Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space

Patrick Ferber[1] and **Malte Helmert** [1] and **Jörg Hoffmann**[2]

Abstract. Neural networks (NN) have been shown to be powerful state-value predictors in several complex games. Can similar successes be achieved in classical planning? Towards a systematic exploration of that question, we contribute a study of hyperparameter space in the most canonical setup: input = state, feed-forward NN, supervised learning, generalization only over initial state. We investigate a broad range of hyperparameters pertaining to NN design and training. We evaluate these techniques through their use as heuristic functions in Fast Downward. The results on IPC benchmarks show that highly competitive heuristics *can* be learned, yielding substantially smaller search spaces than standard techniques on some domains. But the heuristic functions are costly to evaluate, and the range of domains where useful heuristics are learned is limited. Our study provides the basis for further research improving on current weaknesses.

1 Introduction

Heuristic search [26, 6] is among the most successful approaches to classical planning, both in the satisficing (e.g., [20, 27, 14]) and in the optimal setting (e.g., [18, 19, 28]). The key ingredient in planning as heuristic search are efficiently computable evaluation functions, *heuristics*, that map states of the planning tasks to numerical estimates of the distance or cost of reaching the nearest goal state. Roughly speaking, the closer these estimates are to the true distance, the better we can expect a heuristic search algorithm to perform.

Similar evaluation functions have long been used in the context of board games (e.g., [24]), where they are used to estimate the winning chances of a given player from a given game position. Recently, the AlphaGo [30] system has shown that highly accurate evaluation functions for Go can be represented as neural networks (NN) that are learned automatically through self-play. Later work has successfully extended the approach to the games of chess and shogi [31], and recent work has successfully tackled several single-agent puzzles [2]. These successes raise the question whether heuristics based on NN can be equally successful in the context of classical planning.

There are several challenges that make it non-trivial to apply this methodology to AI planning. Firstly, game boards typically have a grid topology, akin to images, allowing to naturally leverage NN architectures originating in image processing [13, 12, 16], a critical aspect of many recent breakthrough achievements. In contrast, general AI planning tasks usually do not have a similar grid structure, and when they do it is not apparent from their representation.

Secondly, games like Go tend to end with a win or loss comparatively quickly even with random or almost random play,[3] which is critical for reinforcement learning techniques that rely on meaningful feedback on good or bad action sequences. In contrast, it is unclear which meaningful feedback can be obtained from random or almost random action sequences in a planning task: if they lead to a goal state, the planning task was easy to solve in the first place.

Thirdly, systems like AlphaGo work with a fixed state space and fixed objective: the rules of Go do not change, and therefore it pays off to invest enormous computational resources into an accurate value function for a single state space. In contrast, AI planning is traditionally focused on domain-independent technology, capable of dealing with a wide variety of different planning tasks.

Despite these difficulties, NN state estimators or policy functions have successfully been learned for classical planning and related problems. In some approaches [4, 33], most of the difficulties are circumvented by training a neural network to *combine* heuristic estimates, which is easier than learning heuristics from scratch. Other works focus on NN learning for *probabilistic* planning tasks [9, 21, 32], which makes it easier to achieve competitive performance. As for games like Go, the solution to probabilistic planning tasks are policies whose size is often prohibitive for exhaustive methods; and strong domain-independent state estimators are not available to the same extent as in classical planning. Toyer et al. [32] demonstrated that it is possible to learn useful policies also at the level of *planning domains*, i.e., for infinite families of planning tasks of scaling size. However, while the approach works very well in domains with a simple repetitive structure, it can also be fooled easily [**?**], and the applicability to more complex domains remains challenging.

Here, we address the problem of learning NN heuristic functions for classical planning. Like prior work on probabilistic planning, we use the raw state description as input to the NN, employ feed-forward NN to process that generic input, and use supervised learning to train these NN. Like prior work except that by Toyer et al., we consider generalization only over states, i.e., an NN is trained for a fixed state space with a fixed goal. Arguably, this is the most canonical setup for NN heuristic learning in classical planning, addressing the above stated three challenges in the simplest possible way.

We choose this setup on the rationale that simple problems need to be properly understood before one can hope to understand more complex ones. In particular, while Toyer et al.'s approach applies also to classical planning, generalization over an entire planning domain means that information about an individual planning task must be compressed to an extent that, in general, complex policies cannot be

[1] University of Basel, Switzerland, email: firstname.lastname@unibas.ch
[2] Saarland Informatics Campus, Saarland University, Germany, email: hoffmann@cs.uni-saarland.de

[3] In its initial stages, AlphaGo and Alpha Zero use a random evaluation function combined with Monte-Carlo tree search (e.g., [25, 8]).

represented. Our research strategy hence is to address the problem from the ground up, understanding step-by-step what NN heuristics can or cannot do in classical planning.

Furthermore, while simple, generalization over states can still pay off in practice, when the goal is fixed but exogenous behavior frequently affects the system state so that a new search for the same goal but on a different starting state needs to be launched. A common setting where this occurs is re-planning. Also, some applications are characterized by repetitive tasks under volatile conditions, like patrolling (which is volatile e.g. for autonomous aerial or underwater vehicles).

We contribute a comprehensive study of hyperparameter space in the canonical setup. We investigate a broad range of hyperparameters pertaining to NN design and training: classification vs. regression, number of hidden layers, activation functions, regularization, how to select training states, training data balancing, and pruning. We evaluate points in this parameter space through the performance of the resulting heuristic functions when used for search in Fast Downward [17]. We use IPC benchmarks where dead-ends don't exist and thus goal-distance estimation does not encompass dead-end detection (which is a qualitatively different problem).

