Optimal Search with Neural Networks: Challenges and Approaches

Tianhua Li,[1] Ruimin Chen,[1] Borislav Mavrin,[1] Nathan R. Sturtevant,[2] Doron Nadav,[3] Ariel Felner[3]

[1]University of Alberta, Canada
[2]Department of Computing Science, Alberta Machine Intelligence Institute (Amii), University of Alberta, Canada
[3]Ben Gurion University of the Negev, Israel
{tianhua, ruimin, mavrin, nathanst}@ualberta.ca, doron.nadav@gmail.com, felner@bgu.ac.il

Abstract

Work in machine learning has grown tremendously in the past years, but has had little to no impact on optimal search approaches. This paper looks at challenges in using deep learning as a part of optimal search, including what is feasible using current public frameworks, and what barriers exist for further adoption. The primary contribution of the paper is to show how to learn admissible heuristics through supervised learning from an existing heuristic. Several approaches are described, with the most successful approach being based on learning a heuristic as a classifier and then adjusting the quantile used with the classifier to ensure heuristic admissibility, which is required for optimal solutions. A secondary contribution is a description of the Batch A* algorithm, which can batch evaluations for more efficient use by the GPU. While ANNs can effectively learn heuristics that produce smaller search trees than alternate compression approaches, there still exists a time overhead when compared to efficient C++ implementations. This point of evaluation points out a challenge for future work.

Introduction

Admissible heuristics have rigid properties that must be maintained in order to be used with search algorithms such as A* (Hart, Nilsson, and Raphael 1968) and IDA* (Korf 1985). But, progress in building improved heuristics has slowed significantly. In 2014 Robert Holte challenged researchers to consider where future improvements to heuristics will arise (Holte 2014), citing the need for research on external memory heuristics, heuristic compression, and alternate representations of heuristics, such as neural networks. There have only been limited efforts to explore these ideas more deeply, and thus progress has been slow.

While deep learning has become immensely popular in many fields of Artificial Intelligence, it has not made significant headway into optimal heuristic search algorithms. This is primarily because the approximation inherent in most learning techniques conflicts with the need for admissibility (non-overestimation), which is required for A* search. While some older work (Samadi et al. 2008) was able to learn admissible heuristics, it had no learning guarantees and needed explicit hash tables to correct overestimated entries. Given the recent advances in deep learning, it is time to revisit the problem of using Artificial Neural Networks (ANNs) to learn admissible heuristics.

However, this requires mixing research in machine learning and heuristic search. Attempts from machine learning to do heuristic search have not always clearly met the standards of heuristic search research. For instance, Shah et al. (2020) describes a method to learn a heuristic with at most ϵ error, but in empirical discussions it is conceded that this is only with high probability - under the assumption that the heuristic function can be precisely learned. High probability is not good enough for A*, although it could work for other algorithms (Ernandes and Gori 2004; Stern, Felner, and Holte 2011; Lelis et al. 2016). Similarly, other recent work (Yonetani et al. 2021) proposed that a neural architecture could learn to do suboptimal pathfinding 'better' than A* and Weighted A*. However, timing results reveal that the approach is orders of magnitude slower than many of the optimal 2014 Grid-Based Pathfinding Competition solvers (Sturtevant et al. 2015).

Within this context, this paper provides advances to the understanding of learning truly admissible heuristics for optimal search by (1) describing several novel techniques that ensure that an ANN learns a heuristic without losing admissibility and (2) developing a novel variant of A*, BatchA*, that can be used with ANNs and still ensure optimality.

Our approach uses supervised learning of pattern database heuristics (Culberson and Schaeffer 1998), which are a common memory-based heuristic,and then use an ANN as a means of compression to reduce the storage required. Compression has been widely studied for memory-based heuristics (Felner et al. 2007; Ball and Holte 2008; Breyer and Korf 2010; Goldenberg et al. 2011), and is particularly important when the heuristics require memory that is larger than RAM (Döbbelin, Schütt, and Reinefeld 2013; Hu and Sturtevant 2019).

In this context, the primary methods proposed for learning admissible heuristics include (1) treating the problem as a classification problem instead of regression, (2) adjusting quantiles used for classification, and (3) using ensembles of neural networks. All of these are guaranteed to learn admissible heuristics, and in practice are able to learn effectively. Given a GPU-based heuristic, BatchA* is then able to use

the SIMD nature of GPUs to parallelize heuristic computations and significantly improve performance. BatchA* with our learned heuristics expands many fewer nodes than the DIV compression technique we compare to, while matching the time performance.

Background and Related Work

In *heuristic search*, the task is to find a path in a graph $\{\mathcal{G} = \{V, E\}, s, g, c, h\}$ from a start state, $s \in V$, to a goal state, $g \in V$, where $c : (e \in E) \rightarrow \mathbb{R}^+$ is a cost function for any edge or pair of states connected by and edge. A heuristic function $h(v)$ estimates the distance to the goal from state v. If $h^*(v)$ is the shortest distance to g, h is *admissible* if for all v, $h(v) \leq h^*(v)$ and h is *consistent* if $h(a) \leq c(a, b) + h(b)$ for every two states a and b. For large state spaces $\mathcal{G}$ is given implicitly, meaning that $\mathcal{G}$ can only be generated by expanding states and obtaining their neighbors. A* and IDA* are guaranteed to find optimal solutions if the heuristic is admissible, but A* is suboptimal with respect to expansions when the heuristic is inconsistent due to node re-expansions (Martelli 1977; Felner et al. 2011).

