Neural Network Heuristic Functions for Classical Planning: Reinforcement Learning and Comparison to Other Methods

Patrick Ferber,[1, 3] **Florian Geißer,**[2] **Felipe Trevizan,**[2] **Malte Helmert,**[1] **Jörg Hoffmann**[3]

[1] University of Basel
first.last@unibas.ch
[2] The Australian National University
first.last@anu.edu.au
[3] Saarland University, Saarland Informatics Campus
last@cs.uni-saarland.de

Abstract

How can we train neural network (NN) heuristic functions for classical planning, using only states as the NN input? Prior work addressed this question by (a) supervised learning and/or (b) per-domain learning generalizing over problem instances. The former limits the approach to instances small enough for training data generation, the latter to domains and instance distributions where the necessary knowledge generalizes across instances. Clearly, reinforcement learning (RL) on large instances can potentially avoid both difficulties. We explore this here in terms of three methods drawing on previous ideas relating to bootstrapping and approximate value iteration, including a new bootstrapping variant that estimates search effort instead of goal distance. We empirically compare these methods to (a) and (b), aligning three different NN heuristic function learning architectures for cross-comparison in an experiment of unprecedented breadth in this context. Key lessons from this experiment are that our methods and supervised learning are highly complementary; that per-instance learning often yields stronger heuristics than per-domain learning; and that LAMA is still dominant but is outperformed by our methods in one benchmark domain.

1 Introduction

Neural networks (NN) have been shown to be able to learn powerful search guidance for several complex games. Successes include the AlphaGo/Zero system series (Silver et al. 2016, 2017, 2018), as well as heuristic search for single-agent puzzles including Rubik's Cube (Agostinelli et al. 2019). 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, Hoffmann, and Katz 2015), training NN as heuristic functions is highly promising there, too. Several works already addressed this problem with different methods and from different angles (Toyer et al. 2018; Garg, Bajpai, and Mausam 2019; Ferber, Helmert, and Hoffmann 2020; Shen, Trevizan, and Thiébaux 2020; Rivlin, Hazan, and Karpas 2020; Yu, Kuroiwa, and Fukunaga 2020; Karia and Srivastava 2021).

Here, we focus on learning heuristic functions for classical planning from scratch using only states as the NN input. Prior work addressed this by (a) supervised learning; and/or (b) *per-domain* learning which generalizes over all instances in a domain, where a "domain" fixes a set of predicates and action schemas, while instances specify object universe, initial state, and goal. Most works fall into category (b), exploring different variants of NN architectures based mostly on graph convolution (Garg, Bajpai, and Mausam 2019; Shen, Trevizan, and Thiébaux 2020; Rivlin, Hazan, and Karpas 2020; Karia and Srivastava 2021). Two works (Ferber, Helmert, and Hoffmann 2020; Yu, Kuroiwa, and Fukunaga 2020) use supervised learning (a) for *per-instance* learning, where the NN heuristic function generalizes only over the states in the state space of the instance (not over goals, nor over object universes), and simple feed-forward NN architectures can be used. Note that per-instance generalization is useful in settings where many different initial states may be encountered online, so that we can use the same learned heuristic function on all of these.

Per-instance supervised learning can yield NN heuristic functions competitive with the state of the art (Ferber, Helmert, and Hoffmann 2020), but it 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 it can train the NN on small instances. Yet this requires to transfer search knowledge from small instances to large ones, which is challenging and might work only for particular domains and instance distributions.

Clearly, per-instance RL can potentially avoid both difficulties. We explore this here in terms of three methods drawing on previous ideas, namely two variants of bootstrapping and one variant using approximate value iteration with Bellman updates.[1] For bootstrapping, we follow Arfaee, Zilles, and Holte (2011). We train a NN heuristic function by running it on increasingly difficult training states, generated by increasingly long backward walks from the goal. While Ar-

[1] We remark that the designation "RL" for these methods is debatable. We do not use rewards, and we do not learn an action policy. Nevertheless, the methods we explore share an essential characteristic with RL, in contrast to supervised learning: they iteratively improve the learned knowledge, generating training data for the next iteration based on the current knowledge. We are not aware of a separate term for this kind of method, and are using "RL" to avoid cumbersome terminology.

faee, Zilles, and Holte used pre-existing heuristic functions as input features (also in classical planning (Arfaee, Holte, and Zilles 2011)), we learn the heuristic function h^{Boot} from scratch using just the planning state as NN input. We furthermore introduce a new variant h^{BExp} which does not learn goal-distance estimation but an estimation of the number of states that greedy best-first search needs to expand when using the heuristic. We show that, under idealized settings, this learning process converges to h^*. For approximate value iteration, we follow Agostinelli et al. (2019). The NN heuristic function h^{AVI} is also trained on states s sampled with backward walks, but now we generate the labels with a k-step look-ahead: we evaluate h^{AVI} on the fringe states of that look-ahead, perform Bellman updates backwards to s, and use the final updated value for training on s.

