Learning to Design Data Structures: A Case Study of Nearest Neighbor Search

Omar Salemohamed [1 2] **Laurent Charlin** [* 3 2] **Shivam Garg** [* 4] **Vatsal Sharan** [* 5] **Gregory Valiant** [* 6]

Abstract

We propose a general framework for automating data structure design and apply it to the problem of nearest neighbor search. Our model adapts to the underlying data distribution and provides fine-grained control over query and space complexity, enabling the discovery of solutions tailored to problem-specific constraints. We are able to reverse-engineer learned data structures and query algorithms in several settings. In 1D, the model discovers optimal distribution (in)dependent algorithms such as binary search and variants of interpolation search. In higher dimensions, the model learns solutions that resemble k-d trees in some regimes, while in others, have elements of locality-sensitive hashing.

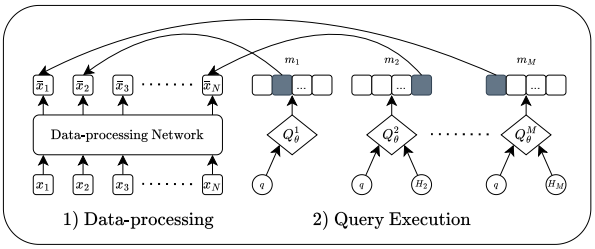


Figure 1. Our model has two components: (i) A data-processing network that transforms raw data into structured data, arranging it for efficient querying and generating additional statistics when given extra space (not shown). (ii) A query-execution network that performs M look-ups into the output of the data-processing network in order to retrieve the answer to some query q. Each lookup i is managed by a separate MLP Q_θ^i, which takes q and the lookup history H_i, and outputs a one-hot lookup vector m_i indicating the position to query.

1. Introduction

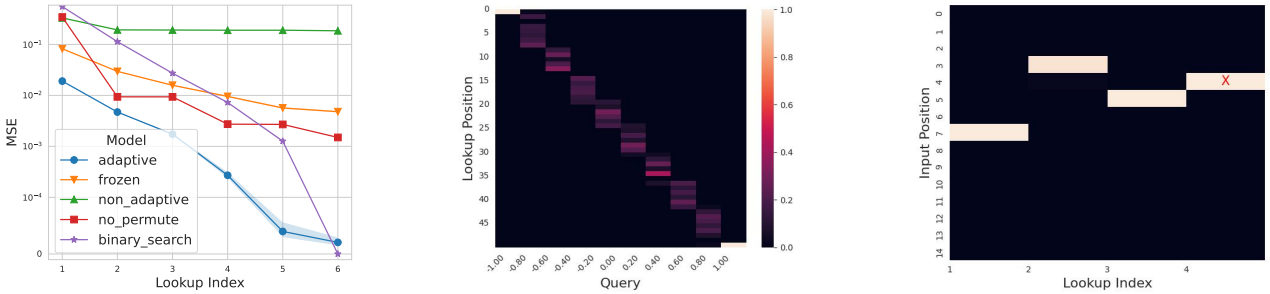
Data structures are ubiquitous objects in computer science that enable efficient querying. Traditionally, they are designed to be worst-case optimal and thus agnostic to specific data and query distributions. However, in many applications, there are patterns in these distributions that can be exploited to design faster algorithms [1]. For instance, interpolation search [2] can significantly outperform binary search for uniformly distributed data. This has motivated recent work on learning-augmented data structures which leverages knowledge of the data distribution to modify existing data structures [1], [3], [4]. In much of this work, the goal of the learning algorithm is only to learn the probability density function of the data distribution and the actual underlying query algorithm/data structure is fixed. While this line of work clearly demonstrates the potential in leveraging distributional information, it still relies on expert knowledge to design and integrate learning into such structures. This raises the more fundamental question: can we learn efficient distribution-dependent data structures from scratch? As an initial step towards this goal, we propose a framework for automating data structure design and apply it to nearest neighbor (NN) search—a problem with extensive theoretical work and widespread practical applications [5]. We show that: (i) In 1D, our model learns to sort the data and apply binary search or variants of interpolation search. (ii) In 2D, our model learns solutions resembling k-d trees, and (iii) In high-dimensions it discovers approximate nearest-neighbor methods resembling locality-sensitive hashing. Additionally, by learning directly from the data distribution, our model can also discover data-dependent data structures and query algorithms that outperform worst-case baselines. Our model learns solutions with a high-degree of interpretability, providing insights for data structure design.