Comparing performance to standard planning techniques, we find that NN heuristics are slower to evaluate than h^{FF}, but tend to be more informative. In some domains, the advantage outweighs the overhead, resulting in better runtime and/or task coverage. A comparison to the state of the art exhibits similar behavior, and a straightforward combination with preferred operators from h^{FF} yields highly competitive performance.

The paper is organized as follows. After briefly outlining our planning framework in Section 2, we explain the fixed aspects of our setup in Section 3. We examine the impact of using regression vs. classification networks in Section 4, we evaluate NN architecture parameters in Section 5, and we examine parameters of NN training in Section 6. We then run a performance comparison of our best configuration against state-of-the-art planners in Section 7. We finally compare in Section 8 NN with a collection of simpler ML models, showing that neural networks are indeed necessary in our context to obtain accurate heuristic functions. Section 9 concludes the paper with an outlook on future work.

2 Planning Framework

We use the *FDR* planning framework [5]. 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, and a non-negative cost $c_a \in \mathbb{R}_0^+$. 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 *fact* is a variable value pair $\langle v, d \rangle$ where $v \in \mathcal{V}$ and $d \in \mathcal{D}_v$.

A *plan* is a sequence of actions $\pi = \langle a_1, \ldots, a_n \rangle$, such that sequentially applying each action in π from $s_{\mathcal{I}}$ leads to a state that satisfies $\mathcal{G}$. A plan is optimal if no other plan has smaller summed-up action costs.

3 Setup

Before we delve into the details regarding the hyperparameters concerning NN architectures and training, let us first clarify some things that are fixed in our setup.

As stated, the major limitation in our learning setting is that we generalize only over states. We learn a heuristic function per planning task. Precisely, we consider only states reachable from the original initial state (in particular, keeping static predicates fixed). The generalization is to different initial states within the same reachable state space.

Our implementation is in Fast Downward (FD). Given a task Π, to generate training data for supervised learning, we perform random walks from the initial state $s_{\mathcal{I}}$, a simple generic method that is applicable in general. We fixed the walk length to 200 steps, making the probability of generating the same state twice negligible. For each state s sampled in this manner, we use a *teacher search* to solve s, within time/memory limits of 30 minutes/3.8 GB. If a plan π is found, we store the states along π along with their goal distance according to π. Different subsets of this data can be used for training, as we shall discuss below. We parallelize this process on a cluster of 200 Intel Xeon E5-2660 cores, with a combined time-out of 400 hours (around 2 hrs/core) to allow the generation of sufficient training data even on large instances.

Our goal-distance prediction NN receive as input a Boolean vector representing all non-static facts in the task. We use fully connected feed-forward NN, with different output layers interpreted as heuristic values (classification vs. regression as discussed below). The number of hidden layers is a parameter (i.e., a hyperparameter we will investigate). We scale the number of neurons in the hidden layers in equal size steps from input layer size to output layer size. Note that, thus, the number of tuneable parameters (weights) in the NN grows with the number of facts, thus adapting the NN's learning capacity to task size.

We perform 10-fold cross-validation, i.e., we split the data into 10 folds and we train 10 different NN. Each NN uses 9 folds as training data and the remaining 1 fold as validation data (test instances are generated separately, see below). We use the *adam* optimizer [22] on the *mean squared error (MSE)* with a batch size of 100. We do so for up to 24 hours, up to 1000 epochs, or until the loss on the validation data does not improve anymore. The training is implemented in Python2 using the Keras framework [11] with Tensorflow [1] as back-end. The training for each NN is run on 4 CPU cores, with a 12 GB memory limit for the training data.

Regarding benchmarks, we use unit costs only, which simplifies the use of classification to encode heuristic values. More importantly, as already stated, we restrict ourselves to domains without dead-ends. We initially selected 10 IPC domains, namely Blocksworld, Depots, Grid, NPuzzle, Pipesworld-NoTankage, Rovers, Scanalyzer, Storage, Transport, and VisitAll. In three of these domains, no useful heuristic functions were learned, i.e., only few test instances were solved. These domains are not suited for hyperparameter evaluation as search performance there is almost uniformly bad. We will therefore include these domains only in our discussion of competitive performance (Section 7), identifying the main reasons for lack of performance; everywhere else, we focus on the 7 domains were interesting results were obtained. These are Blocksworld, Depots, Grid, Pipesworld-NoTankage, Scanalyzer, Storage, and Transport. From these domains, we consider the subset of IPC instances that are solved by our teacher search in > 1 second and < 900 seconds, filtering out tasks that are too easy to be interesting or too hard to generate training data. We end up with 60 tasks as the benchmark collection for our experiments.

For all evaluation purposes, we use search performance on a set of test instances generated from these benchmark tasks. Namely, on

each benchmark task we performed random walks from $s_{\mathcal{I}}$ to obtain 200 states (fresh states, i.e., in the rare case that a state had already been generated in training, it was not used). These 200 states are used as the initial states in 200 test instances per benchmark. In our evaluations, each heuristic trained in one of the 10 cross-validation folds is used to solve 20 distinct instances out of these 200. We use greedy best-first search, a canonical search algorithm given that admissibility cannot be guaranteed. The time/memory limits for solving each test instance are 30 minutes/3.8 GB.

All benchmarks and test instances, as well as our source code, can be accessed at `https://doi.org/10.5281/zenodo.3671553`.