We empirically compare these three RL-inspired methods to (a) per-instance supervised learning as per Ferber, Helmert, and Hoffmann (2020), as well as (b) per-domain learning using *hypergraph networks (STRIPS-HGN)* as per Shen, Trevizan, and Thiébaux (2020). Thus, we align three different NN heuristic function learning architectures for cross-comparison. Given that previous work in this area has never compared NN heuristics from different works against each other, this is an experiment of unprecedented breadth in this context. We invest substantial work into making the comparison fair, adjusting the training processes of prior works to be as comparable as possible. While this comparison still needs to be treated with care, we believe that such cross-comparison is ultimately required to advance the empirical field of NN learning in planning. For example, a plausible hypothesis is that per-instance learning produces better heuristics than methods that have to generalize over an entire domain. Ours is the first work to evaluate that hypothesis for NN heuristic functions in planning.

Specifically, our experiment shows the following key lessons:

- All NN heuristic functions are brittle, excelling in some domains while achieving hardly anything in others.
- The NN heuristic functions are highly complementary. In particular, RL is vastly superior to supervised learning in 4 out of 10 benchmark domains, but vastly inferior in 4 other domains.
- Per-instance learning often trains stronger heuristics than per-domain learning. HGN perform poorly in 7 of our domains.
- While the symbolic state-of-the-art planner LAMA (Richter and Westphal 2010) (specifically its first iteration, which is geared at finding a plan as quickly as possible) is generally still dominant, NN heuristic functions can outperform it in particular domains. In our experiments, this happens for the single domain (Storage) where LAMA's performance is weak.

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*, $s_{\mathcal{I}}$ is the *initial state*, and $\mathcal{G}$ is the *goal*. Every variable has a domain $\mathcal{D}$. A *fact* is a variable-value pair $\langle v, d \rangle$ where $v \in \mathcal{V}$ and $d \in \mathcal{D}_v$. The initial state is a complete variable assignment, i.e. a set which contains once fact for each variable. The goal is a partial variable assignment. Each *action* $a \in \mathcal{A}$ defines a *precondition* pre_a and an *effect* eff_a, both are partial variable assignments. An action a is applicable in a state s if $pre_a \subseteq s$. Applying a in s leads to the successor state $s' = \{\langle v, d \rangle \mid \langle v, d \rangle \in s, \neg \exists d' : \langle v, d' \rangle \in \mathit{eff}_a\} \cup \mathit{eff}_a$. The function $succ(s)$ generates all successor states of s, i.e. the set of states produced by applying the applicable actions of s to s. For simplicity we consider unit action costs (all costs are 1). A *plan* π is a sequence of actions $\langle a_1, \ldots, a_n \rangle$, such that starting from $s_{\mathcal{I}}$ and sequentially applying the action in π results in a state s^* with $s^* \subseteq \mathcal{G}$.

We need some further concepts. First, the *FDR regression* of a partial variable assignment G over an action a is defined as $(G \setminus \mathit{eff}_a) \cup pre_a$ if $\mathit{eff}_a \cap G \neq \emptyset$ and G is consistent with eff_a (i.e. there is no variable assigned different values by G and eff_a); 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 $h^*(s) = 1 + \min_{s' \in succ(s)} h^*(s')$ where h^* denotes the exact state value (here: goal distance). For an approximation h of h^*, a Bellman update is $h(s) := 1 + \min_{s' \in succ(s)} h(s')$.

3 Common Hyperparameters

Before we discuss our three RL-inspired methods in detail, we describe the general neural network architecture and the basic hyperparameters. We use RL to train heuristic functions for a given FDR task Π. Later, greedy best-first search (GBFS) uses these heuristics. As we have unit action costs, our learned heuristic functions are goal-distance estimators (rather than remaining-cost estimators).

We use residual networks (He et al. 2016) (which were successful in image classification, and have previously been used on Rubik's Cube (Agostinelli et al. 2019)), consisting of two dense layers, followed by one residual block with two dense layers, followed by a single output neuron. Each dense layer has 250 neurons. All neurons use the *ReLU* activation function. The inputs to our NN are states represented as fixed-size Boolean vectors. We associate every entry of the input vector with a fact of Π. If the fact corresponding to a vector entry is in the input state, then we set the entry to 1. Otherwise, we set it to 0. The output of our NN is a single number which represents a heuristic value. In contrast to previous work, we do not use an unary or one-hot encoding for the output, because we cannot know an upper limit for the heuristic values when the NN is first created. We use the *mean squared error* as loss function, and the *adam* optimizer with 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 it takes longer to generate a training batch of 250 samples than to train on that batch, we use experience re-

play. 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 and data generation and allows to run the two processes in parallel.

4 Bootstrapping

In what follows, we introduce three different RL-inspired methods. The present section is concerned with two variants based on *bootstrapping*, while the next section introduces a method based on *approximate value iteration*.