Pattern databases (PDBs) (Culberson and Schaeffer 1998) are table-based heuristics. PDBs are commonly used in exponential domains, where a domain abstraction, which will also be referred to as a *pattern*, is used to abstract the vertices in the original graph V into a smaller abstract state space $\phi(V)$. In the abstract graph if there is an edge between v_1 and v_2 then there is also an edge between $\phi(v_1)$ and $\phi(v_2)$. A breadth-first search is used to compute exact distances in $\phi(V)$, which are then stored in a table and used as heuristics in the original state space. For instance, if the original state space is a permutation of values (0 1 2 3 4 5), a domain abstraction abstracts away some values (such as 3–5), resulting in an abstract state of (0 1 2 * * *). In this example the number of vertices, V, is $6! = 720$, while the abstracted state space, $\phi(V)$, has $6!/3! = 120$ states. The abstracted items are called 'don't cares' and can take on any value, ensuring the heuristic is admissible. PDBs typically reduce the size of the state space exponentially with relatively small loss in heuristic accuracy over ground truth (Felner, Sturtevant, and Schaeffer 2009). During search, a state is abstracted, and then a ranking function (Myrvold and Ruskey 2001) is used to convert the abstract state into a unique integer, which indexes the state in a lookup table storing the distance to the abstract goal.

Numerous PDB enhancements have been developed. Two notable enhancements are additive PDBs (Felner, Korf, and Hanan 2004), which have smaller values in each PDB, but multiple PDBs can be added together while still guaranteeing an admissible heuristic. Another common approach with PDBs is to only store the delta between the computed PDB value and an inexpensive base heuristic. At runtime the original value can be restored by adding the stored delta to the base heuristic (Felner, Korf, and Hanan 2004; Sturtevant, Felner, and Helmert 2017). This reduces the total number of unique values in the PDB, which is important in our approach below.

PDB Compression

The most general means of compressing PDBs are based on the fact that a PDB is just a table of numbers. In the table, any method can be used to group entries. Then, grouped entries can be replaced by a single entry with their minimum value, compressing the table while ensuring admissibility. Common grouping methods include DIV, which groups k adjacent entries by dividing the index by k, or MOD, which uses the modulo operator to merge entries offset by $\frac{m}{k}$ in a PDB with m total entries (Felner et al. 2007). The effectiveness of these approaches depends on action dependencies in the state space (Helmert, Sturtevant, and Felner 2017).

There are many other approaches which have been suggested for PDB compression (Ball and Holte 2008; Breyer and Korf 2010; Edelkamp, Kissmann, and Torralba 2012; Sturtevant, Felner, and Helmert 2014). We do not describe all of these because our work here is focused on studying what can be done with recent deep-learning methods.

Relatively little work has been done on using ANNs (artificial neural networks) to learn *admissible* heuristics. An approach called ADP (for ANNs, decision trees, and partitioning) (Samadi et al. 2008) is the primary exception. ADP combines a diverse set of approaches to maintain admissibility. This includes a specialized loss function to penalize overestimates more than underestimates, a decision tree to subdivide states into smaller groups, and ANNs at the leaves of the decision tree which only train on subsets of the state space. Finally, any states that still return inadmissible heuristics are placed in a hash table. In summary, this is a highly engineered design which relies on a hash table to guarantee admissibility. That work also predates current advances in ANNs. By contrast, our work can always learn an admissible heuristic using a single ANN. In the conclusions we will discuss how the ideas in ADP could be used to help scale our work.

Neural Networks

ANNs are inspired by biological neurons. In the simplest case a fully-connected neuron is a composition of a linear function followed by a non-linear function, e.g. tanh or ReLU (Nair and Hinton 2010). Neurons are stacked into layers which allow them to learn different representations simultaneously. Typically, deep ANNs have multiple hidden layers which increases representation power. ANNs typically use fully connected layers, which are general and do not assume any specific structure of data. By contrast, convolutional layers use a specialized type of architecture that tries to exploit the structure of 2D data such as images or game boards. Such convolutional layer consists of multiple 2D filters called kernels. Each kernel learns a translation invariant feature by applying the kernel simultaneously to every $k \times k$ subset of the original image. In general, Convolutional Neural Networks (CNNs) consist of multiple convolutional layers followed by fully connected layers (Krizhevsky, Sutskever, and Hinton 2012). CNNs are a special class of ANNs.

The final layer of an ANN defines the learning task. In *regression* the last layer is fully connected to a single output

which is passed through an activation function to get a real-valued output, which is interpreted as a heuristic value. In *classification* the last fully connected layer has multiple outputs that are re-scaled into class probabilities by a softmax function. Each class is interpreted as a heuristic value.

Take the two sliding-tile puzzle states in Figure 1 as an example. A small ANN was used to learn a heuristic classifier for these states from a PDB heuristic, after which the original heuristic was discarded. To compute the heuristic for these states, they are evaluated on the classifier, softmax is applied to the last layer, and the output probability distribution is found in Table 1. For the first state ($\phi(v_1)$) the ANN predicts a heuristic value of 9 with probability 0.99, while for the second state ($\phi(v_2)$) it predicts a heuristic of 2 with probability 0.99.

ANNs have strong representational power. Theoretically, ANNs can approximate any Lebesgue integrable function defined on a compact set (Lu et al. 2017). Furthermore, Chollet (2019) draws parallels between ANNs and hashtables. However, increasing the approximating power of an ANN requires exponential growth of the number of neurons (Cybenko 1989). In practice, the size of ANN is bounded by GPU memory. A more practical approach to improve the approximating power of an ANN is to combine predictions of multiple ANNs, i.e. ensembling. In particular, boosting (Freund and Schapire 1997) is a well-known approach that creates ensembles by training each new model on the data points that were misclassified by previous ANNs.

Learning Admissible Heuristics

The aim of this paper is to use ANNs to learn an admissible heuristic h_{ANN} from an existing admissible heuristic h. This paper does this on top of PDB heuristics, although the approach can be applied to any admissible memory-based heuristic (such as merge-and-shrink heuristics (Helmert et al. 2014)). In our case, the input is a PDB heuristic and its associated pattern (a domain abstraction). The output is one or more ANNs that map states to admissible heuristic values. The goal is to produce ANNs with size smaller than the size of input PDB and heuristic values that are as large as possible. It is particularly challenging for the learned ANN heuristic to be both (1) admissible and (2) return the largest heuristic values possible. This is treated as a supervised learning problem; overfitting is not directly an issue, since the only goal is to reproduce the input data, not to generalize beyond the training input.