The remainder of this paper evaluates hyperparameters in our framework, and the overall competitive performance of the resulting learned heuristic functions. As default settings, if not otherwise specified, the teacher search is greedy best-first search with h^{FF} [20]; for training we randomly select one state from each plan generated by that search; the NN have 3 hidden layers with sigmoid activation functions; and we do not use any regularization.

Domain	#samples	teacher values	#parameters
blocks	504K	145/327	570K
depots	98K	46/414	722K
grid	123K	72/112	3M
pipes-nt	87K	91/411	1.4M
scanalyzer	43K	19/100	225K
storage	10K	107/133	662K
transport	152K	56/130	940K

Figure 1. Statistics about the training setup: Median number of training samples; smallest and largest maximal teacher value over all tasks in a domain; and median number of tuneable parameters in the default network architecture.

Figure 1 shows some statistics elucidating key aspects of our setup. First, "#samples" highlights the amount of training data (using a single state per plan, as discussed in detail below). The number of training states ranged from 3K to 500K, and was less than 20K only on three tasks. Second, we assess the range of heuristic values to be predicted, which is relevant for the output size of classification NN (see below). We see that the maximum heuristic value is often moderate, but can sometimes be large. Third, column "#parameters" shows the number of tuneable parameters in our NN, which varies between 200K and 3000K. Grid has by far the largest number of parameters, due to its large input size where every key object and the robot can be moved to each tile on the grid.

4 Classification vs. Regression

The way the network output is modeled has a huge impact on its performance: should we use *regression* or *classification*? Regression is the most obvious choice as heuristics values are numbers. On the other hand, often classification networks perform unexpectedly well.

We explore three kinds of output models, one using regression and two different ones using classification. The regression networks have a single output neuron, whose output can directly be used as the heuristic estimate. That neuron uses a *rectified linear unit (ReLU)* activation function, which takes the maximum of the input and 0 so that non-negative heuristic estimates are returned. In each of the two classification models, there are $n + 1$ output units where n is the largest teacher value during training (cf. Figure 1). The difference between the two lies in how these outputs are interpreted. First, in the *one-hot* encoding, the heuristic value h is represented during training by setting output number h to 1 and all others to 0. Hence the i-th class in the classification represents that the heuristic value is i. The output layer uses a softmax activation function to obtain a probability distribution over the classes. The heuristic estimate is the class with the highest probability. Second, in the *unary* encoding [10], heuristic value h is represented during training by setting all outputs $\leq h$ to 1 and those $> h$ to 0. In other words, here the i-th class represents that the heuristic value is $\geq i$. The output layer uses a *sigmoid* activation function, mapping into the open interval $(0, 1)$. To interpret this output as a heuristic function, neuron outputs > 0.01 are treated as 1, others as 0. The heuristic estimate is the highest index i so that all outputs $\leq i$ are set to 1. For example, the output vector 1101 is interpreted as $h = 1$. (The threshold 0.01 consistently performed better than higher or lower thresholds in preliminary experiments, so we fixed it to that value.)

	cls_{OH}	cls_{U}	reg
blocksworld	93.4	97.2	65.3
depots	87.7	76.2	77.3
grid	44.8	93.2	71.0
pipes-nt	84.3	89.6	82.0
scanalyzer	96.2	94.6	80.3
storage	14.5	95.5	98.5
transport	92.2	99.1	88.9
Average	73.3	92.2	80.5

Figure 2. Coverage (% of test instances) of classification with one-hot encoding (cls_{OH}) vs. classification with unary encoding (cls_{U}) vs. regression (reg). All networks have 3 hidden layers.

Figure 2 shows coverage data when using the resulting heuristic functions in FD. Regression networks ("reg" column) are inferior to classification ("cls_{OH}" and "cls_{U}" columns) except in Storage where they yield a small advantage over classification with the unary encoding ("cls_{U}"). Comparing the classification networks to each other, there is some per-domain variance, but generally the unary encoding is more robust and yields clearly superior performance overall. These observations motivate the use of classification with a unary encoding, and we shall henceforth stick to that setting.

5 Hidden Layers, Activation, Regularization

We now shed light on NN architecture hyperparameters, specifically the number of hidden layers, the activation functions used in the hidden layers, and regularization methods. Figure 3 shows all the data. We discuss this for each hyperparameter in turn.

Regarding the number of hidden layers, the leftmost part of Figure 3 shows that networks with 1 and 3 hidden layers perform best. Using 5 layers still performs well, while 0 hidden layers result in much worse performance. In preliminary experiments, networks with more than 5 hidden layers solved hardly any test instances. To shed some light on these differences, the next part of the figure shows search space size data for NN with 1, 3, and 5 hidden layers. We can see that overall the NN heuristics get more informed for more hidden layers. The downside of using more hidden layers is increased NN evaluation time, i.e., slower heuristic functions. The median number of states expanded per second for 1, 3, and 5 hidden layers is 1010,

	coverage #hidden layers				median #expansions #hidden layers			coverage activation					
	0	1	3	5	1	3	5	ReLU	dropout rate 0.2	0.4	L2 regularization weight 0.1	1	10
blocks	30.7	100.0	97.2	83.3	585	899	2122	100.0	95.7	90.9	0.0	0.0	0.0
depots	70.7	80.6	76.2	75.9	499	182	129	77.6	81.2	77.8	0.2	0.2	0.2
grid	41.8	94.0	93.2	47.8	14K	373	2761	78.2	77.0	72.2	0.0	0.0	0.0
pipes-nt	70.9	88.9	89.6	74.8	818	697	699	84.2	89.3	90.1	7.2	7.2	7.4
scanalyzer	49.0	88.2	94.6	83.2	360	173	160	96.5	81.8	73.8	15.4	15.4	15.4
storage	100.0	100.0	95.5	55.0	7155	1917	25K	99.5	56.0	31.5	0.0	0.0	0.0
transport	58.3	98.3	99.1	99.7	47K	2910	719	89.6	99.9	99.9	0.0	0.0	0.0
Average	60.2	92.9	92.2	74.3	-	-	-	89.4	83.0	76.6	3.3	3.3	3.3

Figure 3. Coverage (%) for different number of hidden layers, activation functions, dropout rates, and L2 regularization weights. Median #expansions computed over commonly solved tasks. (Recall that the default configuration has 3 hidden layers, uses a sigmoid activation function and no regularization technique; its coverage data is given here in column "3" of the leftmost part.)