2. Nearest Neighbor Search

Given a dataset $D = \{x_1, ..., x_N\}$ of N points where $x_i \in \mathbb{R}^d$ and a query $q \in \mathbb{R}^d$, the nearest neighbor y of q is defined as $y = \arg\min_{x_i \in D} \ dist(x_i, q)$. We focus on the case where $dist(\cdot)$ corresponds to the Euclidean distance. Our objective is to learn a data structure $\hat{D}_M$ for D such that given q and a budget of M lookups, we can output a (approximate) nearest neighbor of q by querying at most M elements in $\hat{D}_M$. When $M \geq N$, y can be trivially recovered via linear search so $\hat{D}_M = D$ is sufficient. Instead,

[*]Equal contribution [1]Université de Montréal [2]Mila — Quebec AI Institute [3]HEC Montréal [4]Harvard University [5]University of Southern California [6]Stanford University. Correspondence to: Omar Salemohamed <omar.salemohamed@mila.quebec>.

Published at the 2^{nd} Differentiable Almost Everything Workshop at the 41^{st} International Conference on Machine Learning, Vienna, Austria. July 2024.



(a) (b) (c)

Figure 2. (a) Our model (adaptive) trained with 1D data from the uniform distribution over $(-1, 1)$ outperforms binary search and several ablations. (b) Distribution of lookups by the first query model. Unlike binary search, the model does not always start in the middle but rather closer to the query's likely position in the sorted data. (c) When trained on data from the hard distribution, the model finds a solution similar to binary search. The figure shows an example of the model performing binary search ('X' denotes the nearest neighbor location).

we are interested in the case when $M \ll N$[1].

3. Architecture and Training Details

We frame the problem of learning efficient data structures as a two-stage process: 1) data-processing and 2) query-execution (see Fig 1). The role of data-processing is to transform a raw dataset D into a structured database $\hat{D}_M$. Subsequently, the query-execution phase performs M lookups into $\hat{D}_M$ to retrieve the answer for some query q. Below, we discuss the details of the data-processing and query-execution networks in the context of nearest neighbor search but the same framework can be applied to other data structure problems with minor modifications.

3.1. Data-processing Network

The backbone of our data-processing network is a transformer model based on the NanoGPT architecture [6]. The transformer takes as input the dataset D and is trained to output a scalar $o_i \in \mathbb{R}$ representing the rank for each point $x_i \in D$. These rankings $\{o_1, ..., o_N\}$ are then sorted using a differentiable sort function, $sort(\{o_1, o_2 \ldots, o_N\})$ [7], which produces a permutation matrix P that encodes the order based on the rankings. By applying P to the input dataset D, we obtain D_P, where the input data points are arranged in order of their rankings. By learning to rank rather than directly outputting the transformed dataset, the transformer avoids the need to reproduce the exact inputs. Note that this division into a ranking model followed by sorting does not impose any restrictions and the overall model can represent any arbitrary ordering of the inputs. We also consider scenarios where the data structure can use additional space. To support this use case, the transformer can also output T extra tokens $b_1, ..., b_T \in \mathbb{R}^d$ which can be retrieved by the query-execution network. We form the data structure $\hat{D}_M$ by concatenating the permuted inputs and the extra tokens: $\hat{D}_M = [D_P, b_1, ..., b_T]$.

3.2. Query Execution Network

The query-execution network consists of M MLP query models[2] $Q^1_{\theta_1}, ..., Q^M_{\theta_M}$. Each query model $Q^i_{\theta_i}$ outputs a sparse vector $m_i \in \mathbb{R}^{N+T}$ which represents a lookup position in $\hat{D}_M$. To execute the lookup, we compute the value v_i at position m_i in $\hat{D}_M$ as $v_i = m_i^\top \hat{D}_M$. In addition to the query q, each query model $Q^i_{\theta_i}$ also takes as input the query execution history $H_i = \{(m_1, v_1), ..., (m_{i-1}, v_{i-1})\}$ where $H_1 = \emptyset$. The final answer of the network for the nearest-neighbor query is given by $\hat{y} = m_M^\top \hat{D}_M$.

Enforcing sparse lookups To restrict our model to exactly M lookups, we enforce each lookup vector m_i to be a one-hot vector. Enforcing this constraint during training poses a challenge as it is a non-differentiable operation. Instead, during training, our model outputs soft-lookups where m_i is the output of the softmax function and $\sum_j m_{ij} = 1$. This alone, however, leads to non-sparse queries. To circumvent this, we add noise (only during training) to the logits prior to the softmax operation, which leads to sparser solutions (see App E.1 for details).

