Learning Heuristic Functions for Classical Planning by Deep Reinforcement Learning

Anonymous Authors
Anonymous Institute
anonymous@anonymous

Abstract

Given the success of neural networks (NN) in learning powerful search guidance for complex games, their application in AI Planning is highly promising too. Here we focus on learning heuristic functions for classical planning from scratch using only states as the NN input. This has been addressed in prior work by (a) supervised learning which limits training to instances small enough for training data generation to be feasible, and/or (b) generalization over instance size which limits the approach to domains and instance distributions where search knowledge useful for large instances can be generalized from other ones. We investigate the use of reinforcement learning on large instances, generalizing only over initial states but avoiding both (a) and (b). We explore two methods, based on bootstrapping and approximate value iteration. Our experiments show that these methods can learn NN heuristic functions surpassing both learning and non-learning state-of-the-art approaches: we outperform recent NN heuristic function learning methods in several domains, and we outperform LAMA in one domain.

1 Introduction

Neural networks (NN) have proved to be able to learn powerful search guidance for several complex games. Successes include the AlphaGo/Zero system series [Silver *et al.*, 2016; Silver *et al.*, 2018] which excels in Go, Chess, and Shogi, as well as recent work [Agostinelli *et al.*, 2019] that successfully tackles single-agent puzzles including Rubik's Cube. Given that game-state evaluators correspond to heuristic functions, and given the prominence of heuristic search in AI Planning [e.g., Hoffmann and Nebel, 2001; Helmert and Domshlak, 2009; Richter and Westphal, 2010; Helmert *et al.*, 2014; Domshlak *et al.*, 2015], the application of NN to learn heuristic functions clearly is highly promising there too. Indeed, several works have already appeared addressing this problem with different methods and from different angles [Toyer *et al.*, 2018; Garg *et al.*, 2019; Ferber *et al.*, 2020; Shen *et al.*, 2020; Rivlin *et al.*, 2020; Yu *et al.*, 2020].

Here we focus on learning heuristic functions for classical planning from scratch using only states as the NN input. This has been addressed in prior work by (a) supervised learning and/or (b) generalization over instance size, either per-domain or even across domains. Specifically, Ferber *et al.* [2020] use supervised learning with a feed-forward NN structure that allows to generalize (only) over the initial state. Shen *et al.* [2020] design the *HGN* NN architecture that allows *per-domain learning*, generalizing across instances within a domain (and even across domains); they use supervised learning on small instances. Rivlin *et al.* [2020] design a per-domain learning NN architecture based on graph convolution, using reinforcement learning (RL) on small instances; related approaches have been devised for probabilistic planning [Garg *et al.*, 2019]. Ferber *et al.* [2020]'s approach can yield heuristic functions competitive with the state of the art, but is limited to instances small enough for training data generation – solving many sample states with an off-the-shelf planner as the teacher – to be feasible. Per-domain learning solves the teacher-scalability problem as one can train the NN on small instances. Yet, arguably, this can be successful only on domains and instance distributions where search knowledge useful for large instances can be learned on small ones.

We investigate the use of reinforcement learning (RL) on large instances, avoiding above drawbacks; we generalize only over initial states as in Ferber *et al.*'s [2020] work.[1] Adopting previous ideas to the planning context, we explore two distinct methods. First, *bootstrapping* as introduced by Arfaee *et al.* [2011a]. This iteratively trains a NN heuristic function by running it on increasingly difficult training states, generated by increasingly long backward walks from the goal. Whereas Arfaee *et al.* used pre-existing heuristic functions as input features (also in classical planning [Arfaee *et al.*, 2011b]), we investigate learning the heuristic function h^{Boot} from scratch using just the planning state as NN input. Furthermore, we introduce a variant h^{BootExp} where what is learned is not a goal-distance estimation but an estimation of the number of states that greedy best-first search needs to expand when using the heuristic. Second, we investigate what we call *approximate value iteration*, inspired by the work of

[1]Preliminary work along similar lines has appeared in an ICAPS'20 workshop [Yu *et al.*, 2020], but focusing on very small NN as well as per-instance training within IPC time limits.

Agostinelli *et al.* [2019] on Rubik's Cube, where the NN heuristic function h^{AVI} is also trained on states s sampled backwards from the goal. However, instead of a full search we do a k-step look-ahead. We evaluate h^{AVI} on the fringe states of that look-ahead, then perform Bellman updates backwards to s and use the final updated value for training on s.