The complex design of ADP, described previously, results from the failure of a single ANN to learn an admissible heuristic. ADP uses several approaches to break the learning problem into smaller problems that can be learned independently. Our approach is to use a single architecture to learn admissible heuristics. We have done this on top of regression, classifiers, and ensembles of ANNs. The regression approach is based on re-scaling the output from a ReLU activation function to ensure admissibility. The regression approach did not work in practice, so details are provided elsewhere (Li 2022). The other two approaches of classifiers, and ensembles are described next.

Using Classifier Quantiles

Heuristic learning is not typically seen as a classification problem, but, for problems that are NP-complete, the state space will grow exponentially while the solution length grows polynomially, meaning that there are relatively few heuristic values when compared to states. Furthermore, when a heuristic is stored as a delta over an existing heuristic, there is a further reduction in the number of heuristic values needed for a problem. For instance, in problems on the 4x4 sliding tile puzzle we will use a PDB heuristic that initially has 35 distinct values. However, by subtracting the Manhattan Distance heuristic (a memory-free heuristic which is easy to compute) from the PDB heuristic, this is reduced to only 9 values.

Given this, we approach the heuristic learning problem as a classification problem, with the unique property that our classes are ordered. This general problem of ordered classes has been studied previously (Cheng, Wang, and Pollastri 2008), but we use a different approach here to obtain our results, as variants of that work were not effective.

As described previously, after learning a classifier, the ANN can be evaluated on a state to produce an output vector of predictions for each possible class. This vector is passed through a soft-max function, which is then interpreted as a probability distribution on each class. The final classification returned is typically the class with the largest probability. But, in the case where we want to return an admissible heuristic, this might not be the best approach. In particular if the probability mass was split evenly between two classes, it might be necessary to return the class with minimum heuristic value to maintain admissibility. Other more general approaches are possible.

We illustrate the question of how to interpret the classifier probability distribution in Table 1, which shows the classifier output for states $\phi(v_1)$ and $\phi(v_2)$ from Figure 1. The largest output probability for $\phi(v_1)$ is on class 9, which would typically be used as the class (heuristic) prediction. But, because the classes are ordered we can use other metrics. For instance, we can return the first class that has cumulatively (for all lower classes) at least $1\text{e}-4$ of the output probability. For $\phi(v_1)$ this would change the output class to 8, while for $\phi(v_2)$ the output would still be 2.

As such, we can tune the prediction to be more or less aggressive by tuning the quantile of the probability mass used for computing the predicted class for a given state. Using a quantile of 1.0 will return the maximum class that received any probability mass. This maximizes the heuristic prediction, but is unlikely to produce an admissible heuristic. Using a smaller quantile will return a smaller heuristic, at the cost of heuristic accuracy, as some states that originally had correct heuristic values will then produce underestimates. Using a quantile of 0 will return a heuristic of 0, which is admissible.

More precisely, assume that a classifier $h_{\hat{\text{C}}}$ has been learned through training on the input heuristic h that has maximum value h_{max}. Let $P_{h_{\hat{\text{C}}}}(v, i)$ be the classifier probability of the ith class on state v. Then, let $C_{h_{\hat{\text{C}}}}(v, q)$ return the class (heuristic) that cumulatively, from small to

heurisitics	0	1	2	3	4	5	6	7	8	9
v_1	0	0	0	0	5.56e-43	1.09e-32	3.98-24	2.07e-18	1.03e-04	0.99
v_2	3.86e-33	1.63e-18	0.99	1.30e-03	5.50e-11	2.94e-21	3.24e-42	0	0	0

Table 1: Classifier output probabilities for states $\phi(v_1)$ and $\phi(v_2)$ of Figure 1

large classes, has at least probability q. That is $C_{h_{\hat{C}}}(v,q) = \arg\min_i(\sum_{j=0}^{i} P_{h_{\hat{C}}}(v,j)) \geq q$.

We can use $C_{h_{\hat{C}}}(v,q)$ to build a new heuristic $h_{\mathrm{QNT}}(v)$ which is guaranteed to be admissible. This is done in two steps. First, we compute the maximum quantile q_v which guarantees an admissible heuristic for state v. For each $v \in \phi(V)$ let $q_v = \mathrm{argmax}_{q \in 0\ldots1} C_{h_{\hat{C}}}(v,q) \leq h(v)$. Then, we let q^* be the quantile that guarantees an admissible heuristic for all states; $q^* = \min_{v \in \phi(V)} q_v$. Putting this together, $h_{\mathrm{QNT}}(v) = C_{h_{\hat{C}}}(v,q^*)$. This is the largest heuristic that can be returned, using this approach, that is guaranteed to be admissible on every state.

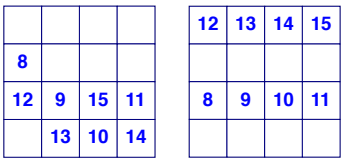
One might fear that q^* would be too small to be useful in practice, but even small values of q^* can be effective. Consider again the states in Figure 1. The PDB values of states $\phi(v_1)$ and $\phi(v_2)$ are 9 and 1 respectively after subtracting Manhattan Distance (MD). The output of $P_{h_{\hat{C}}}$ for each class is given in Table 1. If we selected $q^* = 0.5$, $h_{\mathrm{QNT}}(\phi(v_1)) = 9$ because $\sum_{j=0}^{8} P_{h_{\hat{C}}}(v,j)) \approx 0.01 < 0.5$ and $\sum_{j=0}^{9} P_{h_{\hat{C}}}(v,j)) = 1 \geq 0.5$. However, $h_{\mathrm{C}}(\phi(v_2)) = 2$ for $q^* = 0.5$, which is inadmissible.