4.1 Bootstrapping a Goal-Distance Estimator

We follow the approach to bootstrapping as introduced by Arfaee, Zilles, and Holte (2011), training the heuristic function h on increasingly difficult training states s, generated by backward walks of increasing length. To obtain the training labels, a GBFS using the current h tries to solve those states s. 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, Zilles, and Holte (2011) also used an NN, their network was quite small (single layer) and used pre-existing (model-based) heuristic functions as input features. 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, Zilles, and Holte (2011) 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 FDR regression. We start at the goal of Π, and perform a random walk for n regression steps, where n is uniformly chosen from $\{0 \leq n \leq$*walk_length*$\}$ with *walk_length* being a growing (see below) parameter. The regression walk ends with a partial assignment. We randomly complete that partial assignment to a state, assigning each unassigned variable a random value.[2] We use Fast Downward's translator (Helmert 2009) to identify (some) mutually exclusive fact pairs (mutexes), and we enforce mutexes are respected.

Our other changes amount to parameter tuning. Like Arfaee, Zilles, and Holte (2011), we run GBFS with the current learned heuristic h^{Boot} to obtain labels 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 too uninformed to solve states sampled 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 has no effect anymore, so we double the maximum walk length at most 8 times.

[2] Arfaee, Holte, and Zilles (2011) also mention this possibility, though they suggest to randomly complete the goal (rather than the backward walk endpoint), which performed worse in our preliminary experiments.

4.2 Bootstrapping a Search-Space-Size Estimator

In addition to h^{Boot}, we designed a variant h^{BExp}, which does not learn a goal-distance estimation, but instead learns to estimate the number of states that GBFS will expand. The motivation is that what we really want to give preference to are states that minimize search space size. Goal distance estimates generally tend to correlate with search space size, but in a lose way given the highly volatile behavior of search as a function of the node ordering (small differences in goal distance estimates can lead to huge search space size differences). Estimating search space directly may thus provide a more direct link between the training objective and our ultimate objective to minimize search effort.

Specifically, h^{BExp} iteratively learns to estimate (via a bootstrapping process similar to h^{Boot}) the search space size of GBFS when using h^{BExp} as the heuristic function. While this definition is circular, it does make sense in that, in idealized form, h^{BExp} necessarily converges to h^*. Namely, define *idealized* h^{BExp} as using a lookup-table representation $G \in \mathbb{N}^{|\mathcal{S}|}$ of the heuristic function, obtained by n learning steps that iteratively update all states as follows:

$$G_n(s) = \begin{cases} \#\text{exp. of GBFS}(s, G_{n-1}) & \text{if solvable(s)} \\ \infty & \text{otherwise} \end{cases}$$

For convenience (in the proof below), we start counting the update iterations with $n = 0$. In $G_0(s)$, GBFS is run on an arbitrary initialization (any heuristic function).

The lookup table abstracts from the need to generalize across states in a compact ML model (like our neural network), and the update iterations abstract from the actual training of that ML model. We have:

Theorem 4.1 *With unit action costs, idealized h^{BExp} as just described converges to h^*.*

PROOF: Denote by $\mathcal{S}$ the set of all states and by H^*_n the set of states whose value is h^* after n updates: $H^*_n := \{s \mid s \in \mathcal{S}, G_n(s) = h^*(s)\}$. We show by induction that $H^*_n \supseteq \{s | s \in \mathcal{S}, h^*(s) \leq n\}$.

Induction basis: After iteration $n = 0$, all goal states have the value 0, so, $H^*_0 \supseteq \{s \mid s \in \mathcal{S}, h^*(s) = 0\}$

Induction step: s is a state with $h^*(s) = n$. Then s has a successor s' with $h^*(s') = n - 1$. By induction hypothesis, we have $H^*_{n-1} \supseteq \{s \mid s \in \mathcal{S}, h^*(s) \leq n - 1\}$ and $s' \in H^*_{n-1}$. Thus, there is a path P from s' to the goal with G_{n-1} decreasing by 1 in each step. A GBFS run on s generates s' when expanding s, and afterwards follows P (or another path of the same length) resulting in n expansions. Hence $H^*_n \supseteq \{s | s \in \mathcal{S}, h^*(s) = n\}$. The same argument applies by induction assumption to all states t where $h^*(t) < n$: GBFS follows a direct path to the goal. So we have $t \in H^*_n$, and hence $H^*_n \supseteq \{s | s \in \mathcal{S}, h^*(s) \leq n\}$ as desired.

In all updates, GBFS run on a dead-end state s proves that s is a dead-end. Thus, all states s with infinite h^* value also satisfy $h^*(s) = G_n(s)$, for all n. This proves the claim. □

In practice h^{BExp} is not guaranteed to converge to h^*. We cannot store a large enough lookup table in memory and have to resort to an approximator, e.g. a neural network. Furthermore, it is infeasible to perform an update on all states in each iteration. And finally, GBFS might run out of memory or require more time than available for a state. The first two arguments also show that any approximate value iteration is not guaranteed to converge to h^*.

Our bootstrapping procedure for h^{BExp} uses the same backward walks as before, and runs GBFS using the current heuristic function with a time limit of 10 seconds as before. If the search succeeds, we use the number of expanded states as training label; otherwise, we do the same thing with the number of states expanded up to the time limit as label. Our motivation for the latter is purely empirical – this turned out to be better than using only the successful searches. Intuitively, training on unsuccessful states s provides a useful training signal regarding which states are difficult to handle.