Our experiments show that these methods can learn heuristic functions surpassing those learned by supervised learning as per Ferber *et al.* [2020], and able to tackle larger instances without excessive effort for training data generation. While the state-of-the-art model-based planner LAMA [Richter and Westphal, 2010] is typically still superior to all learning approaches, in one benchmark domain, namely Storage, our new NN heuristics yield vastly better performance. Finally, we compare our method to the per-domain learning approach of Shen *et al.* [2020]. Such a comparison must of course be treated with care, as per-domain learning is very different from per-instance learning. Nevertheless, we believe that comparisons across different NN learning approaches are valuable, and are ultimately required to further the empirical field of NN learning in planning. Specifically in the present work, we believe there is value in evaluating the hypothesis that per-instance RL can yield better heuristic functions than heuristics that have to generalize over a whole domain. Our experiments indeed confirm that this is so, in many domains.

2 Preliminaries

We use the *FDR* planning framework [Bäckström and Nebel, 1995]. A planning task is a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, s_{\mathcal{I}}, \mathcal{G} \rangle$. $\mathcal{V}$ is a set of *variables*, $\mathcal{A}$ is a set of *actions*, the *initial state* $s_{\mathcal{I}}$ is a complete variable assignment, and the *goal* $\mathcal{G}$ is a partial variable assignment. Each *action* $a \in \mathcal{A}$ defines a *precondition* pre_a and an *effect* eff_a, both partial variable assignments. For simplicity we consider unit action costs (all costs are 1).

A *fact* is a variable-value pair $\langle v, d \rangle$ where $v \in \mathcal{V}$ and $d \in \mathcal{D}_v$. An action a is applicable in a state s if pre_a is satisfied in s. Applying a in s leads to a state s' with the same variable assignment as in s except for those variable assignments defined in eff_a. A *plan* is a sequence of actions $\pi = \langle a_1, \dots, a_n \rangle$, such that sequentially applying each action in π from $s_{\mathcal{I}}$ leads to a state that satisfies $\mathcal{G}$. Our objective is to find some plan for Π.

We need a few well-known concepts that we will employ later on. First, the *regression* of a partial variable assignment G over an action a is defined as $(G \setminus eff_a) \cup pre_a$ if $eff_a \cap G \neq \emptyset$ and G is consistent with eff_a, i.e. $eff_a \subseteq G$; otherwise, the regression of G over a is undefined. This operation underlies backward search (in our case: random walks). Second, *Bellman updates* are a well-known method to iteratively improve state-value estimates [e.g., Bertsekas and Tsitsiklis, 1996]. In unit-cost classical planning, the Bellman equation simplifies to $h^*(s) = \min_a[1 + h^*(s')]$ where h^* denotes the exact goal distance. For an approximation h of h^* a Bellman update takes the form $h(s) := \min_a[1 + h(s')]$.

3 Bootstrapping

In the following we explore different RL methods. This section is concerned with two variants based on *bootstrapping*, while the next section introduces a method based on *approximate value iteration*. Before we discuss bootstrapping in detail, we first describe the general neural network architecture and the basic hyperparameters underlying both methods.

3.1 Common Hyperparameters

We use RL to train a heuristic function for a given FDR task Π, to be used in greedy best-first search (GBFS). As we have unit action costs, our learned heuristic functions are goal-distance estimators (rather than remaining-cost estimators).

Our NN are residual networks[2] [He *et al.*, 2016] with two dense layers followed by one residual block containing two dense layers and a single output neuron. Each dense layer contains 250 neurons. All neurons use the *ReLU* activation function. The inputs of our NN are states represented as fixed-size Boolean vectors. We associate every entry of the input vector with a fact of Π. We set the vector entry to 1 if the corresponding fact is in the input state, otherwise it is 0. We train the NN using the *mean squared error* as loss function and the *adam* optimizer with its default parameters [Kingma and Ba, 2015]. To prevent performance instabilities during training, we update the model for the sample generation after at least 50 epochs have passed and the mean squared error is below 0.1.

Because generating a training batch of 250 samples takes longer than training on that batch, we use experience replay. The data generation process pushes all samples into a *first-in-first-out* buffer with a maximum size of 25,000. In each training epoch we uniformly choose 250 samples from the buffer. This allows us to train multiple times on the same (recent) samples. It also decouples training from training data generation so that the two processes can run largely in parallel.