365, 159 respectively. Overall, 3 hidden layers provide the most robust trade-off between informedness and speed, so we fix this as default value.

Regarding the activation functions, up to now all neurons in the hidden layers use the sigmoid activation function. A popular alternative to use the aforementioned ReLU function instead. As Figure 3 shows, however, this performs worse than sigmoid activation here.

Regularization methods are a means to avoid overfitting. We experiment with two such methods here, (1) *dropout* and (2) *L2 regularization*. Regarding (1), in dropout every hidden neuron is assigned a dropout probability. During training, nodes are omitted from the network evaluation with their dropout probability, making the network less reliant on just a few nodes. Regarding (2), L2 regularization adds the L2 norm of the network parameters to the loss during training, to punish large network parameters. Thus only performance-critical weights will remain large, creating a bias to "simpler" models. The strength of this bias is adjusted by a weight for the L2 norm. As the results in Figure 3 show, unfortunately neither of the two methods tends to be useful in our context (with the single mild exception of Depots where performance improves slightly with dropout).

6 Training

Next, we examine different ways to use the available data for training. In the first part, we experiment with different strategies to *select* samples from the generated data to be used during training. In the second part, we evaluate *weighting* the samples to counter-act unbalanced distributions of heuristic values in the training data; we discuss this alongside with *pruning* which avoids training data duplication.

During data generation we store for every solved task all states s_i along the plan found with their cost c_i to the goal $\langle(s_1, c_1), (s_2, c_2), \ldots, (s_n, 0)\rangle$. A straightforward choice is to train on all available state-cost pairs. We call this sample selection strategy *entire-plan*. This approach provides the largest possible number of training states. A potential issue though is that these states are often highly correlated, with small changes from one state to the next (e.g. the position of a single block in Blocksworld). Furthermore, as the space of states close to the goal is typically small, those states tend to be frequently visited so that the training data set is highly repetitive in that region (such states are often visited thousands to hundreds of thousands of times). An alternative that addresses both issues is to select a single randomly chosen pair (s_i, c_i) per plan; we call this the *random-state* selection strategy. The training states thus selected tend to be less close than in entire-plan, and they tend to contain less repetitions. The disadvantage of course lies in the amount of training data – given the same set of teacher plans, the number of training states is reduced by a factor of 10 to 138 in our benchmarks.

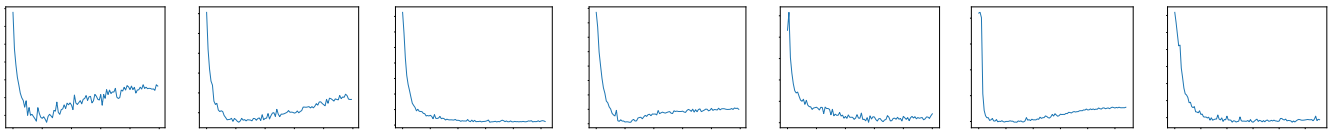
For both entire-plan and random-state, small teacher values will frequently occur in the data. This might be an issue as, for the effectiveness of heuristic search, what matters most are the states far from the goal – the heuristic function should yield crucial guidance there, while close to the goal usually search can easily find the solution anyhow. Motivated by this intuition, we experimented with a third strategy, aimed at emphasizing states with large goal distance during training: we selected only the initial state (s_1, c_1) of each plan. We call this the *init-state* selection strategy. This changes the teacher value distribution dramatically, with most of the weight near the maximum teacher value.

The three leftmost columns in Figure 4 (top) show the performance of the three sample selection strategies, each trained on the same number of training states. As the data shows, random-state is consistently better then entire-plan, clearly showing an advantage of training on uncorrelated states. The performance of init-state is mostly bad. A closer look at the data reveals that coverage tends to be higher on tasks where the training data contains small heuristic values, indicating that our hypothesis motivating this variant – a supposed advantage of emphasizing states far from the goal in the training data – is wrong.

The observed advantage of random-state over entire-plan should be weighed against the larger effort for data generation, resulting from the selection of only a single state per generated plan. The "random-state[#plans]" column in Figure 4 (top) shows performance for random-state when using the same number of *training plans* as entire-plan, i.e., to select a single training state from each training plan used by entire-plan. As we can see, the much reduced training data is insufficient to achieve top performance, so entire-plan is in the advantage when the computational resources for training data generation are limited. That said, performance remains at top level in Depots, Grid and Transport. So, for some domains, less training effort is needed. We will discuss this point in more detail in Section 7.