5 Approximate Value Iteration

Our third RL-inspired NN heuristic function h^{AVI} is trained using approximate value iteration, as follows.

Exact value iteration applies Bellman updates to a tabular value function h which maps every state s to a remaining-cost estimate. If every state s is updated infinitely often, h converges to h^* regardless of the update ordering (Bertsekas and Tsitsiklis 1996). The idea in approximate value iteration is to replace the value table with an approximate value function h, like an NN. Later, the NN can be used as a heuristic function. This has been successfully done in single-agent puzzles including Rubik's Cube (Agostinelli et al. 2019). Here, we adapt this approach to classical planning.

The generation of sample states s for training is done exactly as for bootstrapping, except that we keep the maximum walk length fixed. To generate the training labels, instead of running GBFS, we do a k-step look-ahead from s. The look-ahead depth k here is a parameter; we use $k = 2$ throughout. We evaluate the fringe states of the look-ahead search tree, as 0 if they are goal states and otherwise with h^{AVI}. Then we perform Bellman updates backwards, updating the values of intermediate states t in the tree with those of their children. The updated value at s is used for training.

6 Boosting NN Heuristics through Validation

In preliminary experiments, we observed that the performance of our NN heuristic functions is brittle. For a given benchmark instance, they often solve either all testing states or none at all, with the picture changing radically after retraining. Performance is thus drastically affected by the randomness during training (e.g. parameter initialization and random walks in training data generation). As a simple remedy, we introduce a validation method. For each benchmark instance, we generate 10 new initial states for validation in the same way the test states are generated. After training a NN, we run GBFS with the model as heuristic 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.

In our experiments, we compare our technique against two state-of-the-art techniques for learning NN heuristic functions, namely supervised learning (h^{SL}) by Ferber, Helmert, and Hoffmann (2020), and STRIPS-HGN (h^{HGN}) by Shen, Trevizan, and Thiébaux (2020). For fair comparison, we also use validation on those two approaches. As, in both approaches, training data generation is independent from training, we generate the training data only once and use it to train 10 models. We use the same validation states as for our methods. For h^{SL}, validation led to large coverage changes, while h^{HGN} was more robust (we observed only minor performance fluctuations).

7 Experiment Methodology

As our experiment is complex, we devote this section to its description. We first cover implementation, computational resources, and benchmarks as usual. We then proceed to the key new feature, comparing three different NN heuristic function learning architectures. We invested substantial work into making this comparison fair, and we describe our adjustments for this purpose in what follows.

7.1 Implementation and Time Limits

We implemented our NN heuristic functions on top of FD (Helmert 2006), starting from Ferber et al.'s (2020) code base, and used Lab (Seipp et al. 2017) for our experiments. We run all heuristic functions in greedy best-first search (GBFS), with 4 GB of memory. We run the search on a single core because we compare to the FF heuristic (h^{FF}, Hoffmann and Nebel 2001) and the LAMA planner (Richter and Westphal 2010) and they cannot exploit multiple cores. However, we 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 h^{FF}, which puts them at an intrinsic disadvantage; while 2) it is well known that NN evaluation can be sped up dramatically using GPUs or TPUs (Silver et al. 2016, 2018; Agostinelli et al. 2019), which arguably counter-acts this disadvantage in practice (also, Ferber, Helmert, and Hoffmann (2020) showed that multiple cores speed up state evaluation). We hence set a generous search-time limit of 10 hours, allowing the NN heuristics to exhibit strengths in informedness.

We run training (including data generation) for 28 hours on 4 cores of an Intel Xeon E5-2600 processor with 3.8 GB memory. We use the Keras framework (Chollet 2015) with Tensorflow (Abadi et al. 2015) as back-end.

We compare our goal-distance estimators based on bootstrapping (h^{Boot}), search-space-size estimators based on bootstrapping (h^{BExp}), and goal-distance estimators based on approximate value iteration (h^{AVI}) against the supervised learning approach (h^{SL}) of Ferber, Helmert, and Hoffmann (2020), STRIPS-HGN (h^{HGN}) by Shen, Trevizan, and Thiébaux (2020), the FF heuristic (h^{FF}) by Hoffmann and Nebel (2001), and LAMA (Richter and Westphal 2010). Our

code, benchmarks, and experimental results are online available(Ferber et al. 2021).

7.2 Benchmarks

We use the same benchmark domains as Ferber, Helmert, and Hoffmann (2020): Blocksworld, Depots, Grid, NPuzzle, Pipesworld-NoTankage, Rovers, Scanalyzer, Storage, Transport, and VisitAll. For each domain, Ferber, Helmert, and Hoffmann (2020) selected instances difficult enough to be interesting and easy enough to generate training data. We refer to these tasks as *moderate*, in contrast to the smaller *easy* tasks, which we ignore; and to the larger *hard tasks* we also consider. In Blocksworld and Grid, there were no such larger instances, so we generated new ones. We do not consider the benchmarks used by Shen, Trevizan, and Thiébaux (2020) to evaluate STRIPS-HGN, as all these instances are classified as easy or are very close to this classification.