We run the training – including data generation – for 28 hours on 4 cores of an Intel Xeon E5-2600 processor with 4 GB of memory. For training the neural networks, we used the Keras framework [Chollet, 2015] with Tensorflow [Abadi *et al.*, 2015] as back-end. We implemented the data generation and all searches in Fast Downward [Helmert, 2006], and used Lab [Seipp *et al.*, 2017] to setup our experiments.

3.2 Bootstrapping a Goal-Distance Estimator

We follow the approach to bootstrapping as introduced by Arfaee *et al.* [2011a]. The idea is to train the heuristic function h by running it on increasingly difficult training states. The training states are obtained by increasingly long backward walks from the goal, and the training data on each state is obtained by running a heuristic search with the current h. Thus h is iteratively improved from the more accurate goal-distance estimates obtained using search. We make the following adaptations to obtain our heuristic function h^{Boot}:

- While Arfaee *et al.* [2011a] also used a neural network, their network was quite small (single layer) and used pre-existing (model-based) heuristic functions as input features.

[2]Residual networks have been successfully applied to image classification, and have previously been used on Rubik's Cube [Agostinelli *et al.*, 2019].

Instead, we use the FDR state as input, and evaluate the extent to which a more complex NN architecture can cope with this basic representation (discovering the relevant features itself).

- We need to define what "backward walks" mean in our context; Arfaee *et al.* [2011a] focused on domains where the goal state is fully specified, while in planning typically the goal is a partial state only. To address this, we use regression as specified in the previous section. We start at the goal of Π, and perform a random walk for n regression steps, where n is uniformly chosen from $\{0 \leq n \leq \textit{walk\_length}\}$ with *walk_length* being a (growing, see below) parameter. The regression walk ends with a partial assignment. We randomly complete the partial assignment to a state, assigning each unassigned variable a random value.[3] We use Fast Downward's translator [Helmert, 2009] to identify (some) mutex value pairs. To enforce that no mutex pair is violated, we iterate over the unassigned variables in random order and assign only values that do not violate mutexes to variables already assigned.

Our other changes amount to parameter tuning. Like Arfaee *et al.* [2011a], we use GBFS with the current learned heuristic h^{Boot} to obtain training data on the training states. We use a timeout of 10 seconds in this search. If the search succeeds, we store all states s_i along the plan for training, the associated goal-distance estimates being taken from the plan (number of actions starting from s_i). We observed that in the beginning h^{Boot} is typically not good enough to solve sampled states far away from the goal. Therefore, the sampling starts with a walk length between 0 and 5. We double the maximum random walk length whenever GBFS finds a plan for more than 95% of the training states. At some point, further increasing the random walk length typically does not sample states further away from the goal, so we double the maximum walk length at most 8 times.

3.3 Bootstrapping a Search-Space-Size Estimator

In addition to h^{Boot}, we furthermore designed a variant h^{BootExp}, where what is learned is not a goal-distance estimation, but an estimation of the number of states that GBFS needs to expand when using the learned heuristic.

This approach is motivated by the observation that in GBFS we do not want to select the state with the smallest heuristic estimate as successor, but the state which minimizes the number of states to expand (search space size). Often, the heuristic estimate correlates with the search space size, but in a lose way given the highly volatile behavior of search as a function of the node ordering. For example, two states could be evaluated to the same heuristic value, but one state leads to a plateau in the heuristic function and causes the search to expand drastically more states than the other. Estimating instead the size of the search space provides a more direct link between the learning objective and search behavior. We can directly use the search space estimation as a node ordering in GBFS, because nodes with smaller estimates are naturally preferable. In particular, with unit action costs, $h^*(s)$ for a state s is exactly the number of nodes expanded by GBFS on s. Hence, it is possible for RL to converge to a function with 0 error (h^* is one such function; others may exist).

Our training procedure for h^{BootExp} uses the same backward walks to generate training states s, as before. We then run GBFS using the current h^{BootExp} and a time limit of 10 seconds, also as before. If the search succeeds, we train on s and the number of expanded states. If the search does not succeed, we do the same thing with the number of states expanded up to the time limit. Our motivation for the latter is purely empirical – this turned out to be better than when using only the successful searches. Intuitively, training on unsuccessful states s provides a useful training signal regarding which states are difficult to handle.