Next, we evaluate the impact of *weighting* and *pruning* samples in the training data. The motivation for weighting is the imbalanced nature of the sample sets produced by the three sample selection strategies. In all three strategies, some heuristic values may be underrepresented (in particular very high or very small ones), which might not match the data distribution during search. Weighting is a standard method to address this kind of phenomenon. During training, one multiplies the loss with a weight factor chosen individually per data point, in our case per heuristic value. The weight factor for heuristic value h is $1/N_h$ where N_h is the number of sample states with that

	sample selection strategy				pruning/weighting			
	init-state	random-state	entire-plan	random-state[#plans]	P^+W^+	P^+W^-	P^-W^+	P^-W^-
blocksworld	0.0	97.2	81.6	42.2	60.1	0.0	66.5	97.2
depots	28.6	76.2	71.2	78.6	45.2	76.7	39.8	76.2
grid	12.2	93.2	48.0	71.5	88.8	70.0	89.5	93.2
pipes-nt	90.6	89.6	75.8	63.8	83.9	87.0	89.9	89.6
scanalyzer	16.8	94.6	79.4	60.0	72.8	81.5	76.6	94.6
storage	13.5	95.5	52.0	21.0	24.5	45.0	24.5	95.5
transport	17.8	99.1	93.7	96.1	99.8	99.8	99.7	99.1
Average	25.6	92.2	71.7	61.9	67.8	65.7	69.5	92.2



(a) Blocksworld (b) Depots (c) Grid (d) Pipesworld (e) Scanalyzer (f) Storage (g) Transport

Figure 4. Top: Coverage (%) for different configurations of training. Left part: sample selection strategies on same amount of training states, init-state (selecting the initial state of every teacher-generated plan) vs. random-state (selecting a single random state per plan) vs. entire-plan (selecting all states along every plan). Middle Part: random-state restricted to the same number of plans as entire-plan. Right part: combinations of enabling (+)/disabling (-) pruning and weighting. Bottom: Exemplary learning curves, for one example task per domain. Loss on validation data plotted over the number of training epochs.

value, to the effect that every heuristic value has the same training impact.

Pruning includes each sample state only once in the training data. The motivation is that states may appear multiple times, especially for the entire-plan strategy where states close to the goal may occur very often. The network might memorize those states, affecting generalization.

Because the phenomena addressed by weighting and pruning are correlated, we evaluate all combinations of the two methods. The data is shown in the rightmost part of Figure 4 (top). The impact of the two methods varies greatly per domain, but overall they are clearly detrimental so our default is to switch them off.

To illustrate the learning process, Figure 4 (bottom) shows exemplary learning curves, i.e., error on validation data plotted over training epochs. We see that, in all domains, the neural networks achieve low error after few training iterations; afterwards, they tend to start overfitting. For most training runs, the best model (smallest validation loss) is found in less than 100 epochs. The training is stopped after at most 1000 epochs, 24 hours, or when the validation loss stops improving. Upon termination, we store the best model found.

7 Competitive Performance

We now evaluate our best configuration (i.e., the default setting) from the point of view of competitive performance. First, we compare to state-of-the-art (non-learning) planning systems. Then we examine the impact of using less training data, i.e., reducing the training overhead relative to non-learning systems. Finally, we reconsider the three domains where no competitive heuristic functions were learned, and highlights the reason for this lack of performance.

Comparison to Standard Planners

We are primarily interested in the comparison to the teacher search: *Can the NN we learn outperform the teacher?* As the teacher uses the h^{FF} heuristic, this is also a comparison to the most wide-spread standard technique. The answer to the question turns out to be "yes", though with weaknesses regarding heuristic evaluation speed.

Another question of interest of course is: *How do the NN heuristic functions we learn compare to the state of the art?* We address this by comparing to LAMA [27] and Mercury [14]. The answer turns out to be more negative for NN heuristic functions on their own, but a straightforward combination with h^{FF} preferred operators yields a planner competitive with the state of the art.

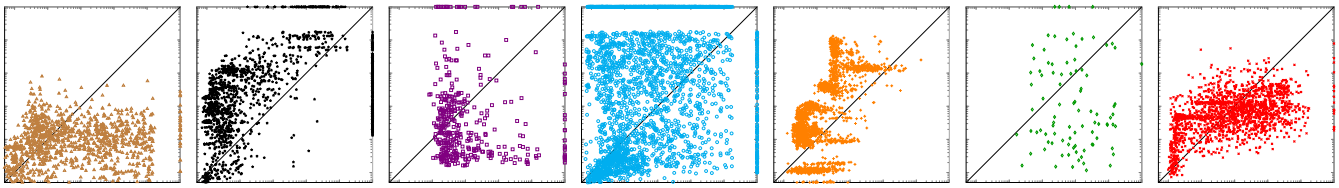
Figure 5 (top) shows the data. Consider first the comparison to the teacher search, GBFS with h^{FF}. In terms of overall coverage, our approach is comparable, winning in Pipesworld-NoTankage while losing in Depots, with small losses in the other domains. In terms of expansions, we do substantially better in 6 out of 7 domains, but that advantage is sometimes outweighed by the disadvantage in evaluation speed. In Depots, Pipesworld-NoTankage, and Scanalyzer we get better median runtime (the picture for arithmetic mean runtime is qualitatively similar).

Figure 5 (bottom) sheds more light on this with a per-instance view on runtime. Blocksworld, Storage and Transport favor h^{FF}, Depots and Scanalyzer favor NN heuristics. Pipesworld-NoTankage and Grid are extremely mixed (recall here though the much higher coverage of NN heuristics in Pipesworld-NoTankage). Certainly, competitive NN heuristics of strengths complementary to h^{FF} can be learned.