3.3. Data Generation and Training

Each training example is a tuple (D, q, y) consisting of a dataset D, query q, and nearest neighbor y generated as follows: (i) sample dataset $D = \{x_1, ..., x_N\}$ from dataset distribution $\mathcal{D}$, (ii) sample query q from query distribution Q_D, (iii) compute nearest neighbor $y = \arg\min_{x_i \in D} ||x_i - q||_2$. The dataset and query distributions $\mathcal{D}, \mathcal{Q}_D$ vary across the different settings we consider and are defined later. Given a training example (D, q, y), the data-processing network transforms D into the data structure $\hat{D}_M$. Subsequently, the query execution network, conditioned on q, queries the data structure to output $\hat{y}$. We use SGD to minimize the

[1] E.g. in 1D, binary search requires $M = log(N)$ lookups given a sorted list.

[2] The query models do not share weights.

loss $||\hat{y} - y||_2^2$ averaged over all training examples. After training, we test our model on inputs (D, q, y) generated in the same way. We describe the exact model architecture and training hyper-parameters in App A.

4. Experiments

We now evaluate our model (referred to as **adaptive**) on one-dimensional, two-dimensional, and high-dimensional nearest-neighbor problems. We primarily focus on data structures that do not use extra space, but in Section 4.4, we also explore scenarios with additional space.

Baselines We compare against suitable NN data structures in each setting (e.g., sorting followed by binary search in 1D). In addition, to study the impact of various model components, we compare against several ablations. The **frozen** model does not train the data-processing network, relying on rankings generated by the initial weights. The **no-permute** model removes the permutation component of the data-processing network so that the transformer has to learn to transform the data points directly. The **non-adaptive** model ablation conditions each query model $Q_{\theta_i}^i$ on only the query q and not the query history H_i.

4.1. 1D

Uniform Distribution We consider a setting where $\mathcal{D}$ and Q_D correspond to the uniform distribution over $(-1, 1)$, $N = 50$ and $M = 6$. We plot the mean squared error[3] after each lookup in Figure 2(a). At each lookup index we plot $||v_i^* - y||_2^2$ where v_i^* is the closest element to the query among the first i lookups: $v_i^* = \arg\min_{v \in \{v_1, \ldots, v_i\}} ||v - q||_2^2$. We do this for all methods.

We verify that our model has learned to sort the inputs by measuring the fraction of inputs that are mapped to the correct position in the sorted order, averaged over multiple datasets. After training, our model correctly positions approximately 97% of the inputs. Despite using a separate function for sorting rankings, the model must still learn to output the correct rankings. In comparison, the **frozen** ablation (untrained transformer) positions only about 38% of inputs correctly, explaining its underperformance. The **non-adaptive** baseline, lacking query history access, underperforms as it fails to learn adaptive solutions crucial for 1D NN search. The **no-permute** ablation also underperforms due to its inability to fully retain inputs (verified by measuring the distance between the transformer's inputs and outputs). These ablations highlight the crucial role of both learned orderings and query adaptivity for our model.

Our model outperforms binary search for $M < 6$. This is because unlike binary search (which is optimal only in the worst-case), our model exploits knowledge of the data distribution to start its search closer to the nearest neighbor, similar to interpolation search [2]. For instance, if the query $q \approx 1$, the model begins its search near the end of the list (Fig 2(b)). The minor sorting error ($\sim 3\%$) our model makes likely explains its worse performance on the final query.

In summary, starting from scratch, the data-processing network discovers that the optimal way to arrange the data is in sorted order. Simultaneously, the query-execution network learns to efficiently query this sorted data, leveraging the properties of the data distribution.

Hard Distribution. To verify that our model can also learn worst-case optimal search algorithms such as binary search, we design a hard distribution $\mathcal{D}_{\mathcal{H}}$ with the property that for any given query it is hard to learn a strong prior over the position of its nearest neighbor in the sorted data (see App. C for more details about $\mathcal{D}_{\mathcal{H}}$). We generate our queries by first sampling a point (uniformly at random) from the dataset and then adding noise from the standard normal distribution. In Fig 2(c), we demonstrate a representative example showing that the trained model searches in a manner similar to binary search (see Fig 9 for more examples). In Fig 7, we plot the error curve for the model which closely resembles that of binary search.

4.2. 2D

Uniform Distribution In 2D, we use a similar setup to 1D, sampling coordinates independently from the uniform distribution on $(-1, 1)$. We compare our model to a k-d tree baseline with $N = 50$ and $M = 5$ queries (Fig. 5 in App. B). A k-d tree is a binary tree for organizing points in k-dimensional space, with each node splitting the space along one of the k axes, cycling through the axes at each tree level.