4 Approximate Value Iteration

Another way of viewing RL for heuristic functions is as a form of approximate value iteration. Exact value iteration consists in applying Bellman updates to a tabular value function h mapping every state s to a remaining-cost estimate and, if every state s is updated infinitely often, it converges to h^* regardless the update ordering [Bertsekas and Tsitsiklis, 1996]. The idea in approximate value iteration simply is to replace the value table with a function representing an approximate value function h to be used as a heuristic for search. This has been successfully done in single-agent puzzles including Rubik's Cube [Agostinelli *et al.*, 2019]. We adopt this to classical planning here, designing our third heuristic function h^{AVI}.

Our neural network architecture and hyperparameters remain as described in Section 3.1. The generation of sample states for training is done exactly as for bootstrapping, except that we keep the maximum walk length fixed because an increasing walk length is not necessary here. What we change is the generation of training data for a sample state s. Instead of running GBFS starting at s, we merely do a k-step look-ahead, where k is a parameter and $k = 2$ is used in our experiments. We construct the k-step search tree rooted in s. We evaluate the fringe states of that tree as 0 if they are goal states and otherwise with h^{AVI}. Then we perform Bellman updates backwards, updating in each step the value of some intermediate state t with those of its children in the tree. Upon termination, the updated value at s is used for training.

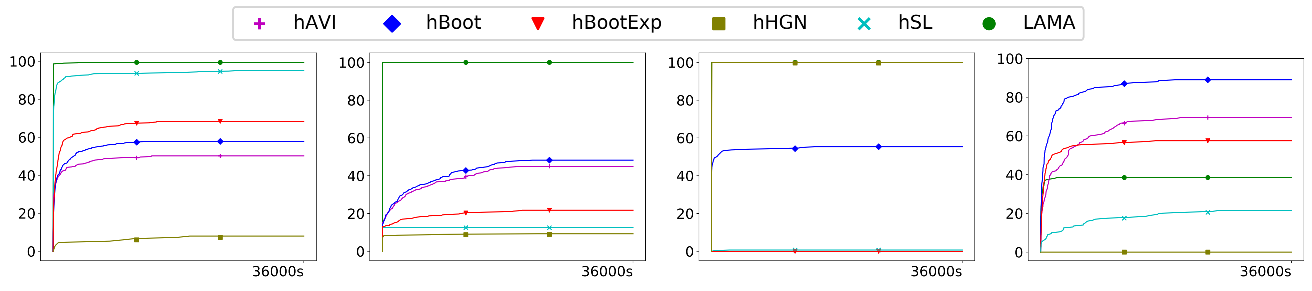
5 Performance

We implemented our NN heuristics on top of FD, starting from Ferber et al.'s [2020] code base. We evaluate the heuristics in greedy best-first search (GBFS). Each search is run with 4 GB of memory. We run the search on a single core, as the LAMA planner we compare to cannot exploit multiple cores. However, we do design our experiments to account for the facts that 1) NN heuristic functions are extremely slow in state evaluation compared to model-based heuristics such as used in LAMA, which puts them at an intrinsic disadvantage; while 2) at the same time, it is well

[3]Arfaee *et al.* [2011b] suggest this as an alternative to the per-domain training in small instances they use. They suggest to randomly complete the goal though (rather than the final regressed subgoal of the walk), which in our experiments did not work as well.