For each benchmark instance, we evaluate our heuristic functions on 50 distinct test states. For the moderate tasks, we use the states published by Ferber, Helmert, and Hoffmann (2020). For the hard tasks, we use their method to create test states, by random forward walks (200 steps) from the original initial state.

7.3 Adaptations for $h^{\mathbf{SL}}$

Only minor changes are needed to compare with h^{SL}, which like our heuristic functions is based on per-instance learning. Ferber, Helmert, and Hoffmann (2020) generated training data for up to 400 hours, on a single CPU core. Then they trained for up to 48 hours using 4 CPU cores and 12 GB of memory. This exceeds our resource limits by far, and also Ferber et al. state themselves that, in many domains, a fraction of the training data is sufficient. Thus we adapted their resource limits to our setting, as follows.

For each benchmark instance, we generate training data on a single core for 56 hours. We train 10 heuristic functions. Each heuristic function is trained on two cores for up to 2.8 hours. Supervised learning has to keep the training data in memory, thus we use Ferber et al.'s original memory limit of 12 GB. We use validation as described above, and evaluate the resulting heuristic function h^{SL} in our experiments.

7.4 Adaptations for $h^{\mathbf{HGN}}$

STRIPS-HGN is designed for good performance with short training time, and in their original work Shen, Trevizan, and Thiébaux (2020) train the networks for only 10 minutes on small-sized problem instances. To provide a fair comparison, we adapted the training procedure of STRIPS-HGN to account for the extra training time and the source of training data used by the other learning approaches. Precisely, for each domain, we trained 10 different STRIPS-HGN networks simultaneously for up to 28 hours using 4 cores and 3.8 GB per core. We split the training time between data generation (10 hours) and network training (1.8 hour per network). Initially, we tried out different training parameters for STRIPS-HGN. We observed that, for Blocksworld, Scanalyzer and Transport, the original training time of 10 minutes and a shorter data generation time of 2 hours leads to more robust performance. We hence used this setup for these three domains. In all cases, we use the validation states to select the best STRIPS-HGN network per domain.

We generate the training data for STRIPS-HGN as follows. We sample, with replacement, a moderate or hard instance, perform the same backward walk as h^{Boot} and $h^{\mathrm{BExp}}$for n steps (see below), and solve the generated task using A* instead of GBFS (as in the original STRIPS-HGN). We repeat this procedure until time is up. We discard every task solved within 5 minutes as they are to easy. We also discard tasks not solved after 30 minutes as it is unlikely that we will find a solution. From the solved tasks, we use the states along the (optimal) plans as training data.

The random walk length n here is uniformly chosen from $\{\underline{n} \leq n \leq \overline{n}\}$ where $\underline{n}$ and $\overline{n}$ are initially 50 and 500, respectively. Whenever our procedure generates an easy task, it updates the lower bound $\underline{n}$ to $(\underline{n} + 3n)/4$; whenever our procedure generates a timed-out task, it updates the upper bound $\overline{n}$ to $(\overline{n} + n)/2$. The number of state-value pairs obtained following this procedure ranges from 78 for Transport to 1563 for VisitAll. Al the mentioned parameters were tuned so as to optimize h^{HGN}'s performance in GBFS.

8 Experiment Results

Table 1 summarizes our empirical findings in terms of coverage, i.e. the fraction of solved test states when using the different heuristic functions in GBFS. The table is split into three parts, which we will discuss in turn below. The effect of validation as per Section 6 is evaluated separately, on the moderate tasks, through the left part (w/o validation) and middle part (w/ validation) of the table. The right part of the table evaluates the heuristic functions' capability to scale to the hard tasks. Within each table part, we highlight the best-performing NN heuristic function in bold.

8.1 Validation

Consider first the data regarding validation, comparing the left part of Table 1 to the middle part. The data shows that for each of our RL techniques and in most domains, coverage increases. The improvement is often substantial, e.g. from 31.7% to 60.3% for h^{Boot} in Depots. As validation hardly ever deteriorates performance (the only case in our data is from 61.% to 57.5% for h^{BExp} in Storage), we keep it switched on in what follows.

8.2 Coverage Comparison for Moderate Tasks

Consider now the middle part of Table 1, and consider first the relative performance of our three RL-inspired methods. No method dominates the others in all domains. h^{Boot} has highest coverage in 7 domains, h^{BExp} in 4. Both techniques solve all states in Grid and Transport. h^{AVI} is close to the highest coverage in 2 domains.

Comparing the RL methods against h^{SL}, we see that h^{SL} outperforms RL in 4 domains, h^{Boot} outperforms h^{SL} in 6 domains, h^{BExp} outperforms h^{SL} in 2 domains, and h^{AVI} outperforms h^{SL} in 2 domains. Most strikingly, the coverage differences are drastic in many domains. For example, the RL methods have poor coverage in Blocksworld, while h^{SL}