Note that other works using NN for search guidance [30, 31, 2] employ much more powerful resources for NN evaluation than a single CPU core. We use the latter here for the sake of fair comparison. But that neglects the parallelizability of NN evaluation, in particular the possible use of GPUs. As an initial exploration, we ran preliminary experiments using multiple cores. With 2 cores we get a speed-up of about 50%, with 4 cores of another 20%. It remains a question for future work to explore this direction systematically (see Section 9).

The comparison to LAMA and Mercury is more negative, though NN heuristics are still competitive in some regards. More importantly however, the advantages of LAMA and Mercury are mostly due to their optimizations beyond heuristic functions, in particular the use of preferred operators (and, for Mercury, of a short-cut technique recognizing relaxed plans that work in reality). The rightmost part of Figure 5 (top) adds preferred operators (generated by h^{FF}) to our

	coverage in %				median #expansions				median #exp. per second				median runtime				NN-POh^{FF}		
	NN	h^{FF}	LAM	Merc	NN	h^{FF}	LAM	Merc	NN	h^{FF}	LAM	Merc	NN	h^{FF}	LAM	Merc	cov	#exp	time
blocksworld	97.2	100	100	100	1381	12665	465	516	857	14536	19593	14875	1.7	0.8	0.1	0.8	98.8	442	0.9
depots	76.2	92.1	100	99.2	182	47731	8734	8746	530	1490	4792	3749	0.4	17.9	1.1	3.4	91	65	0.3
grid	93.2	93.2	100	100	493	2439	150	183	125	1810	4525	1739	4.1	1.5	0.2	4.9	94	493	3.2
pipes-nt	89.6	63.6	98.7	92.5	354	832	1676	138	295	752	3563	1640	1.2	1.5	0.6	2.7	97.9	103	0.8
scanalyzer	94.6	97.8	100	100	269	482	208	217	183	38	1403	725	1.4	5.7	0.7	8.3	98.2	97	1.1
storage	95.5	94.5	96	97	4208	2089	24712	16055	51	722	7309	5283	83.9	3	3.4	4.5	98	4145	39.5
transport	99.1	100	100	100	3002	4586	192	0	458	799	3415	0	8	6.3	0.2	2.8	100	122	1.1
Average	92.2	91.6	99.2	98.4	1413	10118	5162	3693	357	2940	6616	4116	14.4	5.2	0.9	3.9	96.8	781	6.7



(a) Blocksworld (b) Depots (c) Grid (d) Pipesworld (e) Scanalyzer (f) Storage (g) Transport

Figure 5. Top: Competitive performance. NN is our best NN heuristic configuration. h^{FF} is the teacher search (GBFS with h^{FF}, not using preferred operators). LAMA (LAM) and Mercury (Merc) included as a representation of the state of the art. NN-POh^{FF} is FD's dual-queue GBFS with the preferred operators generated by h^{FF} (with columns showing coverage, median #expansions, median runtime). Medians are calculated over the the commonly solved tasks. Bottom: Per-instance runtime, best NN heuristic configuration (x-axis) vs. GBFS with h^{FF} (y-axis); both axes log-scaled.

planner in the most straightforward fashion. In effect, the coverage gap almost closes; we tend to do significantly better in terms of expansions; and we outperform Mercury in runtime.

Training with Less Data

Any advantages vs. purely model-based techniques must be weighed against the training overhead incurred by NN heuristics. That overhead pays off, in our current setting, if many different initial states within the same state space will be encountered online. As we argued before, this is realistic in certain situations, e.g. if exogenous behavior may unpredictably change the system state. In such a scenario, an offline training overhead will eventually be amortized by a performance advantage on online initial states.

	coverage in %					median #expansions				
	100	75	50	25	2.5	100	75	50	25	2.5
blocks	97.2	96.2	94.2	87.8	45.3	409	265	230	364	93K
depots	76.2	89.3	90.6	90.8	82.8	181	78	94	150	883
grid	93.2	64.2	70.8	70.8	72.5	367	1005	1266	1302	1948
pipes	89.6	95.4	93.6	93.1	78.4	462	121	136	215	3501
scan	94.6	95.1	93.2	82.9	57.4	86	50	45	45	208
storage	95.5	18.5	59.5	32.5	1	626	222K	37K	228K	-
trans	99.1	99.8	99.0	97.7	94.4	2686	146	165	248	910
Avrg	92.2	79.8	85.8	79.4	61.7	688	32K	6K	33K	-

Figure 6. Performance of NN heuristics trained on only a fraction (100%, 75%, 50%, 25%, 2.5%,) of the available data; 100% included for comparison.

That said, it is of course a relevant question how much training effort is required to obtain good results: *How much data do we need?* Figure 6 shows performance when training our NN with less data. As we can see, performance is relatively robust in this parameter. In Transport, even 2.5% of the generated data suffice to obtain the same performance. The training overhead is dominated by the effort taken to generate the data (the actual learning process is around two orders of magnitude faster). So the above training-data fractions translate directly into fractions of the 400 hours allocated in our setup to training-data generation (cf. Section 3). To generate the 2.5% data sufficient for Transport, only 10 hours instead of 400 hours are required. To benefit from these observations, training data should be generated on demand per domain, stopping based on the development of either MSE or search performance.