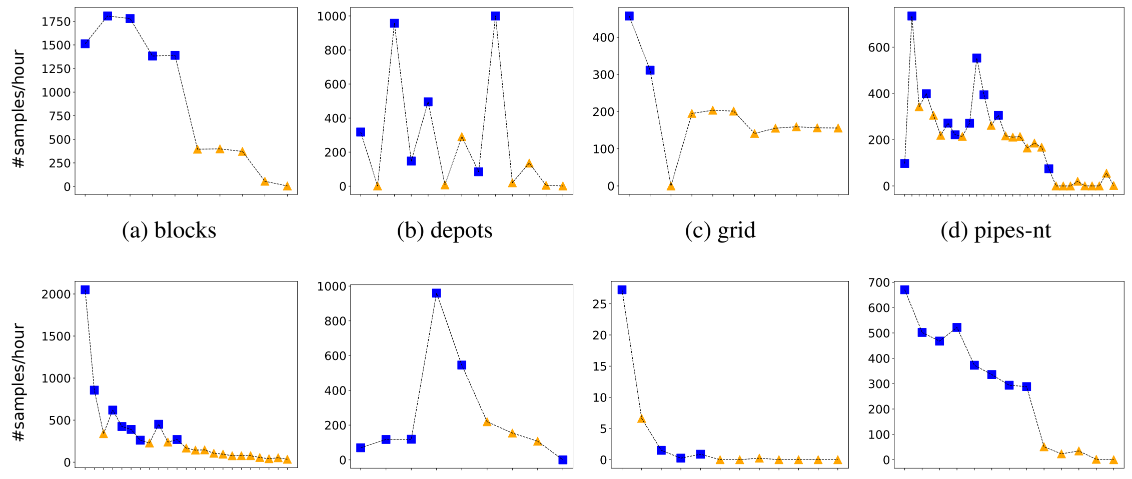
	Moderate Tasks										Hard Tasks				
	No Validation			Validation						Validation				
Domain	h^{Boot}	h^{BootExp}	h^{AVI}	h^{Boot}	h^{BootExp}	h^{AVI}	h^{HGN}	h^{SL}	Lama	h^{Boot}	h^{BootExp}	h^{AVI}	h^{HGN}	Lama
blocks	0.0	0.0	0.0	18.0	0.0	0.0	94.3	100.0	100.0	0.0	0.0	0.0	34.5	96.8
depots	31.7	17.7	43.7	60.3	32.7	54.7	0.0	83.7	100.0	8.3	4.3	12.9	0.0	82.6
grid	100.0	100.0	51.0	100.0	100.0	51.0	0.0	76.0	100.0	87.8	95.0	70.5	0.0	100.0
npuzzle	27.0	0.0	1.0	28.0	0.0	1.0	0.7	0.0	100.0	0.0	0.0	0.0	0.0	86.5
pipes-nt	36.2	51.2	21.4	57.8	68.4	50.2	10.0	95.2	99.4	23.4	19.1	8.0	0.0	69.3
rovers	36.5	15.2	34.2	48.2	21.8	45.0	12.2	12.5	100.0	2.8	0.8	6.5	0.0	100.0
scanalyzer	33.3	59.7	66.7	33.3	70.7	67.3	18.8	83.3	100.0	3.3	0.0	60.7	0.0	100.0
storage	89.0	61.0	67.0	89.0	57.5	69.5	0.0	21.5	38.5	27.2	13.2	15.8	0.0	11.5
transport	83.8	79.5	70.0	100.0	100.0	87.5	94.5	100.0	100.0	0.0	0.0	2.4	0.0	92.8
visitall	17.0	0.0	0.0	55.3	0.0	0.0	100.0	0.7	100.0	28.0	0.0	0.0	100.0	100.0
average	45.4	38.4	35.5	59.0	45.1	42.6	45.1	57.3	93.8	15.6	13.2	17.0	13.5	84.0

Table 1: Coverage (in %) of the reinforcement learning methods and the baselines for the different domains. The left-hand side shows the coverage on the moderate task set, the right-hand side shows the coverage on the hard task set. In the setting without validation (*No Validation*) we train every model exactly once. In the setting with validation (*Validation*), we retrain models depending on its performance on the validation tasks. For the supervised baseline (h^{SL}) we use the models published by Ferber *et al.* [2020].



(a) pipes-nt (b) rovers (c) visitall (d) storage

Figure 1: Coverage (%) as a function of the search time limit, on all example moderate task for four domains.



(a) blocks (b) depots (c) grid (d) pipes-nt

(e) rovers (f) scanalyzer (g) storage (h) transport

Figure 2: Every plot shows for the tasks of one domain how many training samples can be generated per hour for the supervised learning approach. The tasks are in their IPC order. Tasks with a blue square belong to the moderate tasks, tasks with an orange triangle belong to the hard tasks. The domains NPuzzle and VisitAll are skipped, because the supervised learning approach solves almost no tasks in those domains.

known that NN evaluation can be sped up dramatically using GPUs or TPUs [Silver *et al.*, 2016; Silver *et al.*, 2018; Agostinelli *et al.*, 2019], which arguably counter-acts this disadvantage in practice. Also, Ferber et al.'s previous work already showed that using multiple cores speeds up state evaluation. To account for this while sticking to a fair experimental setting where all competitors have the exact same resources, we set a generous search-time limit of 10 hours, allowing the NN heuristics to exhibit strengths in informedness.