Using instead $q^* = 1.63\mathrm{e}{-18}$ results in $h_{\mathrm{C}}(\phi(v_2)) = 1$ which is admissible. $h_{\mathrm{QNT}}(\phi(v_2))$ for $q^* = 1.63\mathrm{e}{-18}$ is then 7 instead of 9, causing a small loss, but the heuristic is admissible. Although the quantile margins are small, the computations are deterministic, so there is no danger of losing admissibility.

There must exist some q^* such that h_{QNT} is admissible. This is because $q^* = 0$ results in a heuristic of 0 for every state, which is admissible. Thus, it follows that some $q^* \geq 0$ will result in h_{QNT} being admissible.

Ensemble of Neural Networks

Our second method that preserves admissibility is to use an ensemble of ANNs, where the returned heuristic is the minimum of the heuristic returned by each ANN in the ensemble. Formally, we build a set $H_k = \{h_0, h_1, \ldots, h_k\}$ of heuristics which are combined for the ensemble heuristic:



$\phi(v_1)$ $\phi(v_2)$

Figure 1: Abstracted states $\phi(v_1)$ and $\phi(v_2)$ for the 4x4 Sliding Tile Puzzle

$$h_{\mathrm{ENS}(H_k)}(s) = \min_{i \in 0\ldots k} h_i(s)$$

While the set H could be composed of any set of heuristics, we will build them by training successive ANNs on states that are inadmissible in the current heuristic. In particular, given H_t after t training steps, H_{t+1} is built by training h_{i+1} on the states $S = \{s \in \phi(V) \mid h_{\mathrm{ENS}(H_t)}(s) > h(s)\}$. The overall procedure is started by training h_0 on h.

Lemma 1. *Adding h_{i+1} to a set of heuristics H_i (creating set H_{i+1}) cannot increase the number of states with inadmissible heuristics in $h_{\mathrm{ENS}(H_i)}$.*

Proof. The heuristic of any state s computed by $h_{\mathrm{ENS}(H_i))}$ is the minimum of the heuristics $h \in H_i$. Thus, adding an additional heuristic can only decrease the heuristic value for any state s, not increase it. □

Given this, we can now show that this process can learn an admissible heuristic.

Theorem 1. *Assuming that (1) ANN heuristics are deterministic and (2) that an ANN can also be trained to learn at least one state in the training set, then there exists some value k for which $h_{\mathrm{ENS}(H_k)}$ will be admissible.*

Proof. Since each new ANN heuristic added to the ensemble makes at least one new state admissible and no new states inadmissible, the set H will only require a finite number of heuristics before $h_{\mathrm{ENS}(H)}$ is admissible. □

As with the quantile method, the ensemble method is guaranteed to produce an admissible heuristic. In our experiments two ANNs were sufficient in an ensemble to achieve admissibility, even on a heuristic with $|\phi(V)| = 5 \times 10^8$ states.

Improved Training

The approaches described above are sufficient to preserve admissibility in a learned heuristic. We next discusses ways to further improve the quality of heuristics that are learned.

The ensemble approach focuses on overestimated entries to achieve 100% admissible rate. Underestimated entries do not affect admissibility, nonetheless, they do affect the quality of heuristics. Because the second ensemble heuristic is only trained on a subset of the full data, the ANN will be forced to generalize to states that are unseen during training. If many of the states in the second ANN training set in the ensemble have low heuristic values, it is likely that these will be generalized to other states, lowering the overall heuristic.

One way to improve the quality of heuristics is to include part or all of the states that have already been learned into the dataset that a succeeding ANN will be trained on. Because they already have had their heuristic learned admissibly in the first ANN, the underestimated entries can be be given

Algorithm 1: Batch A*

```
1: Input: s, g
2: OPEN ← s
3: WAIT ← ∅
4: limit ← f(s)
5: while OPEN or WAIT is not empty do
6:     if OPEN is empty or OPEN.top.f > limit then
7:         Eval(WAIT)
8:     end if
9:     n ← OPEN.pop()
10:    limit ← max(n.f, limit)
11:    for each child c of n do
12:        Add c to WAIT
13:        if WAIT is full then
14:            Eval(WAIT)
15:        end if
16:    end for
17: end while
```

the maximum heuristic value to ensure they are not further underestimated. If the succeeding ANNs together with preceding ones are sufficient to preserve admissibility, adding underestimated entries will improve the average heuristic value predicted by ANNs. In all of our experiments with ensembles we measure the number of underestimated entries i and randomly add $10i$ admissible entries to the training set with the maximum heuristic value. This significantly improves the performance of the ensemble learning. For instance, when training on the 8-15 PDB for the sliding tile puzzle this increases the average heuristic by 50%.

A second way to improve the heuristic is to combine the quantile and ensemble methods, creating a new heuristic $h_{\mathrm{Q+E}}$. After training H_0 in an ensemble, we can choose some $q > q^*$ to decrease the number of overestimated entries, while still leaving some entries overestimated. This reduces the training set required for the next ANN in the ensemble and simplifying the learning problem. For this method we sample a small set of values ($q \in \{0.1, 0.2, 0.3\}$) that are used for the first heuristic in the ensemble.

Batch A*

GPUs are SIMD processors that can perform operations in parallel on separate data. Thus it is, in theory, no slower to evaluate 1 ANN heuristic on a GPU than 10 heuristics. BatchA* is a variant of A* designed to exploit this efficiency. We provide a short description of BatchA* and sketch of correctness here; more details and similar extensions applied to IDA* are described elsewhere (Nadav 2021). But, it is worth noting that BatchA* is a general algorithm that can be used for other problems besides ANN-based heuristics. For instance, the SIMD capabilities of a CPU could be also exploited to simultaneously compute the rank of several states simultaneously for PDB lookups using Batch A*.