Results on the 3 Unsuccessful Domains

Figure 7 shows performance data on the three dead-end free domains where no competitive heuristic functions were learned: NPuzzle, Rovers, VisitAll. As is immediately apparent from the data, not only have we the disadvantage of slow heuristic evaluation, but also informativity is mostly bad. In Rovers we have the compounded problem of search space size and per-node effort.

	coverage		#expansions		#exp. per sec.		runtime	
	NN	h^{FF}	NN	h^{FF}	NN	h^{FF}	NN	h^{FF}
npuzzle	0.0	97.3	–	–	–	–	–	–
rovers	53.0	73.9	3543	1165	43	1800	82.8	0.5
visitall	0.2	94.1	1065K	93K	893	35354	1251.5	2.6
Average	17.7	88.4	–	–	–	–	–	–

Figure 7. Performance comparison to GBFS with h^{FF} on the three dead-end free IPC domains where no competitive heuristic functions were learned. All parts except coverage show median values over commonly solved tasks.

In NPuzzle and VisitAll, the NN heuristics are extremely badly informed. An explanation for this is the extremely bad quality of the teacher plans (returned by GBFS with h^{FF}) in these domains, producing training states with excessively overestimated values. For exam-

	Reg	LR	RF	SVR_{10}	SVR_{100}
Reg	-	53	44	51	42
LR	5	-	13	21	11
RF	14	45	-	45	38
SVR_{10}	7	37	13	-	0
SVR_{100}	16	47	20	58	-

	Reg	LR	RF	SVR_{10}	SVR_{100}
Reg	-	18	18	18	8
LR	2	-	8	5	3
RF	2	12	-	5	4
SVR_{10}	2	15	15	-	0
SVR_{100}	12	17	16	20	-

Figure 8. Pairwise comparison of ML techniques. A figure entry row X column Y shows in how many task technique X obtains a better MSE than technique Y. Left part: comparison for the domains Blocksworld, Depots, Grid, Pipesworld-NoTankage, Scanalyzer, Storage, and Transport where useful heuristic functions are learned. Right part: comparison for NPuzzle, Rovers, and VisitAll where that is not the case. Data shown for regression networks (Reg), linear regressors (LR), random forest regressors (RF), and support vector regressors (SVR) with a C value of 10, or 100 and a radial basis function as kernel. Classification networks are not included here as the error they are trained on is incomparable (see text).

ple, in NPuzzle, some training states have teacher values of 2500 and more. This problem may be alleviated by using a different teacher, e.g. LAMA, a bounded suboptimal planner, or an optimal planner. Indeed, in preliminary experiments with weighted A^* as the teacher, coverage in one of the NPuzzle benchmark tasks increases from 0% to 40%. One should note that although an (bounded sub-)optimal teacher might lead to a NN with better heuristic estimates, it does not provide guarantees on the learned heuristic.

8 Comparison to Simpler ML Methods

We have seen that NN can learn useful heuristic functions. However, NN are large and complex structures that are expensive to train. The question arises whether that complexity is needed. Hence we now compare our NN heuristic function to simpler ML methods: linear regression, random forests [7], and support vector regression [15]. We train each technique once on each benchmark task. As support vector regression (using the *radial basis function* as kernel) is not effective in training data size (training time scales cubically in the number of samples), we use a relatively small fragment of our data. Specifically, we use 35,000 sample states as training data, and another 5,000 as test data. For NN, we use 5,000 of the 35,000 training data as validation data instead. With this data, the network training times are between the training times of support vector regression, with a C parameter of 10 and 100.[4]

We perform a pairwise comparison of techniques, counting for each comparison X vs. Y on how many tasks X obtains a smaller MSE on the test data than Y. Classification techniques are not included here as their error is incomparable to that of all the other techniques, where a single output represents the heuristic value. To see this, say that a training state s_i has cost $c_i = 100$ but has learned heuristic value 1. For all methods here except classification, the MSE results from the difference $100 - 1$. For both types of classification, however, one wrongly set output (0 instead of 1 or vice versa) suffice to obtain the wrong heuristic value.

Figure 8 shows the data. We see that regression NN are superior to all other techniques. For those 7 domains where competitive heuristics were learned (left-hand side of the figure), even the best competitor, SVRs with a C parameter of 100, is better on only a small fraction of our benchmark tasks. For the other 3 domains (right-hand side), the SVRs with a C parameter of 100 are only slightly better.

One may wonder whether the simpler ML techniques could compensate their worse informativity through increased evaluation speed, i.e., faster heuristic functions. However that is not so: except for linear regression, evaluation is at least one order of magnitude slower than our regression NN. Altogether, these results indicate that the complexity of NN is needed here to obtain good results.

[4] The C parameter penalizes misclassification of individual samples. High values misclassify fewer training samples, but shrink the margin between the hyperplane which shall separate the classes and the training samples.

9 Conclusion and Outlook

We propose to address NN heuristic function learning for classical planning from the ground up, and we have explored the simplest possible setup here. We have identified the behavior of relevant hyperparameters, and we have shown that, on a collection of domains, the state of the art can be reached and can be outperformed in terms of heuristic informedness and runtime.

This is merely a first step on the long road towards NN heuristic function learning in classical planning, but we believe it provides a solid starting point for further investigations.

One issue for future work is to reduce the training effort and enable learning on tasks infeasible for the teacher, for example by learning in iteratively growing distances backwards from the goal as done in some previous works [3, 4, 2]. Another issue is improving heuristic evaluation speed. One possibility could be to evaluate states in batches as done by Agostinelli et al. [2], leveraging ways to evaluate a NN on multiple inputs in parallel across multiple processors or even GPUs. An interesting avenue is to replace our output layers with action-choice classification, and use the resulting NN as search guidance in the style of preferred operators.