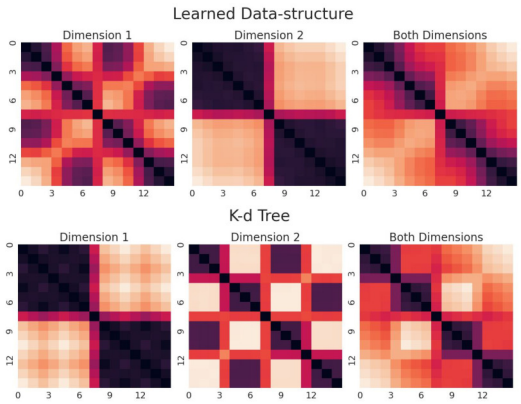
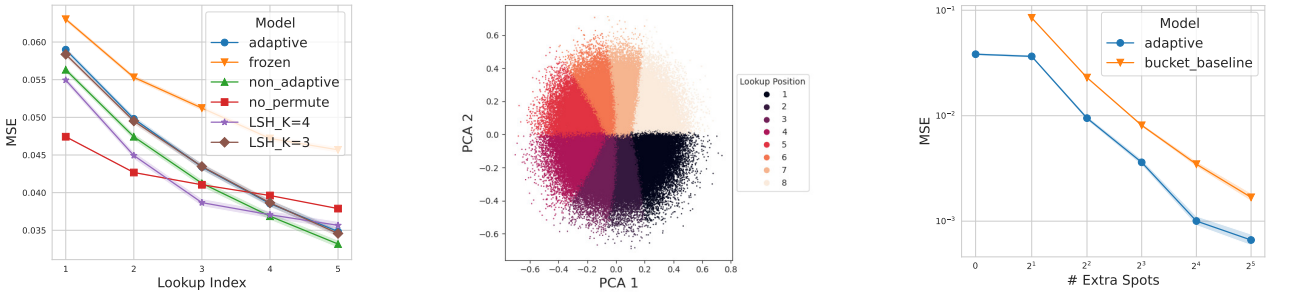


Figure 3. The learned data structure resembles a k-d tree in 2D. We show the average pairwise distances (across the first, second and both dimensions) between points at different positions for the learned data structure and k-d tree, with lighter colors indicating smaller distances. For the k-d tree, data is arranged by in-order traversal of the tree. The plots look similar for k-d trees and the learned data structure, with dimensions 1 and 2 flipped.

[3] We include accuracy plots as well in Appendix B.



(a) (b) (c)

Figure 4. (a) For NN search in higher dimensions (d = 30), the trained models perform comparably to (adaptive) or better than (non-adaptive) locality-sensitive hashing (LSH) baselines. (b) When trained with a single query, the model partitions the query space based on projection onto two vectors, similar to LSH. We show the query projection onto the subspace spanned by these vectors and the lookup positions for different queries. (c) For NN search in 1D (N=32), the model learns to use extra space and outperforms a bucketing baseline.

Similar to the 1D setting, our model outperforms the k-d tree as it can exploit distributional information. Although the **no-permute** ablation outperforms our model, it does not fully retain the inputs, so it is not a feasible alternative. By studying the permutations, we find that our model learns to put points that are close together in the 2D plane next to each other in the permuted order (see Fig. 11 for an example).

Hard Distribution We also consider the case where we sample both coordinates independently from the hard distribution considered in the 1D setup (see Fig 10 for the corresponding error curve). We observe that the data structure learned by our model is surprisingly similar to a k-d tree (see Fig 3). This is striking as a k-d tree is a non-trivial data structure, requiring either an O(N) median finding algorithm or sorting data on both dimensions.

4.3. High Dimensions

High-dimensional NN search poses a challenge for traditional low-dimensional algorithms due to the curse of dimensionality. K-d trees, for instance, can require an exponential number of queries in high dimensions [8]. This has led to the development of approximate NN search methods such as locality sensitive hashing (LSH) which have a milder dependence on d [9], relying on hash functions that map closer points in the space to the same hash bucket with high probability.

In high dimensions, we train our model on datasets uniformly sampled from the d-dimensional unit hypersphere. The query is sampled to have a fixed correlation $\rho \in [0, 1]$ with a dataset point, where $\rho = \frac{|u^T v|}{||u||||v||}$ for vectors $u, v \in \mathbb{R}^d$. When $\rho = 1$, the query matches a data point, making hashing-based methods sufficient. For $\rho < 1$, LSH-based solutions are competitive. We train our model for $\rho = 0.8$ and compare it to an LSH baseline when $N = 50, M = 5$, and $d = 30$. In Fig 4(a), we observe that our model performs competitively with LSH baselines (see details of the baselines in App D). The non-adaptive model does slightly better as adaptivity is not needed to do well in this setting (e.g., LSH is non-adaptive), and lack of adaptivity likely makes training easier. To better understand the data structure our model learns we consider a smaller setting where $N = 8$ and $M = 1$. We find that the model learns an LSH like solution, partitioning the space by projecting onto two vectors in $\mathbb{R}^{30}$ (see Fig 4(b)). We provide more details in App E.3.