Domain	Moderate Tasks without Validation			Moderate Tasks with Validation							Hard Tasks with Validation						
	h^{Boot}	h^{BExp}	h^{AVI}	h^{Boot}	h^{BExp}	h^{AVI}	h^{SL}	h^{HGN}	h^{FF}	LAMA	h^{Boot}	h^{BExp}	h^{AVI}	h^{SL}	h^{HGN}	h^{FF}	LAMA
blocks	0.0	0.0	0.0	18.0	0.0	0.0	80.4	**100.0**	98.8	100.0	0.0	0.0	0.0	0.0	**50.0**	61.6	96.8
depots	31.7	17.7	**43.7**	60.3	32.7	54.7	**90.3**	0.0	98.0	100.0	8.3	4.3	12.9	**35.4**	0.0	36.0	82.6
grid	**100.0**	**100.0**	51.0	**100.0**	**100.0**	51.0	93.0	0.0	96.0	100.0	87.8	**95.0**	70.5	60.2	0.0	53.2	100.0
npuzzle	**27.0**	0.0	1.0	**28.0**	0.0	1.0	0.0	0.3	97.5	100.0	0.0	0.0	0.0	0.0	0.0	33.2	86.5
pipes-nt	36.2	**51.2**	21.4	57.8	68.4	50.2	**92.2**	7.6	82.4	99.4	23.4	19.1	8.0	**48.7**	0.0	27.4	69.3
rovers	**36.5**	15.2	34.2	**48.2**	21.8	45.0	26.0	14.0	84.2	100.0	2.8	0.8	**6.5**	1.5	0.3	13.9	100.0
scanalyzer	33.3	59.7	**66.7**	33.3	70.7	67.3	**82.7**	11.0	98.3	100.0	3.3	0.0	**60.7**	60.0	0.0	98.0	100.0
storage	**89.0**	61.0	67.0	**89.0**	57.5	69.5	24.5	0.0	48.0	38.5	**27.2**	13.2	15.8	0.0	0.0	13.8	11.5
transport	**83.8**	79.5	70.0	**100.0**	**100.0**	87.5	99.2	94.7	98.5	100.0	0.0	0.0	**2.4**	0.0	0.0	0.0	92.8
visitall	**17.0**	0.0	0.0	55.3	0.0	0.0	0.0	**100.0**	93.3	100.0	28.0	0.0	0.0	0.0	**100.0**	74.0	100.0

Table 1: Coverage (in %) of all techniques. The left block shows the coverage of our approaches on the moderate task set if no validation is used. The middle and the right block show the coverage of all techniques on the moderate respectively tasks. Here, we use validation for all learning based techniques. Per block, we highlight the best learning based technique.

solves 80% of the moderate tasks. In Rovers, the tables have turned, with h^{SL} solving 26% while h^{Boot} and h^{AVI} solve twice as many.

Adding h^{HGN} to the comparison, we see that it excels in 2 domains, namely Blocksworld and VisitAll where the RL approaches have difficulties; and delivers good performance in Transport; but has very poor coverage in all other 7 domains. Often, the reason is the hypergraph size underlying STRIPS-HGN, which scales with the task size. For many tasks in Depots, Storage and Grid, the hypergraph size exceeds memory. 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.

The primary conclusion from this data is that *the different NN heuristic functions are highly complementary to each other*. No heuristic dominates any other, and each approach favours a subset of the domains. There seems, however, to be a tendency that per-instance learning can often yield more effective heuristics than per-domain learning, at least as represented by h^{HGN}.

Turning the comparison to model-based planners, as in the experiments run by Ferber, Helmert, and Hoffmann (2020), the data shows that LAMA and h^{FF} are highly competitive across all domains, generally outperforming all learning-based approaches. Indeed LAMA has perfect or almost perfect coverage everywhere, so it is impossible to beat on these benchmarks. The exception is Storage, which is the single domain where LAMA struggles. In that domain, all our three RL approaches – but neither h^{SL} nor h^{HGN} – outperform LAMA. h^{Boot} solves almost 90% of the tasks compared to only 39% for LAMA and 48% for h^{FF}.

8.3 Coverage Comparison for Hard Tasks

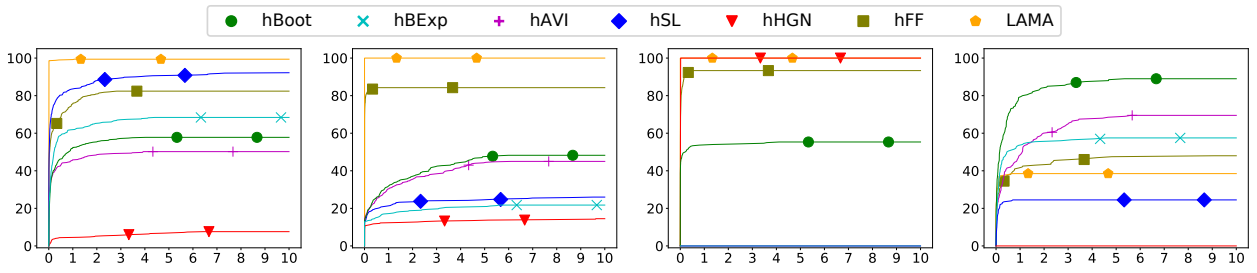
Turning now to the hard tasks (right block of Table 1), we observe that coverage drops for all techniques in almost all domains. Qualitatively though, the comparison across approaches is similar to the moderate tasks. The learned heuristic functions are still highly complementary, indeed now every such heuristic has domains in which it yields highest coverage. h^{HGN} performance drops to near-0 in Transport, but the same happens for all other learning-based methods; in Blocksworld and VisitAll, h^{HGN} still excels. Regarding h^{SL}, the original work on that heuristic (Ferber, Helmert, and Hoffmann 2020) did not train networks for the hard tasks as training data generation was expected to be a bottleneck. Indeed, this is what happens in some domains, especially in Depots, NPuzzle, Rovers, and Storage. In the other domains, the amount of training data also decreases with growing instance size (to varying degrees). Yet in some cases – Depots, Grid, Pipesworld-NoTankage, and Scanalyzer – the data is still sufficient to learn useful heuristic functions. As before, LAMA is outperformed in Storage where all our three RL techniques have higher coverage; the only significant advantage now is for h^{Boot} though.

