Therefore, the trivial policy of always applying 0 N does not lead to success, even though initially the pole is in upright position. The two-dimensional state space consists of the pole's angle φ and the angular velocity $\dot{\varphi}$. A reward of 0 is given if $\varphi \in [-\frac{\pi}{2}, \frac{\pi}{2}]$. If the pole leaves this area a reward of -1 is given and the episode ends. The time constant used is $\Delta t = 0.1$ s, the discount factor is set to $\gamma = 0.95$. For details on the dynamics we refer to [4]. A policy is considered successful if it is able to repeatedly balance the pole for at least 3,000 steps.

B. Setup

To generate observations of the MDP, we used episodes of random exploration. When applying actions randomly, the pole falls (and therefore the episode ends) after approximately six steps. We used data sets of 25 (≈ 150), 50 (≈ 300), and 100 (≈ 600) episodes (observations/transitions). For each episode length we generated 50 data sets and used those to generate policies with the different methods. To assess a policy's quality, it was run 100 times for at most $3,000$ steps.

C. Network Topologies and Training Procedure

We used two different network topologies. The first is a standard 4-layer network consisting of an input layer, two hidden layers, and an output layer. The input layer contains five neurons, two for coding the state (angle and angular velocity) and three for binary coding the action (input for action 1: $(+1, -1, -1)$, input for action 2: $(-1, +1, -1)$, input for action 3: $(-1, -1, +1)$). Each hidden layer contains five neurons. The input and output layers use the identity as transfer function, the hidden layers use the hyperbolic tangent as transfer function.

The other topology is a deep, cascaded neural network that contains eight hidden layers with ten neurons each. In addition to a connection to the next layer each hidden layer is connected to the output as well (figure 4). This constrains upper layers to combine features of a lower

layer to compensate residuals of the lower layers as all layers contribute to the output. Although this topology does not enable NFQ to determine better policies on the pole balancing benchmark in general, we added it to increase the diversity of the ensemble.

The training algorithm used is VarioEta [19]. Every network training was performed in phases with decreasing learning rate. The training with each learning rate was stopped when the improvement on the validation error fell below a threshold. The available data was split randomly in 70% training and 30% validation data. See figure 3 for more details.

In addition to the two different network topologies we introduced variety into the ensemble by randomly initializing the weights of the networks (uniformly in $[-0.2, 0.2]$) and randomly splitting the data in a training and validation set for each network training, thus realizing some form of bagging [14].

- set $\eta = 0.1$
- learn_to_min(num_epochs = 30, $\varepsilon = 10^{-4}$)
- set $\eta = 0.01$
- learn_to_min(num_epochs = 30, $\varepsilon = 10^{-5}$)
- set $\eta = 0.001$
- learn_to_min(num_epochs = 30, $\varepsilon = 10^{-6}$)

Figure 3. The procedure for training a neural network. η denotes the learning rate of the VarioEta learning algorithm. learn_to_min() trains the network in blocks consisting of num_epochs epochs until the improvement of the validation error from one block to another drops below ε.

D. Results and Discussion

The results of the experiments are shown in three tables as number of successful policies and the average and standard deviation of the number of steps balanced. Table I shows the results using single networks. *4L* denotes the standard 4-layer network, *10LD* the deep, cascaded architecture. The other two tables show results of ensembles consisting of single network policies. For the results in table II majority voting was used (i.e., the action that most ensemble members would choose is selected as final action), for the results in table III *Q*-averaging was used (i.e., for each action the estimated *Q*-values from each ensemble member are averaged and the action maximizing this averaged *Q*-function is selected).