We evaluate our models on the same domains as Ferber *et al.* [2020]: Blocksworld, Depots, Grid, NPuzzle, Pipesworld-NoTankage, Rovers, Scanalyzer, Storage, Transport, and VisitAll. For each domain, Ferber *et al.* [2020] selected tasks difficult enough to be interesting and easy enough to generate training data. We will refer to these tasks as *moderate*, in contrast to the larger *hard tasks* we also consider. The latter is possible here as our RL techniques get rid of the teacher-scalability problem. The hard tasks are simply the standard benchmark instances of size beyond the moderate tasks. In Blocksworld, there are no such instances so we generated new larger tasks with more blocks. Similarly, in Grid there was only a single non-moderate benchmark instance so we designed new hard instances by extending the standard ones.

Every task describes a state space and a goal. For every state space we evaluate the models on 50 distinct initial states. For the moderate task set, we use initial states published by Ferber *et al.* [2020]. For the hard tasks, we use their method to create new initial states. That is, we start a random forward walk of 200 steps from the original initial state and store the final state as test state.

We compare our goal-distance estimators based on bootstrapping (h^{Boot}), search-space-size estimators based on bootstrapping (h^{BootExp}), and goal-distance estimators based on approximate value iteration (h^{AVI}) against the supervised learning approach (h^{SL}) of Ferber *et al.* [2020], STRIPS-HGN (h^{HGN}) by Shen *et al.* [2020], and LAMA [Richter and Westphal, 2010]. All our code is online available[4]. We next discuss a simple solution to brittle learning performance, then how we adapted STRIPS-HGN for fair comparison in our experiments. We then turn to the performance comparison across the different methods.

5.1 Boosting NN Heuristics through Validation

Consider the results for moderate tasks shown in Table 1 under the headline *No Validation*. We observed that the performance was often very brittle. The heuristics solve either all testing initial states for one benchmark instance or none at all, with the picture changing radically after re-training. Performance is thus drastically affected by the randomness during training (e.g. parameter initialization, random walks in training data generation). As a simple remedy, we introduce a validation method. For each state space (benchmark instance), we generate 10 new initial states for validation. After training a NN, we run heuristic search using it on the validation states, with a search-time limit of 30 minutes. If less than 80% of the validation states are solved, we retrain. We retrain at most three times, and the last trained NN is used. This results in more robust performance based on the assumption that good performance on the validation states correlates with good performance on the test states. Table 1 shows under *Validation* that this is indeed the case: the validation method improves performance significantly. The coverage of all reinforcement learning approaches increases on almost all domains.

[4]URL to be announced

5.2 Adapting STRIPS-HGN

In order to compare against STRIPS-HGN, we adapted its training procedure to account for the extra training time and the source of training data used by the other learning approaches. More precisely, for each domain, we trained 10 different STRIPS-HGN networks simultaneously for up to 28 hours using 4 cores and 4 GB per core. The training time is split between data generation (10 hours) and training (1.8 hour per network). A single round of validation is then used for selecting the best STRIPS-HGN network per domain for testing against the other approaches.

The training data for STRIPS-HGN was generated by sampling with replacement a problem from the moderate and hard set of problems, performing a random walk from its goal with length n (as described in Section 3.2), and solving the generated task using A* instead of GBFS (as in the original STRIPS-HGN). This procedure is repeated until data generation time-out is reached. Tasks solved in less than 5 minutes are discarded and referred as easy tasks. Tasks not solved in less than 30 minutes are also discarded and referred as timed-out tasks. The optimal plans for all tasks that are not easy and not timed-out are used to generate the training set of state-value pairs. The random walk length n used for problem generation is uniformly chosen from $\{\underline{n} \leq n \leq \overline{n}\}$ where $\underline{n}$ and $\overline{n}$ are initially 50 and 500, respectively. When a task generated using n is an easy task, the lower bound $\underline{n}$ is updated to $(\underline{n} + 3n)/4$; conversely, $\overline{n}$ is updated to $(\overline{n} + n)/2$ when a timed-out task is found. The number of state-value pairs obtained following this procedure ranges from 78 for Transport to 1563 for VisitAll.

For Blocksworld, Scanalyzer and Transport, the data generation and training time was reduced to 2 hours and 10 minutes per network, respectively. This was done after we observed that some of the STRIPS-HGN networks were overfitting when using the full 28 hours.

5.3 Overall Performance Comparison

We now consider the comparison of our reinforcement learning approaches against each other, and against the competing approaches. For the supervised learning heuristic functions h^{SL}, we use the NN trained and published by Ferber *et al.* [2020]. For this reason, data for h^{SL} is included only for the moderate tasks run by Ferber *et al.* [2020].