4.4. Leveraging Extra Space

The previous experiments demonstrate our model's ability to learn useful orderings for efficient querying. However, data structures can also store additional pre-computed information to speed up querying. For instance, with infinite extra space, a data structure could store the nearest neighbor for every possible query, enabling $O(1)$ search. To evaluate if our model can effectively use extra space, we run an experiment in 1D on the uniform distribution with $N = 32, M = 2$. We allow the data-processing network to output $T \in \{0, 2^1, 2^2, 2^3, 2^4, 2^5\}$ tokens $b_1, ..., b_T \in \mathbb{R}$ in addition to the N rankings. We plot the NN error as a function of T in Fig 4(c) compared to a simple bucketing baseline (described in App E.4.1). The error monotonically decreases with extra space demonstrating that the data-processing network learns to pre-compute useful statistics that enable more efficient querying. We provide some insight into the learned solution in App E.4.2.

5. Conclusion

We propose a framework for learning data structures from scratch and apply it to nearest neighbor search. Our model discovers structures like sorted lists, k-d trees, and locality-sensitive hashing, and simultaneously learns efficient data-dependent algorithms to query them. Additionally, our model leverages extra space to store pre-computed statistics, effectively balancing query time and space complexity. Related work and future work are discussed in App. F & G.

References

[1] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 international conference on management of data*, 2018, pp. 489–504.

[2] W. W. Peterson, "Addressing for random-access storage," *IBM Journal of Research and Development*, vol. 1, no. 2, pp. 130–146, 1957. DOI: 10.1147/rd.12.0130.

[3] M. Mitzenmacher and S. Vassilvitskii, "Algorithms with predictions," *Communications of the ACM*, vol. 65, pp. 33–35, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:219708471.

[4] H. Lin, T. Luo, and D. Woodruff, "Learning augmented binary search trees," in *Proceedings of the 39th International Conference on Machine Learning*, K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvari, G. Niu, and S. Sabato, Eds., ser. Proceedings of Machine Learning Research, vol. 162, PMLR, 17–23 Jul 2022, pp. 13 431–13 440. [Online]. Available: https://proceedings.mlr.press/v162/lin22f.html.

[5] A. Andoni, "Nearest neighbor search: The old, the new, and the impossible," Ph.D. dissertation, Massachusetts Institute of Technology, 2009.

[6] A. Karpathy, *Nanogpt*, https://github.com/karpathy/nanoGPT, Accessed: 2024-05-28, 2024.

[7] A. Grover, E. Wang, A. Zweig, and S. Ermon, "Stochastic optimization of sorting networks via continuous relaxations," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=H1eSS3CcKX.

[8] J. M. Kleinberg, "Two algorithms for nearest-neighbor search in high dimensions," in *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, 1997, pp. 599–608.

[9] A. Andoni, P. Indyk, and I. Razenshteyn, "Approximate nearest neighbor search in high dimensions," in *Proceedings of the International Congress of Mathematicians: Rio de Janeiro 2018*, World Scientific, 2018, pp. 3287–3318.

[10] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[11] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

[12] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," 2017. [Online]. Available: https://arxiv.org/abs/1611.01144.

[13] T. Lykouris and S. Vassilvitskii, "Better caching with machine learned advice," 2018.

[14] H. Lin, T. Luo, and D. Woodruff, "Learning augmented binary search trees," in *International Conference on Machine Learning*, PMLR, 2022, pp. 13 431–13 440.

[15] Y. Dong, P. Indyk, I. Razenshteyn, and T. Wagner, "Learning space partitions for nearest neighbor search," *arXiv preprint arXiv:1901.08544*, 2019.

[16] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[17] P. Veličković, R. Ying, M. Padovano, R. Hadsell, and C. Blundell, "Neural execution of graph algorithms," *arXiv preprint arXiv:1910.10593*, 2019.

[18] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT solver from single-bit supervision," *arXiv preprint arXiv:1802.03685*, 2018.

[19] S. Garg, D. Tsipras, P. S. Liang, and G. Valiant, "What can transformers learn in-context? a case study of simple function classes," *Advances in Neural Information Processing Systems*, vol. 35, pp. 30 583–30 598, 2022.

[20] E. Akyürek, D. Schuurmans, J. Andreas, T. Ma, and D. Zhou, "What learning algorithm is in-context learning? investigations with linear models," *arXiv preprint arXiv:2211.15661*, 2022.