The performance of our networks when using 50 random episodes as training data approximately matches the performance reported by Riedmiller for his NFQ approach [5]. He does not give results for 25 episodes, our performance for 100 episodes is significantly worse (Riedmiller achieved 48/50 successful trials). With more optimization of the learning process it would probably be possible to further improve the results for 50 and 100 episodes. In particular, we suspect an adaption of the num_epochs parameter w.r.t.
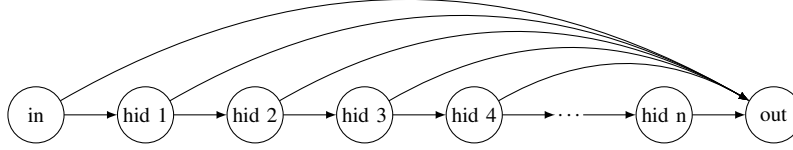
Figure 4. Deep, cascaded neural network where each layer is connected to the output layer. Each circle represents a layer of neurons, each arrow denotes a weight matrix realizing a full connection of the respective layers.

Table I

RATIO OF SUCCESSFUL POLICIES (FIRST LINE) AND AVERAGE (STANDARD DEVIATION) OF THE NUMBER OF STEPS BALANCED (SECOND LINE) USING SINGLE NETWORKS. 4L DENOTES THE STANDARD 4-LAYER NETWORK, 10LD THE 10-LAYER DEEP CASCADED NETWORK.

	number of episodes		
	25	50	100
1x 4L	24/50 (48%) 2139 (1208)	20/50 (40%) 2061 (1259)	37/50 (74%) 2852 (455)
1x 10LD	14/50 (40%) 1656 (1384)	22/50 (44%) 1837 (1294)	24/50 (48%) 2051 (1291)

Table II

RATIO OF SUCCESSFUL POLICIES (FIRST LINE) AND AVERAGE (STANDARD DEVIATION) OF THE NUMBER OF STEPS BALANCED (SECOND LINE) OF ENSEMBLE POLICIES DERIVED BY MAJORITY VOTING.

	number of episodes		
	25	50	100
5x 4L	28/50 (56%) 2410 (1062)	38/50 (76%) 2673 (866)	43/50 (86%) 2956 (200)
10x 4L	32/50 (64%) 2414 (1079)	37/50 (74%) 2689 (792)	45/50 (90%) 2995 (19)
15x 4L	33/50 (66%) 2503 (935)	38/50 (76%) 2718 (756)	46/50 (92%) 2990 (43)
20x 4L	34/50 (68%) 2589 (881)	37/50 (74%) 2691 (772)	48/50 (96%) 2992 (51)
5x 10LD	24/50 (48%) 2189 (1070)	26/50 (52%) 2357 (1082)	37/50 (74%) 2736 (711)
10x 10LD	30/50 (60%) 2599 (843)	35/50 (70%) 2748 (674)	45/50 (90%) 2868 (561)
15x 10LD	30/50 (60%) 2658 (811)	37/50 (74%) 2777 (661)	45/50 (90%) 2910 (433)
20x 10LD	32/50 (64%) 2694 (757)	40/50 (80%) 2793 (653)	48/50 (96%) 2966 (182)
5x 4L & 5x 10LD	33/50 (66%) 2655 (831)	36/50 (72%) 2795 (708)	47/50 (94%) 2998 (13)
10x 4L & 10x 10LD	37/50 (74%) 2673 (812)	43/50 (86%) 2915 (425)	50/50 (100%) 3000 (0)
15x 4L & 15x 10LD	36/50 (72%) 2700 (790)	43/50 (86%) 2895 (495)	50/50 (100%) 3000 (0)
20x 4L & 20x 10LD	39/50 (72%) 2709 (787)	45/50 (86%) 2904 (471)	50/50 (100%) 3000 (0)

to the number of training examples to be crucial (we used a fixed value of 30; first experiments using num_epochs = 15 for 50 episodes (not reported here) showed a significant improvement of single policy quality). However, when looking at the results of table II it becomes obvious that by combining different networks to ensembles it is possible to match (100 episodes) or even surpass (50 episodes) the performance of a fairly optimized standard NFQ approach.