Comparing average coverage on the moderate tasks shows that h^{Boot} and h^{SL} perform similarly well, with a coverage of approximately 60%. Performance varies a lot per domain, and our best new reinforcement learning heuristics h^{Boot} surpasses h^{SL} on 5 domains (Grid, NPuzzle, Rovers, Storage, VisitAll). This comparison needs to be offset against the fact that the training time for our reinforcement learning heuristics is only 28 hours, whereas h^{SL} is trained for up to 48 hours and makes use of training data generated using up to 400 hours.

The comparison to h^{HGN} indicates that, while our approach achieves higher coverage overall, there are complementary strengths between the two heuristics. h^{HGN} shows very strong performance in Blocksworld and VisitAll where all other learning approaches either fail to derive a meaningful heuristic or to scale up. At the same time, h^{HGN} solves a small number of tasks or fails to scale up in several domains where the other learning methods excel (e.g., Storage and Grid). The reason often is the hypergraph size underlying STRIPS-HGN, which scales with task size. The hypergraph for many tasks in Depots, Storage and Grid does not fit into memory and. For other domains, evaluating the h^{HGN} heuristic takes too much time due to the large hypergraphs. The strong performance on Blocksworld and VisitAll is, at least in part, due to the relatively small size of the generated hypergraphs: less than 1000 nodes and 1500 hyperedges.

Similarly to Ferber *et al.* [2020], the comparison to LAMA is not favorable. LAMA outperforms the learned heuristics on all domains except for one. In that domain, Storage, our new RL heuristics significantly outperforms LAMA though.

On the hard tasks, we see a drop in coverage for all techniques. LAMA still outperforms the learned heuristics in most domains. In Grid our new RL heuristics perform comparably, and in Storage they excel. As Ferber *et al.* [2020] concluded and as we will reconfirm below, h^{SL} is not suited to address the hard tasks due to the excessive training-data generation effort that would be required.

Given our comparatively huge search time limit of 10 hours, it is relevant to shed light on the influence of that time limit. Figure 1 does so by showing coverage as a function of runtime for moderate tasks in four selected domains. The data are qualitatively similar in other domains and tasks, so are essentially representative in terms of the main observations. Comparing different NN heuristics, the general finding is that coverage superiority does not change as a function of the time limit. With few exceptions, an approach that is better after 30 minutes is still better after 2 or more hours.

As expected the picture with respect to LAMA is very different, as LAMA (like all state-of-the-art model-based heuristic search planners) tends to solve a task either quickly or not at all. With its comparatively fast heuristic functions, LAMA quickly runs up against the memory limit. The NN heuristic functions in contrast are very slow (run on a single core!) as discussed, and thus require some time to “catch up” with LAMA. Their coverage still tends to increase even after a long run-time, and their advantages over LAMA – where present – become apparent for larger time limits.

5.4 Training-Data Generation Effort for Supervised Learning on the Hard Tasks

In some domains, in particular Blocksworld, Pipesworld-NoTankage, and Transport, supervised learning is clearly superior to reinforcement learning on the moderate tasks. This raises the question how far the method could actually scale to the hard tasks not attempted by Ferber *et al.* [2020].

To evaluate this question, we generated training data on the hard tasks for 8 hours (instead of 400, which would result in an enormous overall run-time on these tasks). Figure 2 shows for each domain how many samples are generated per hour as a function of instance size. We skipped NPuzzle and VisitAll because supervised learning does not even solve the moderate tasks in those domains. Blue squares in the plots represent the moderate task set, orange triangles represent the hard task set. Clearly, in most domains it is extremely difficult to generate training data for hard tasks. For many hard tasks in Depots, Pipesworld-NoTankage, Storage, Transport, hardly any samples are generated per hour.

Domain	Min	Average	Max
blocks	315	514	21K
depots	8	37	2.5K
grid	15	18	21
pipes-nt	3	∞	∞
scanalyzer	49	67	100
storage	1.4K	∞	∞
transport	75	173	30.4K

Table 2: Estimated time in hours to sample a sufficiently large training set for the hard tasks; min/average/max taken over the different instances in each domain.

Table 2 shows estimates how long it would take to generate useful training data for the hard tasks. This is calculated based on data provided by Ferber *et al.* [2020]: for each domain we choose N so that, according to Ferber *et al.* [2020], N training examples suffice to obtain coverage $\leq 30\%$ worse than the best coverage obtained. We then divide N by the training rate as per Figure 2. Note that this calculation optimistically assumes that the hard tasks do not require more training examples. Nevertheless, as Table 2 shows, training-data generation is infeasible for hard tasks in many domains.