[21] D. Fu, T.-Q. Chen, R. Jia, and V. Sharan, "Transformers learn higher-order optimization methods for in-context learning: A study with linear models," *arXiv preprint arXiv:2310.17086*, 2023.

[22] M. Cuturi, O. Teboul, and J.-P. Vert, "Differentiable ranking and sorting using optimal transport," *Advances in neural information processing systems*, vol. 32, 2019.

[23] F. Petersen, C. Borgelt, H. Kuehne, and O. Deussen, *Monotonic differentiable sorting networks*, 2022. arXiv: 2203.09630.

[24] Y. Xie, H. Dai, M. Chen, *et al.*, "Differentiable top-k with optimal transport," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20 520–20 531, 2020.

[25] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," *arXiv preprint arXiv:1803.02155*, 2018.

A. Training Details

The transformer in the data-processing network is based on the NanoGPT architecture [6] and has 8 layers with 8 heads each and an embedding size of 64. Each query model Q_θ^i is a 3-layer MLP with a hidden dimension of size 1024. Each hidden layer consists of a linear mapping followed by LayerNorm [10] and the ReLU activation function **relu**. In all experiments we use a batch size of 1024, 1e-3 weight decay and the Adam optimizer [11] with default PyTorch **pytorch** settings. For both the transformer and the MLP models we use a learning rate of 1e-4. All models are trained for 4 million gradient steps with early-stopping. We apply the Gumbel Softmax [12] with a temperature of 2 to the lookup vectors to encourage sparsity. For the experiments in 1D, we found it beneficial to only add Gumbel noise to the final lookup vector until training had converged and then add noise to all vectors to find a sparser solution.

B. 1D, 2D, and 30D MSE and Accuracy Plots

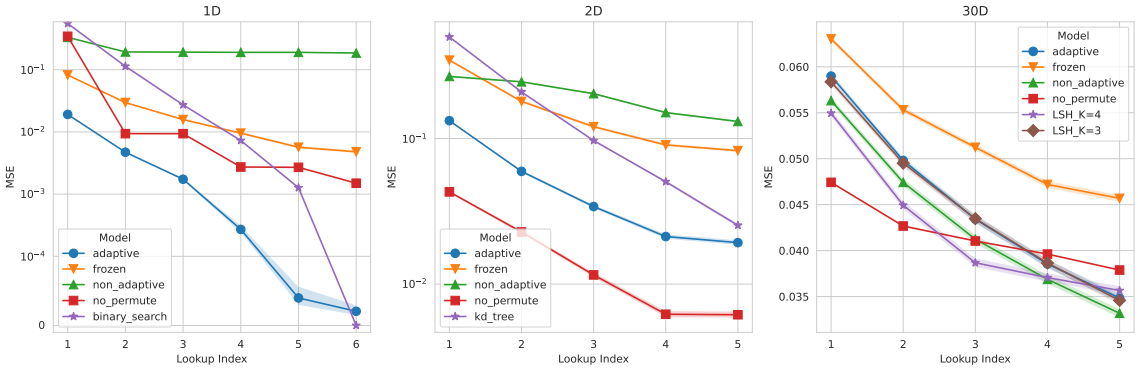


Figure 5. 1D, 2D, 30D N=50 MSE

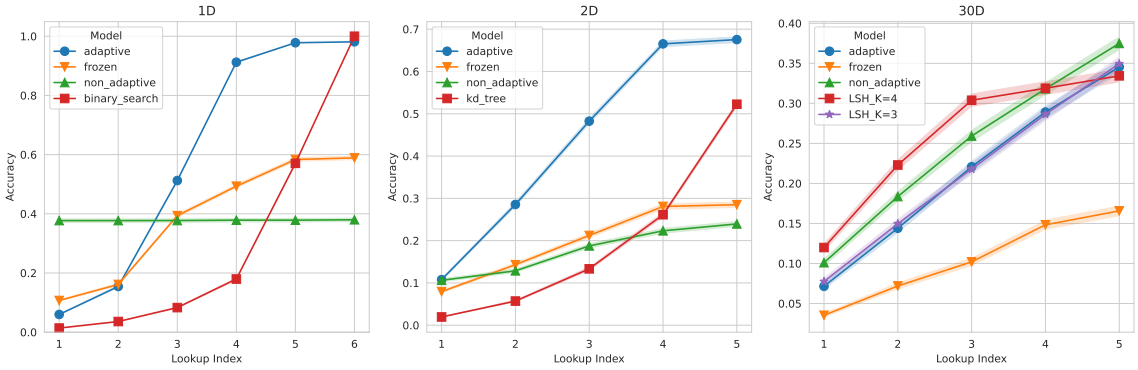


Figure 6. 1D, 2D, 30D N=50 Accuracy

C. Hard Distribution

To generate data from the hard distribution, we first sample the element at the 50th percentile from the uniform distribution over a large range. We then sample the 25th and 75th percentile elements from a smaller range and so on. The intuition behind this distribution is to reduce concentration such that $p(NN|q)$ is roughly uniform where NN denotes the index of the nearest-neighbor of q in the sorted list.