Adding networks to the ensemble increases the performance to a certain point, which is not always reached here (adding even more networks than our maximum of 20 would be required). Among networks of the same type there seems to be already enough variety to benefit from an ensemble, but combining networks of different types is better—not only are the heterogeneous ensembles containing the most members (15x 4L & 15x 10LD and 20x 4L & 20x 10LD) better than all homogeneous ensembles, in 11/12 cases heterogeneous ensembles perform better than homogeneous ones of the same size. Comparing the aggregation techniques, majority voting is superior to Q-averaging. While for the ensembles of 4L networks both perform equivalently, for combination of 10LD networks and the heterogeneous ensembles majority voting is clearly better (8/12 and 12/12 cases, respectively). A reason for this might be that the different networks' Q-functions have different ranges. Another reason for majority voting being superior might lie in the fact that a single really bad Q-function can dominate the average (drastically decreasing or increasing it); with majority voting, the bad Q-function has only one vote, the magnitude of the Q-values plays no role.

VI. CONCLUSION

While RL with neural networks as function approximators has proven to be very powerful, it is still difficult to handle in practice. In this paper we proposed to use ensembles to make the learning process more robust and reliable and less dependent on fine-tuning of various parameters. We showed various ways of aggregating single learners to a common policy and demonstrated the potential of the approach. As it turns out, majority voting is superior to Q-averaging and using different network topologies is advisable.

Future work will include experiments with other RL problems and other aggregation schemes. Furthermore, it would be beneficial to have some quality measure for a single policy that could be used for a weighted majority voting. According to our experiments, the validation error is not sufficient to asses the quality of the resulting policy.

REFERENCES

[1] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[2] G. J. Gordon, "Stable function approximation in dynamic programming," in *Proc. of the Int. Conf. on Machine Learning*, 1995.

Table III
RATIO OF SUCCESSFUL POLICIES (FIRST LINE) AND AVERAGE (STANDARD DEVIATION)
OF THE NUMBER OF STEPS BALANCED (SECOND LINE) OF ENSEMBLE POLICIES DERIVED
BY Q-AVERAGING.

|	number of episodes		
	25	50	100
5x 4L	31/50 (62%) 2560 (878)	36/50 (72%) 2592 (932)	43/50 (86%) 2911 (356)
10x 4L	31/50 (62%) 2647 (708)	32/50 (64%) 2643 (793)	48/50 (96%) 2996 (20)
15x 4L	36/50 (72%) 2606 (790)	31/50 (62%) 2675 (711)	49/50 (98%) 2999 (2)
20x 4L	34/50 (68%) 2616 (812)	35/50 (70%) 2754 (623)	50/50 (100%) 3000 (0)
5x 10LD	24/50 (48%) 2250 (1172)	37/50 (74%) 2381 (1072)	36/50 (72%) 2626 (884)
10x 10LD	31/50 (62%) 2474 (1004)	33/50 (66%) 2579 (985)	40/50 (80%) 2652 (887)
15x 10LD	29/50 (58%) 2497 (1002)	37/50 (74%) 2500 (1086)	41/50 (82%) 2671 (835)
20x 10LD	30/50 (66%) 2520 (991)	37/50 (78%) 2562 (985)	42/50 (84%) 2734 (814)
5x 4L & 5x 10LD	30/50 (60%) 2673 (792)	31/50 (62%) 2647 (806)	43/50 (86%) 2898 (463)
10x 4L & 10x 10LD	31/50 (66%) 2640 (806)	39/50 (78%) 2654 (903)	43/50 (86%) 2882 (483)
15x 4L & 15x 10LD	31/50 (66%) 2586 (920)	40/50 (80%) 2592 (965)	44/50 (88%) 2816 (667)
20x 4L & 20x 10LD	33/50 (66%) 2619 (877)	40/50 (80%) 2685 (804)	44/50 (88%) 2819 (651)
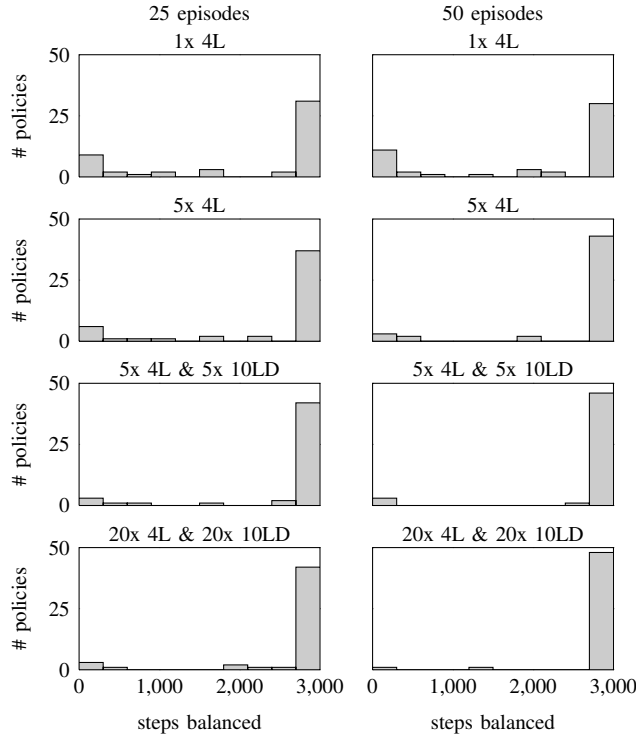
Figure 5. Histograms showing the distributions of policy quality for single networks (top row) and various ensembles.