BatchA* is identical to A* except that it places states into a queue when they are generated. When the queue is full or the queue must be evaluated to ensure correctness, all the states are evaluated and added to the OPEN list. The psuedocode for the most important details of BatchA* are found in Algorithm 1. We assume that the environment/heuristic are provided elsewhere, and the only input is the start (s) and goal (g) states. BatchA* maintains a *limit*, which represents the maximum f-cost that has been expanded. States waiting to have their heuristics batch evaluated are on a WAIT list. Before increasing the f-cost limit, BatchA* must ensure that there are no states on WAIT waiting to be evaluated. The *Eval* method evaluates the heuristic of all states in parallel before adding them to the appropriate data structure (open/closed). As long as there are many states with the same f-cost, we can expect that the WAIT list will primarily be evaluated when it is filled up (line 13). Additional optimizations are possible, such as checking to see if a state has already had its heuristic evaluated before putting it on the WAIT list, or checking if a state is already on WAIT instead of putting it on a second time.

We first address the behavior of BatchA* with a consistent heuristic, where f-costs monotonically increase. In this case we must only show that BatchA* does not expand states with larger f-cost before expanding all states with smaller f-cost, or terminate without finding a path that exists. There are only two places where these can occur. The first case is when the OPEN list is empty. In this case there may be states on the WAIT list, so any waiting states must be evaluated (line 7). The second case is when the minimum f-cost on OPEN increases above the previous minimum (line 7). At this point there could be states on the WAIT list with lower f-cost that need to be expanded first, so any waiting states must first be evaluated.

With an inconsistent heuristic f-costs can decrease if heuristics are not propagated (Felner et al. 2011). In this case BatchA* must re-expand states when shorter paths are found. When this happens, BatchA* does not need to lower the *limit* being used (line 10), however a complete discussion of BatchA* and inconsistency is beyond the scope of this paper.

Batch A* Expansions

Although BatchA* performs well in our experiments, we analyze why BatchA* may do significantly more expansions than A* in some cases. In particular, the efficiency of Batch A* depends on how many states can have their heuristics evaluated in parallel.

One might think that if A* were to expand k states, that Batch A* with a batch size of b would expand no more than $k + b$ states, meaning that Batch A* might just expand a single batch of extra states to finish the search. However, this analysis is incorrect. If the goal has f-cost of C^*, Batch A* may be forced to expand all states with $f = C^*$ before expanding the goal, where A* may only expand the states on the optimal path.

This occurs when the goal is at the end of a long path of states all with $f = C^*$. Because Batch A* can only expand one of the states in this path at a time, it may, as a result, end up expanding all other states with $f = C^*$ while trying to get to the end of this chain. If the chain is the only set of

states with $f = C^*$, then BatchA* will no longer be able to batch lookups, and will evaluate a single state at a time.

Related Algorithms

Batch A* is similar to Batch Weighted A* (BWAS) (Agostinelli et al. 2019), which also batches states for expansions. However, the BWAS implementation (1) does not limit the f-cost of states expanded in each batch and (2) does a goal check on each expanded state. Thus, it could find a solution which does not meet the suboptimality bound.

In general this can be prevented either by limiting the f-cost of states in each batch (as in Batch A*), or by allowing expansions with larger f-cost, but ensuring the search still proves the (bounded) optimality of the solution. BWAS was developed in the context of an inadmissible heuristic; in that context this distinction is less important.

Experiments

Using ANNs to learn admissible heuristics is significant on its own. But, as we will show, our approaches are not only able to learn admissible heuristics, they are also able to learn more accurate heuristics than the standard baseline compression techniques. After describing our experimental setup, we provide our primary learning results. We then explore the ability to efficiently use these heuristics during search, pointing out key differences in heuristic search and machine learning implementations and evaluation.

Note that we limit our experiments here to compressing heuristics which fit in the memory of all modern devices, because the emphasis in this paper is how to learn admissible heuristics. In the conclusions we provide a detailed look at the research required to scale this work to heuristics that are significantly larger.

Experimental Setup

While many ANN approaches require significant training time on many parallel GPUs or TPUs, our approach used relatively modest resources. We used CNNs and train all the CNN models sequentially with Pytorch on CUDA 10.1 using one 2080ti GPU. Our primary experiments are conducted on the 4x4 Sliding Tile Puzzle (STP), 5x5 STP, and 16-tile TopSpin. Each of these domains can be represented as a permutation of numbers. The STP is a grid of numbers that must be sorted while the TopSpin puzzle is a continuous loop of numbers that must be sorted by rotating 4 adjacent tiles. PDBs were built using HOG2[1].

The baseline PDB heuristic for the 4x4 STP used additive PDBs (Felner, Korf, and Hanan 2004) of tiles 1-7 and tiles 8-15 taken as a delta over Manhattan distance (MD). In the 5x5 STP we use four 6-tile additive PDBs as a delta over MD. In TopSpin we use a 0-7 tile non-additive PDB. The sizes and average heuristic values (delta over Manhattan Distance for the STP domain) are found in the h_{PDB} columns in Table 2.

We use CNNs to learn admissible heuristics using domain abstractions as follows. The 1-7 and 8-15 4x4 STP PDBs are represented as 7- and 8-channel 4x4 binary images. The 6-tile patterns in the 5x5 STP are represented as 6-channel 5x5 binary images. A 3x3 convolution (which uses the same weights to evaluate all 3x3 sub-images) is used for both of these domains. The PDB pattern in TopSpin includes tiles 0-7 and is represented as a 8-channel 16 dimensional vector. Because the topology of TopSpin is a continuous loop, the input is two copies of the puzzle appended to each other, which simulates a full loop. In this domain a 1x8 convolution is applied to all combinations of 8 adjacent tiles.