Precisely, to sample N points from the hard distribution we generate a random balanced binary tree of size N. All vertices are random variables of the form $Uniform(0, a^{\log n-k})$ where a is some constant and k is the level in the tree that the vertice belongs to. If the $i-th$ node in the tree is the left-child of its parent, we generate the point x_i as $x_i = x_{p(i)} - d_i$ where $p(i)$ denotes the parent of the $i-th$ node and d_i is a sample from node i of the random binary tree. Similarly, if node

i is the right child of its parent, $x_i = x_{p(i)} + d_i$. For the root element $x_0 = d_0$. In our experiments we set $a = 7$. The larger the value of a, the greater the degree of anti-concentration. We found it challenging to train models with $N > 16$ as the range of values that x_i can take increases with N. Thus for larger N, the model needs to deal with numbers at several scales, making learning challenging.

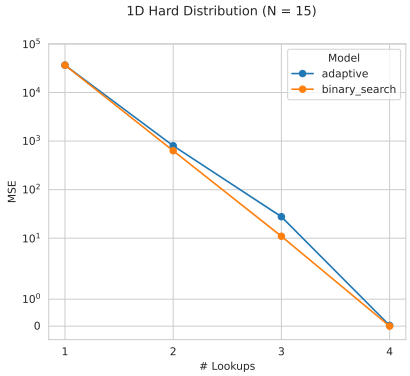


Figure 7. Our model's performance is closely aligned with binary search on the hard distribution in 1D. By design, this distribution does not have a useful prior our model can exploit and so it learns a binary search like solution.

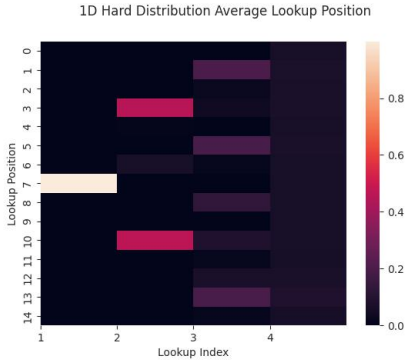


Figure 8. The positional distribution per lookup in the 1D Hard experiment. Our model closely aligns with binary search, first looking at the middle element, then (approximately) either the 25th or 75th percentile elements, and so on.

D. LSH Baseline

Our LSH baseline samples K random vectors $\mathbf{r_1}, ..., \mathbf{r_K}$ from the standard normal distribution in $\mathbb{R}^d$. For a given vector $\mathbf{v} \in \mathbb{R}^d$, its hash code is computed as $hash(\mathbf{v}) = [sign(\mathbf{v^T r_1}), ..., sign(\mathbf{v^T r_K})]$. In total, there are 2^K possible hash codes. To create a hash table, we assign each hash code a bucket of size $N/2^K$. For a given dataset $D = \{x_1, ..., x_N\}$, we place each input in its corresponding bucket (determined by its hash code $hash(x_i)$. If the bucket is full, we place x_i in a vacant bucket chosen at random. Given a query q and a budget of M lookups, the baseline retrieves the first M vectors in the bucket corresponding to $hash(q)$. If there are less than M vectors in the bucket, we choose the remaining vectors at random from other buckets. We design this setup like so to closely align with the constraints of our model (i.e. only learning a permutation).

E. Additional Experiment Findings

E.1. Noise Injection for Lookup Sparsity

We find that adding noise prior to applying the soft-max on the lookup vector m_i leads to sparser queries. We hypothesize that this is because the noise injection forces the model to learn a noise-robust solution which corresponds to a sparse solution.