Ambitious and explorative tasks for future research remain in NN architecture design. To reduce NN size and, therewith, improve heuristic evaluation speed, an idea is to learn auto-encoders of states at the NN front end, transforming the state representation into a more compact learned representation. To facilitate learning in deeper NN, so-called residual blocks [16] might be useful, where the output of layer X provides additional input to layer X+2. This technique originates in image recognition, but is of a generic nature and was exploited also by the aforementioned work on Rubik's Cube [2].

The grand challenge, of course, is stronger generalization, up to entire domains which, given the generality of the "domain" concept, constitutes a veritable "General AI" challenge. We believe this should be approached step by step.

Generalizing over goals is straightforward in principle, simply by making them an additional input. However, in full generality this will massively increase the amount of training data required. So it makes sense to distinguish different levels of goal generalization, like a fixed number of possible goals, a fixed number of goal variables whose value may change, or a bound on variance in the distribution of goals.

The next step could be generalization over static predicates, e.g. road-map graphs, while keeping the object universe fixed. Observe that this in itself is a major challenge as it would encompass, and vastly surpass, the special case of navigation on arbitrary fixed-size

road maps. One idea in this context might be the use of graph convolution [23] at the input layers of the neural network.

Finally, domain generalization can draw on Toyer et al.'s [32] ideas for weight sharing over predicate and action-schema instances, or could potentially be done by structuring states into variable-dependency patterns where weight sharing is over patterns, or by encoding states like Sievers et al. [29] as graph structures and training graph convolutional networks on them. We believe that a meaningful categorization of domains into easier and harder classes would be useful to sub-structure these endeavors.

ACKNOWLEDGEMENTS

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

We thank Alan Fern and Scott Sanner for the insightful discussions.

REFERENCES

[1] Mart'ın 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 Man'e, 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 Vi'egas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[2] 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).

[3] Shahab J. Arfaee, Sandra Zilles, and Robert C. Holte, 'Bootstrap learning of heuristic functions', in *Proc. SoCS 2010*, pp. 52–60, (2010).

[4] Shahab J. Arfaee, Sandra Zilles, and Robert C. Holte, 'Learning heuristic functions for large state spaces', *AIJ*, **175**, 2075–2098, (2011).

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

[6] Blai Bonet and Héctor Geffner, 'Planning as heuristic search', *AIJ*, **129**(1), 5–33, (2001).

[7] Leo Breiman, 'Random forests', *Machine Learning*, **45**(1), 5–32, (2001).

[8] Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton, 'A survey of monte carlo tree search methods', *IEEE Transactions Computational Intelligence and AI in Games*, **4**(1), 1–43, (2012).

[9] Olivier Buffet and Douglas Aberdeen, 'FF + FPG: Guiding a policy-gradient planner', in *Proc. ICAPS 2007*, pp. 42–48, (2007).

[10] Jianlin Cheng, Zheng Wang, and Gianluca Pollastri, 'A neural network approach to ordinal regression', in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pp. 1279–1284, (June 2008).

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

[12] Dan C. Ciresan, Ueli Meier, and Jürgen Schmidhuber, 'Multi-column deep neural networks for image classification', in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3642–3649, (2012).

[13] Dan Claudiu Ciresan, Ueli Meier, Jonathan Masci, Luca Maria Gambardella, and Jürgen Schmidhuber, 'Flexible, high performance convolutional neural networks for image classification', in *Proc. IJCAI 2011*, pp. 1237–1242, (2011).

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

[15] Harris Drucker, Chris J. C. Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik, 'Support vector regression machines', in *Proceedings of the 9th International Conference on Neural Information Processing Systems*, pp. 155–161, (1996).

[16] 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)*, pp. 770–778, (June 2016).

[17] Malte Helmert, 'The Fast Downward planning system', *JAIR*, **26**, 191–246, (2006).

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

[19] 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).

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

[21] Murugeswari Issakkimuthu, Alan Fern, and Prasad Tadepalli, 'Training deep reactive policies for probabilistic planning problems', in *Proc. ICAPS 2018*, pp. 422–430, (2018).

[22] Diederik P. Kingma and Jimmy Ba, 'Adam: A method for stochastic optimization', in *Proc. ICLR 2015*, (2015).

[23] Thomas N. Kipf and Max Welling, 'Semi-supervised classification with graph convolutional networks', in *5th International Conference on Learning Representations (ICLR 2017)*, (2017).

[24] Donald E. Knuth and Ronald W. Moore, 'An analysis of alpha-beta pruning', *AIJ*, **6**(4), 293–326, (1975).

[25] Levente Kocsis and Csaba Szepesvári, 'Bandit based Monte-Carlo planning', in *Proc. ECML 2006*, pp. 282–293, (2006).

[26] Judea Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984.

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

[28] Jendrik Seipp, 'Better orders for saturated cost partitioning in optimal classical planning', in *Proc. SoCS 2017*, pp. 149–153, (2017).

[29] Silvan Sievers, Michael Katz, Shirin Sohrabi, Horst Samulowitz, and Patrick Ferber, 'Deep learning for cost-optimal planning: Task-dependent planner selection', in *Proc. AAAI 2019*, pp. 7715–7723, (2019).

[30] 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).

[31] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by self-play with a general reinforcement learning algorithm. arXiv:1712.01815v1 [cs.AI], 2017.

[32] Sam Toyer, Felipe Trevizan, Sylvie Thiébaux, and Lexing Xie, 'Action schema networks: Generalised policies with deep learning', in *Proc. AAAI 2018*, pp. 6294–6301, (2018).

[33] Jesus Virseda, Daniel Borrajo, and Vidal Alcázar, 'Learning heuristic functions for cost-based planning', in *Proceedings of ICAPS workshop on Planning and Learning*, (2013).