The size of a learned heuristic can be tuned by changing the size and number of layers in the CNN. For example, in one 3.2 MB CNN used to learn the h_{QNT} 8-15 PDB in 4x4 STP, there is one convolutional layer followed by two fully connected layers and one output layer. The number of input and output channels are 8 and 32, respectively. The number of input and output features in the first fully connected layer are 512. The number of input and output features in the second fully connected layer are 512 and 1024, respectively. The number of input and output feature in the output layer are 1024 and 10, respectively. Multiplying the number of weights in each fully connected layer gives approximately 800k weights. When each uses a 4-byte float, 3.2 MB are required for the full network. When building ensemble heuristics, our ensemble ANNs are half the size of our quantile ANNs because we expect to store two of them in memory. The parameters used for all architectures in Table 2 are found in Table 4.

We use a cross-entropy loss function, Adam as the optimizer, softmax as the activation function in the last layer, and ReLU as the activation function in other layers for all CNNs. Abstract states with a heuristic value of i are labeled as the i-the class in TopSpin. In STP, deltas over MD are all even, so heuristic values of i were mapped to the $\frac{i}{2}$-th class and the $\frac{i}{2}$-th class in STP since there are no odd heuristics.

We summarize our results using average heuristic values of all states in the heuristic. While the average value cannot be used as a general predictor for performance (see, for example, (Clausecker and Schintke 2021)), the PDBs used here are well-understood and we have validated that the average heuristic does predict performance. Repeated training runs have reproduced these results with very small variance.

Note that the closest approach to ours is ADP (Samadi et al. 2008), which we cannot directly compare against. Without a benchmark implementation, there are too many details that are not described in the paper to be able to do a complete re-implementation of the approach. We did also experiment with regression-based learning approaches. The classification approaches described here have far superior performance in practice (Li 2022).

Learning Results

Results for our new approaches are found in Table 2. A parallel table giving precise details on the architecture used for each of these experiments is in Table 4. All learned heuristics are admissible. The table shows the memory used by different heuristics (left) and the average heuristic value (right) for each approach. We used a uniform training and compression factor of 100 for all experiments except one where we also

[1] https://github.com/nathansttt/hog2/tree/PDB-refactor

		Heuristic Size (MB)					Average Heuristic Value				
Domain	PDB Pattern	h_{PDB}	h_{DIV}	h_{QNT}	h_{ENS}	$h_{\mathrm{Q+E}}$	h_{PDB}	h_{DIV}	h_{QNT}	h_{ENS}	$h_{\mathrm{Q+E}}$
5x5 STP	1, 5-6, 10-12	127.51	1.27	1.27	1.27	1.27	2.0773	1.7219	**1.9154**	1.5600	1.0772
	2-4, 7-9						0.9639	0.3050	**0.7353**	0.4075	0.3767
	13-14, 18-19, 23-24						0.9639	0.2673	**0.7856**	0.4379	0.3751
	15-17, 20-22						0.9639	0.3361	**0.7972**	0.4237	0.3275
4x4 STP	1-7	57.66	0.58	0.57	0.54	0.62	3.9122	2.0825	1.7868	**2.8442**	2.4562
	8-15	518.92	5.19	5.12	5.12	6.33	3.9728	1.6521	2.3362	1.8535	**2.5094**
TopSpin	0-7	518.92	5.18	5.18	5.18	5.18	9.1350	**6.9862**	6.8593	6.0478	5.8854
TopSpin	0-7	518.92	0.29	0.20	0.26	0.23	9.1350	5.6442	4.6089	**5.7716**	5.5841

Table 2: Summary results comparing all techniques.

show a compression factor of 1000[2].

We compare the original PDB heuristic (h_{PDB}) with DIV compression (h_{DIV}), quantile heuristics (h_{QNT}), ensemble heuristics (h_{ENS}) with two ANNs, and the combination of both quantile and ensemble methods ($h_{\mathrm{Q+E}}$). Because we perform supervised learning on the PDB heuristics, h_{PDB} is the maximum learnable heuristic. The best compression results are in bold.

We will look at this data row by row, beginning with the 5x5 STP domain. While 5x5 STP state space is larger overall, the most common PDB heuristics used in this domain are 127MB in size, smaller than the largest 4x4 PDB heuristic. This heuristic was easily learnable by both h_{QNT} and h_{ENS} for all heuristic patterns. $h_{\mathrm{Q+E}}$ did not work well because the first heuristic in the ensemble was already too strong; there was no need to lower the quantile to reduce the number of states used for the second ensemble ANN.

In the 4x4 STP domain the 1-7 PDB is 58 MB (meaning we train on 58 million states). h_{QNT} was not able to outperform h_{DIV} on its own, giving an average heuristic of 1.7868 vs 2.0825 for h_{DIV}. However, h_{ENS} was able to return an average heuristic of 2.8422, significantly better than h_{DIV}. On the 519MB 8-15 PDB $h_{\mathrm{Q+E}}$ had the best performance with an average heuristic of 2.5094, significantly better than 1.6521 for h_{DIV}.

The TopSpin PDB is the same size as the 4x4 STP 8-15 PDB with 518 million entries. Here h_{DIV} has relatively strong performance given a compression factor of 100, and is able to outperform h_{QNT} and h_{ENS}. We also report the results for a compression factor of 1000 because we discovered that, as we used larger compression factors, the performance of the ANNs grew relative to h_{DIV} and h_{ENS} was able to outperform h_{DIV}.

To summarize, we see that ANNs can be used successfully for compression PDB heuristics. In the case of h_{QNT}, we are using a *single* ANN to learn the admissible heuristic. This is a significant achievement, because previous approaches that ensured admissibility used much more complex approaches to achieve this. Overall, the techniques we introduce are able to produce larger heuristic values than h_{DIV} with the same size of memory in every domain. As these are a first result in this direction, it would not be surprising if more advanced learning approaches (Vaswani et al. 2020) were able to improve the overall training process, and post-learning optimization of the ANN could provide further compression.