Given our comparatively large search time limit of 10 hours, Figure 1 shows coverage as a function of runtime. We show results for moderate tasks in four domains; the data is qualitatively similar in the other domains and tasks.

Comparing different NN heuristics, the general finding is that coverage superiority persists over any time limit. With few exceptions, an approach that is better after 30 minutes is still better after 2 or more hours. The picture with respect to LAMA is 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!), and thus require some time to "catch up" with LAMA. They still solve additional tasks even after very long run-times, and relative performance differences become more pronounced over time. In particular, the advantages over LAMA in Storage grow as a function of the run-time limit.

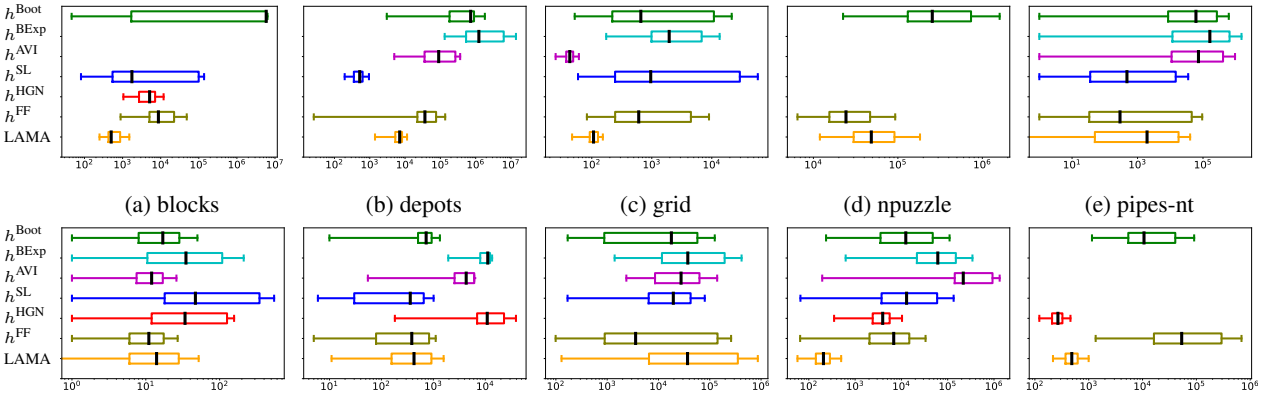
8.4 Informedness

Let us finally compare informedness across the different approaches, measured in terms of the number of expansions. Figure 2 shows the distribution of expansions per domain, for commonly solved tasks. In each domain we ignore algorithms that solve less than 10% of the tasks, as otherwise the



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

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



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

(f) rovers (g) scanalyzer (h) storage (i) transport (j) visitall

Figure 2: Expansions on commonly solved moderate tasks, removing algorithms with coverage $< 10\%$. In each plot, the line within the body indicates the median, the body of the box plot indicates the 25 and 75 percentile, and the whiskers show the 5 and 95 percentiles.

set of commonly solved tasks would become too small.

Again, the primary conclusion from these results is that *the techniques are highly complementary* – at a glance, just consider how the different colors in Figure 2 move to and fro in the plots. Comparing neural network heuristic functions against each other, h^{HGN} is only well informed in the 3 domains in which it yields high coverage. The comparison between h^{SL} and our RL methods is similar as for coverage, exhibiting performance differences in the same domains (which is expected as the per-state runtime of these heuristic functions is very similar). Finally, the NN heuristic functions are quite competitive with h^{FF} and LAMA in terms of informedness. In Depots, Grid, and VisitAll the lowest number of expansions is achieved by a NN heuristic function (a different one in each case); and in Pipesworld-NoTankage, Rovers, and Scanalyzer, the best NN heuristic function is basically on par with h^{FF}.

9 Conclusion

We explored how to arrange RL ideas to learn NN heuristic functions in classical planning, taking only the state as input; we designed three different methods along these lines (h^{Boot}, h^{BExp}, h^{AVI}). We conducted an experiment of unprecedented scope in this area, comparing our RL-inspired methods to two other NN heuristic function approaches – h^{SL} by Ferber, Helmert, and Hoffmann (2020) and h^{HGN} by Shen, Trevizan, and Thiébaux (2020) – including in particular a comparison between per-domain learning and per-instance learning. The results show that the heuristic functions are extremely complementary. In particular, h^{Boot} achieves competitive performance, outperforming both h^{SL} and h^{HGN} in 4 out of 10 domains. In one domain – namely the single domain where LAMA has severe performance issues – h^{Boot} even outperforms LAMA. To our knowledge this is the first time that an NN heuristic function was reported to beat LAMA. Elsewhere though, LAMA still reigns supreme.