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

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

[5] M. Riedmiller, "Neural fitted Q-iteration – first experiences with a data efficient neural reinforcement learning method," in *Proc. of the 16th European Conf. on Machine Learning*, 2005, pp. 317–328.

[6] T. Dietterich, "Ensemble methods in machine learning," *Multiple classifier systems*, pp. 1–15, 2000.

[7] M. Wiering and H. van Hasselt, "Ensemble algorithms in reinforcement learning." *IEEE transactions on systems, man, and cybernetics*, vol. 38, no. 4, 2008.

[8] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[9] D. Schneegass, S. Udluft, and T. Martinetz, "Neural rewards regression for near-optimal policy identification in Markovian and partial observable environments," in *Proc. of the European Symposium on Artificial Neural Networks*, 2007.

[10] G. J. Gordon, "Reinforcement learning with function approximation converges to a region," *Advances in neural information processing systems*, pp. 1040–1046, 2001.

[11] T. Gabel and M. Riedmiller, "Reducing policy degradation in neuro-dynamic programming," *Proc. of the European Symposium on Artificial Neural Networks*, 2006.

[12] S. Thrun and A. Schwartz, "Issues in using function approximation for reinforcement learning," in *Proc. of the 1993 Connectionist Models Summer School, Hillsdale, NJ*, 1993.

[13] C. Gaskett, "Q-learning for robot control," Ph.D. dissertation, The Australian National University, 2002.

[14] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[15] Y. Freund, R. Schapire, and N. Abe, "A short introduction to boosting," *Journal of the Japanese Society for Artificial Intelligence*, vol. 14, pp. 771–780, 1999.

[16] H.-G. Zimmermann, R. Grothmann, A. M. Schaefer, and C. Tietz, "Modeling large dynamical systems with dynamical consistent neural networks," in *New Directions in Statistical Signal Processing: From Systems to Brain*, S. Haykin, J. Principe, T. Sejnowski, and J. McWhirter, Eds. MIT Press, 2006, pp. 203–242.

[17] A. M. Schaefer, S. Udluft, and H.-G. Zimmermann, "A recurrent control neural network for data efficient reinforcement learning," in *Proc. of the IEEE Int. Symposium on Approximate Dynamic Programming and Reinforcement Learning*, Honolulu, HI, 2007.

[18] H. van Hasselt, personal communication, 2010.

[19] R. Neuneier and H.-G. Zimmermann, "How to train neural networks," in *Neural Networks: Tricks of the Trade*, G. B. Orr and K.-R. Müller, Eds., 1996, pp. 373–423.