[2]During parameter tuning we were often able to achieve even better performance for a particular instance with non-uniform parameters.

Average over benchmark Korf instances				
Heuristic	b	Time(s)	Expanded	Generated
C++ PyTorch				
h_{ENS}	1	183.2	102,310	313,639
h_{ENS}	10	30.0	102,429	314,016
h_{ENS}	100	13.7	103,157	316,310
h_{ENS}	1000	12.7	109,886	337,405
C++ PyTorch with CUDA				
h_{ENS}	1	175.6	102,310	313,639
h_{ENS}	10	21.1	102,429	314,016
h_{ENS}	100	4.5	103,157	316,310
h_{ENS}	1000	2.9	109,886	337,405
CPU				
h_{DIV}	-	3	856,749	2,591,844
h_{PDB}	-	0.03	12,325	38,856

Table 3: BatchA* with different batch sizes.

Heuristics in Search

While it is important to be able to learn admissible heuristics, these heuristics must be usable in practice. To provide a fair comparison against non-learning algorithms, all of our implementations are in C++. The learning code uses PyTorch's C++ bindings for the ANN heuristic lookups.

In these experiments we compare h_{PDB}, h_{DIV}, and h_{ENS} heuristics in the 4x4 sliding tile puzzle. The first h_{ENS} heuristic is learned from the delta of PDB 1-7 and Manhattan distance, and the other is learned from the delta of PDB 8-15 and Manhattan distance. The two h_{ENS} heuristics are then added together with Manhattan Distance to produce a final admissible heuristic. We experiment on the 100 standard test instances (Korf 1985), reporting the average time, nodes expanded, and nodes generated on these problems.

The results are in Table 3. The first line is essentially a standard A* implementation. While only 100k nodes are expanded, on average, for a given problem, the PyTorch library is a bottleneck to performance and is only able to perform 500 heuristic evaluations per second. (The entire heuristic

Domain	PDB Pattern	Model	Convolutional Layer		Linear Layer 1st		2nd		3rd		4th		5th	
			In	Out	In	Out	In	Out	In	Out	In	Out	In	Out
4x4 STP	1-7	QNT	7	32	128	312	312	312	312	8		-		
	8-15		8	32	128	256	256	512	512	1024	1024	568	568	10
	1-7	ENS & Q+E	7	32	128	164	164	256	256	8		-		
	8-15		8	32	128	512	512	854	854	10		-		
5x5 STP	All	QNT	6	32	288	396	396	496	496	6				
	All	ENS & Q+E	6	32	288	288	288	248	248	6				
TS	0-7	QNT	8	32	488	488	488	512	512	836	836	512	512	13
		ENS & Q+E	8	32	488	488	488	512	512	408	408	13	-	

Table 4: Model Architectures

requires evaluating the 4 ANNs in the ensemble.)

BatchA* with batch size $b = 1$ is just A*. The next three lines in the table show the impact of batching heuristic lookups. With $b = 1000$ BatchA* is 14x faster than with $b = 1$, and the number of nodes expanded or generated does not increase significantly. When compared to h_{DIV} of a comparably sized PDB, BatchA* does 8x fewer node expansions. But, it is still 4 times slower than A* with h_{DIV}. As a result, we used the CUDA extensions for PyTorch to further improve performance. With $b = 1$ this does not significantly speedup the implementation. However, with $b = 1000$ the CUDA implementation is 63 times faster than the base PyTorch implementation with $b = 1$. At this point the h_{DIV} and BatchA* results are the same speed, although the BatchA* implementation expands 8x fewer nodes. On a unified memory process, such as Apple's recent M1 or M2 chips, the cost of sending data across the bus to the GPU is entirely eliminated. Thus, we would expect to achieve further performance improvements with further hardware investments.

We note that we can see the impact of the batch size on the number of node expansions in BatchA*. When the batch size is larger, BatchA* expands 7% more states than A* would, for the reasons described previously. However, this overhead in expansions is more than offset by the speed gains resulting from the batch processing.

We have performed the same evaluations on the other 4x4 STP heuristics, achieving the fastest results with h_{ENS}. We are continuing to optimize and explore the h_{QNT} and $h_{\mathrm{Q+E}}$ implementations before publishing full results of this comparison.

Future Work and Broader Applications

This paper shows how to combine classifiers, quantile methods, and ensemble methods to learn strong admissible heuristics for optimal heuristic search.

One important challenge in future work will be to scale our implementation to learn heuristics which are far larger than main memory, such as the 2.6TiB Rubik's Cube PDB (Hu and Sturtevant 2019) or the 1.4TiB 5x5 8-tile PDBs (Döbbelin, Schütt, and Reinefeld 2013), which are more than 3000x larger than the heuristics learned in this paper. This scaling requires significant research which is beyond the scope of this paper. Questions to be addressed in this scaling include: (1) should we still learn a single ANN or several independent ANNs, (2) the impact of this choice on the speed of training, (3) the impact of this choice on the runtime performance of BatchA*, (4) whether we should train on all heuristic entries, or on a subset of the data hoping for generalization, and, if we attempt to learn independent ANNs, (5) how should the data be divided for the learning process.

Acknowledgements

This work was funded by the Canada CIFAR AI Chairs Program. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). This research was sponsored by the United States-Israel Binational Science Foundation (BSF) under grant number 2017692 and by Israel Science Foundation (ISF) under grant numbers 844/17 and 210/17.

References

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(8): 356–363.

Ball, M.; and Holte, R. C. 2008. The Compression Power of Symbolic Pattern Databases. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2–11.

Breyer, T. M.; and Korf, R. 2010. 1.6-bit pattern databases. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 39–44.