The major open question in our view remains whether and how more reliable performance can be obtained with NN heuristic functions in planning. As of now, informedness varies wildly across domains for all methods evaluated here. One may argue that this phenomenon pertains to virtually all heuristic functions, but our impression is that this is more drastic for NN than for model-based techniques. Coverage rises high vs. drops to rock-bottom for the same method even in intuitively closely related domains. The same kind of variance occurs even in multiple training runs on the same planning task, so that a simple validation method boosts performance. How can we improve this erratic behavior? This question remains wide open. We speculate that other learning tasks, closer to the requirements of search than numeric heuristic values, may help; or perhaps architectures in which the heuristic function is represented in a neuro-symbolic manner rather than monolithically through a neural network.

10 Acknowledgments

This work was funded by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science), and by the Swiss National Science Foundation (SNSF) as part of the project "Certified Correctness and Guaranteed Performance for Domain-Independent Planning" (CCGP-Plan).

References

Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Goodfellow, I.; Harp, A.; Irving, G.; Isard, M.; Jia, Y.; Jozefowicz, R.; Kaiser, L.; Kudlur, M.; Levenberg, J.; Mané, D.; Monga, R.; Moore, S.; Murray, D.; Olah, C.; Schuster, M.; Shlens, J.; Steiner, B.; Sutskever, I.; Talwar, K.; Tucker, P.; Vanhoucke, V.; Vasudevan, V.; Fern; a Viégas; Vinyals, O.; Warden, P.; Wattenberg, M.; Wicke, M.; Yu, Y.; and Zheng, X. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. URL http://tensorflow.org/.

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence* 1: 356–363. doi:10.1038/s42256-019-0070-z.

Arfaee, S. J.; Holte, R. C.; and Zilles, S. 2011. Bootstrap Planner: an Iterative Approach to Learn Heuristic Functions for Planning Problems. In *IPC-7 planner abstracts*.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning Heuristic Functions for Large State Spaces. *AIJ* 175: 2075–2098.

Bäckström, C.; and Nebel, B. 1995. Complexity Results for SAS^+ Planning. *Computational Intelligence* 11(4): 625–655.

Bertsekas, D. P.; and Tsitsiklis, J. N. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

Chollet, F. 2015. Keras. https://keras.io. URL https://keras.io.

Domshlak, C.; Hoffmann, J.; and Katz, M. 2015. Red-black planning: A New Systematic Approach to Partial Delete Relaxation. *AIJ* 221: 73–114.

Ferber, P.; Geißer, F.; Trevizan, F.; Helmert, M.; and Hoffmann, J. 2021. Code, benchmarks and experiment data for the PRL 2021 workshop paper "Neural Network Heuristic Functions for Classical Planning: Reinforcement Learning and Comparison to Other Methods". https://doi.org/10.5281/zenodo.5026899.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020. Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space. In *Proc. ECAI 2020*, 2346–2353.

Garg, S.; Bajpai, A.; and Mausam. 2019. Size Independent Neural Transfer for RDDL Planning. In *Proc. ICAPS 2019*, 631–636.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778. ISSN 1063-6919. doi:10.1109/CVPR.2016.90.

Helmert, M. 2006. The Fast Downward Planning System. *JAIR* 26: 191–246.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *AIJ* 173: 503–535.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proc. ICAPS 2009*, 162–169.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *JACM* 61(3): 16:1–63.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *JAIR* 14: 253–302.

Karia, R.; and Srivastava, S. 2021. Learning Generalized Relational Heuristic Networks for Model-Agnostic Planning. In *Proc. AAAI 2021*.

Kingma, D. P.; and Ba, J. 2015. Adam: A Method for Stochastic Optimization. In *Proc. ICLR 2015*.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *JAIR* 39: 127–177.

Rivlin, O.; Hazan, T.; and Karpas, E. 2020. Generalized Planning With Deep Reinforcement Learning. In *ICAPS Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning (PRL)*, 16–24.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Shen, W.; Trevizan, F.; and Thiébaux, S. 2020. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. In *Proc. ICAPS 2020*, 574–584.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529(7587): 484–489.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362(6419): 1140–1144.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; Chen, Y.; Lillicrap, T.; Hui, F.; Sifre, L.; van den Driessche, G.; Graepel, T.; and Hassabis, D. 2017. Mastering the Game of Go Without Human Knowledge. *Nature* 550(7676): 354–359.

Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies with Deep Learning. In *Proc. AAAI 2018*, 6294–6301.

Yu, L.; Kuroiwa, R.; and Fukunaga, A. 2020. Learning Search-Space Specific Heuristics Using Neural Network. In *ICAPS Workshop on Heuristics and Search for Domain-independent Planning*, 1–8.