Overall, our data confirms the conclusion of Ferber *et al.* [2020] that, typically, per-instance supervised learning will run up against a training-data generation barrier.

6 Conclusion

Our work has shown that reinforcement learning ideas can be arranged to achieve competitive performance for per-instance learning in classical planning. They remove the teacher-scalability barrier of supervised per-instance learning, and they typically perform better than the recent HGN approach trained per-domain. While LAMA remains typically superior, it is outperformed in one benchmark.

The major open question in our view remains whether and how more reliable performance can be obtained, with per-instance NN learning, and with NN learning in planning in general. As of now, performance varies wildly across domains. One may argue that this phenomenon pertains to virtually all planning techniques, but our impression is that this is more drastic for neural networks than for classical model-based heuristic functions. Coverage rises high vs. drops to rock-bottom for the same method even in intuitively closely related domains, and as we have outlined the same kind of variance occurs even in multiple training runs on the same planning task. How can we improve this erratic behavior? The authors speculate that other learning tasks, closer to the requirements of search than numeric heuristic values, may help; but as of now this question remains wide open.

References

[Abadi *et al.*, 2015] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fern, a Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[Agostinelli *et al.*, 2019] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1:356–363, 2019.

[Arfaee *et al.*, 2011a] Shahab J. Arfaee, Sandra Zilles, and Robert C. Holte. Learning heuristic functions for large state spaces. *AIJ*, 175:2075–2098, 2011.

[Arfaee *et al.*, 2011b] Shahab Jabbari Arfaee, Robert C. Holte, and Sandra Zilles. Bootstrap planner: an iterative approach to learn heuristic functions for planning problems. In *IPC-7 planner abstracts*, 2011.

[Bäckström and Nebel, 1995] Christer Bäckström and Bernhard Nebel. Complexity results for SAS$^+$ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[Bertsekas and Tsitsiklis, 1996] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[Chollet, 2015] François Chollet. Keras. https://keras.io, 2015.

[Domshlak *et al.*, 2015] Carmel Domshlak, Jörg Hoffmann, and Michael Katz. Red-black planning: A new systematic approach to partial delete relaxation. *AIJ*, 221:73–114, 2015.

[Ferber *et al.*, 2020] Patrick Ferber, Malte Helmert, and Jörg Hoffmann. Neural network heuristics for classical planning: A study of hyperparameter space. In *Proc. ECAI 2020*, pages 2346–2353, 2020.

[Garg *et al.*, 2019] Sankalp Garg, Aniket Bajpai, and Mausam. Size independent neural transfer for RDDL planning. In *Proc. ICAPS 2019*, pages 631–636, 2019.

[He *et al.*, 2016] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.

[Helmert and Domshlak, 2009] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. ICAPS 2009*, pages 162–169, 2009.

[Helmert *et al.*, 2014] Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *JACM*, 61(3):16:1–63, 2014.

[Helmert, 2006] Malte Helmert. The Fast Downward planning system. *JAIR*, 26:191–246, 2006.

[Helmert, 2009] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *AIJ*, 173:503–535, 2009.

[Hoffmann and Nebel, 2001] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

[Kingma and Ba, 2015] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proc. ICLR 2015*, 2015.

[Richter and Westphal, 2010] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR*, 39:127–177, 2010.

[Rivlin *et al.*, 2020] Or Rivlin, Tamir Hazan, and Erez Karpas. Generalized planning with deep reinforcement learning. In *ICAPS Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL)*, pages 16–24, 2020.

[Seipp *et al.*, 2017] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. https://doi.org/10.5281/zenodo.790461, 2017.

[Shen *et al.*, 2020] William Shen, Felipe Trevizan, and Sylvie Thiébaux. Learning domain-independent planning heuristics with hypergraph networks. In *Proc. ICAPS 2020*, pages 574–584, 2020.

[Silver *et al.*, 2016] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[Silver *et al.*, 2018] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[Toyer *et al.*, 2018] Sam Toyer, Felipe Trevizan, Sylvie Thiébaux, and Lexing Xie. Action schema networks: Generalised policies with deep learning. In *Proc. AAAI 2018*, pages 6294–6301, 2018.

[Yu *et al.*, 2020] Liu Yu, Ryo Kuroiwa, and Alex Fukunaga. Learning search-space specific heuristics using neural network. In *ICAPS Workshop on Heuristics and Search for Domain-independent Planning*, pages 1–8, 2020.