Cheng, J.; Wang, Z.; and Pollastri, G. 2008. A neural network approach to ordinal regression. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, 1279–1284.

Chollet, F. 2019. The Measure of Intelligence. *arXiv preprint arXiv:1911.01547*.

Clausecker, R. K.; and Schintke, F. 2021. A Measure of Quality for IDA* Heuristics. In *Proceedings of the International Symposium on Combinatorial Search*, volume 12, 55–63.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern Databases. *Computational Intelligence*, 14(3): 318–334.

Cybenko, G. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4): 303–314.

Döbbelin, R.; Schütt, T.; and Reinefeld, A. 2013. Building Large Compressed PDBs for the Sliding Tile Puzzle. In *IJCAI Workshop on Computer Games*, 16–27.

Edelkamp, S.; Kissmann, P.; and Torralba, A. 2012. Symbolic A* Search with Pattern Databases and the Merge-and-Shrink Abstraction. In *European Conference on Artificial Intelligence*, 306–311.

Ernandes, M.; and Gori, M. 2004. Likely-admissible and sub-symbolic heuristics. In *European Conference on Artificial Intelligence*, volume 16, 613. Citeseer.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22: 279–318.

Felner, A.; Korf, R. E.; Meshulam, R.; and Holte, R. C. 2007. Compressed pattern databases. *Journal of Artificial Intelligence Research*, 30: 213–247.

Felner, A.; Sturtevant, N.; and Schaeffer, J. 2009. Abstraction-based heuristics with true distance computations. *Symposium on Abstraction, Reformulation and Approximation*, 9.

Felner, A.; Zahavi, U.; Holte, R.; Schaeffer, J.; Sturtevant, N.; and Zhang, Z. 2011. Inconsistent heuristics in theory and practice. *Artificial Intelligence (AIJ)*, 175(9-10): 1570–1603.

Freund, Y.; and Schapire, R. E. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1): 119–139.

Goldenberg, M.; Sturtevant, N. R.; Felner, A.; and Schaeffer, J. 2011. The Compressed Differential Heuristic. In *AAAI Conference on Artificial Intelligence*, 24–29.

Hart, P.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4: 100–107.

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

Helmert, M.; Sturtevant, N. R.; and Felner, A. 2017. On Variable Dependencies and Compressed Pattern Databases. *Symposium on Combinatorial Search (SoCS)*, 129–133.

Holte, R. C. 2014. Korf's Conjecture and the Future of Abstraction-based Heuristics. In *Heuristics and Search for Domain Independent Planning Workshop at International Conference on Automated Planning and Scheduling (ICAPS)*.

Hu, S.; and Sturtevant, N. R. 2019. Direction-Optimizing Breadth-First Search with External Memory Storage. *International Joint Conference on Artificial Intelligence (IJCAI)*, 1258–1264.

Korf, R. E. 1985. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27(1): 97–109.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 1097–1105.

Lelis, L.; Valenzano, R.; Nazar, G.; and Stern, R. 2016. Searching with a corrupted heuristic. In *International Symposium on Combinatorial Search*, volume 7, 63–71.

Li, T. 2022. *Learning Admissible Heuristics with Neural Networks*. Master's thesis, University of Alberta.

Lu, Z.; Pu, H.; Wang, F.; Hu, Z.; and Wang, L. 2017. The expressive power of neural networks: A view from the width. In *Advances in neural information processing systems*, 6231–6239.

Martelli, A. 1977. On the Complexity of Admissible Search Algorithms. *Artificial Intelligence*, 8(1): 1–13.

Myrvold, W.; and Ruskey, F. 2001. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79: 281–284.

Nadav, D. 2021. *Enhancing CNN PDB Heuristics for Different Algorithms with Parallel lookups*. Master's thesis, Ben Gurion University.

Nair, V.; and Hinton, G. E. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, 807–814.

Samadi, M.; Siabani, M.; Felner, A.; and Holte, R. 2008. Compressing Pattern Databases with Learning. In *European Conference on Artificial Intelligence*, 495–499.

Shah, A.; Zhan, E.; Sun, J.; Verma, A.; Yue, Y.; and Chaudhuri, S. 2020. Learning Differentiable Programs with Admissible Neural Heuristics. In Larochelle, H.; Ranzato, M.; Hadsell, R.; Balcan, M. F.; and Lin, H., eds., *Advances in Neural Information Processing Systems*, volume 33, 4940–4952. Curran Associates, Inc.

Stern, R.; Felner, A.; and Holte, R. 2011. Probably approximately correct heuristic search. In *Symposium on Combinatorial Search*, 158–163.

Sturtevant, N.; Felner, A.; and Helmert, M. 2014. Exploiting the Rubik's Cube 12-edge PDB by Combining Partial Pattern Databases and Bloom Filters. In *Symposium on Combinatorial Search (SoCS)*, 175–183.

Sturtevant, N. R.; Felner, A.; and Helmert, M. 2017. Value Compression of Pattern Databases. In *AAAI Conference on Artificial Intelligence*, 912–917.

Sturtevant, N. R.; Traish, J.; Tulip, J.; Uras, T.; Koenig, S.; Strasser, B.; Botea, A.; Harabor, D.; and Rabin, S. 2015. The Grid-Based Path Planning Competition: 2014 Entries and Results. In *Eighth Annual Symposium on Combinatorial Search*, 241–251.

Vaswani, S.; Mishkin, A.; Laradji, I.; Schmidt, M.; Gidel, G.; and Lacoste-Julien, S. 2020. Painless Stochastic Gradient: Interpolation, Line-Search, and Convergence Rates. arXiv:1905.09997.

Yonetani, R.; Taniai, T.; Barekatain, M.; Nishimura, M.; and Kanezaki, A. 2021. Path planning using neural A* search. In *International Conference on Machine Learning*, 12029–12039. PMLR.