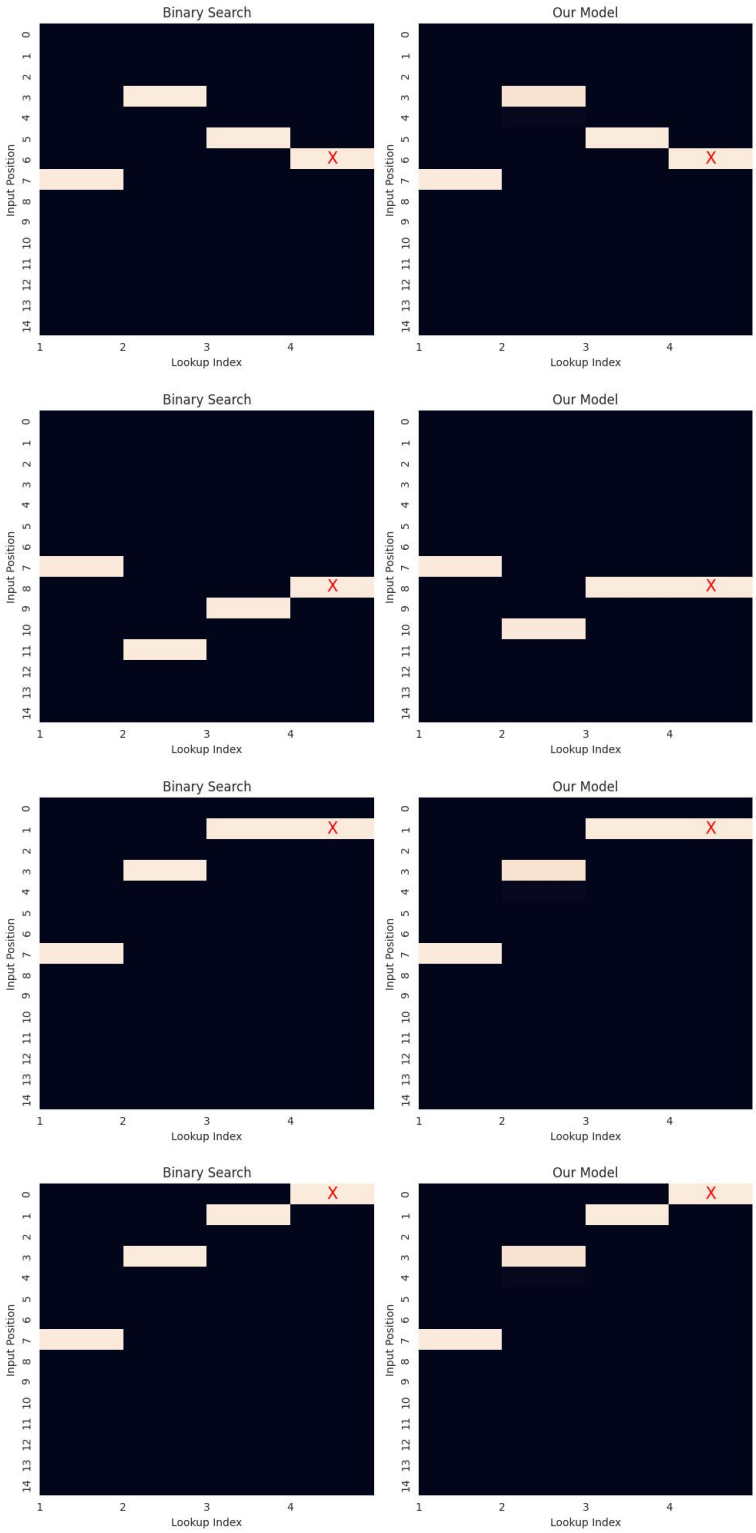


Figure 9. Binary Search vs. our model on the hard distribution in 1D

Consider a simplified setup in 1D where the query model is not conditioned on q and is only allowed one lookup ($M = 1$) and D is a sorted list of three elements: $D = [x_1, x_2, x_3]$. For a given query q and its nearest neighbor y, the query-execution

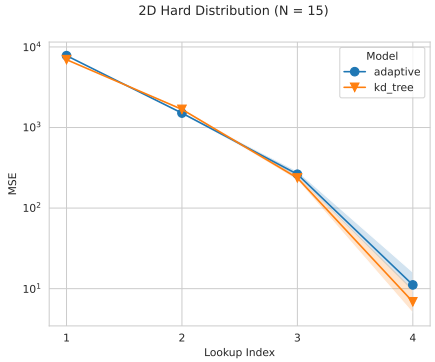


Figure 10. On the 2D hard distribution our model roughly tracks the performance of a k-d tree.

network is trying to find the optimal vector $\hat{m} \in \mathbb{R}^3$ that minimizes $||y - m^T D||_2^2$ where $m = softmax(\hat{m} + \epsilon), \epsilon \sim$ Gumbel distribution [12]. Given that $M = 1$, the model cannot always make enough queries to identify y and so in the absence of noise the model may try to predict the 'middle' element by setting $\hat{m}_1 = \hat{m}_2 = \hat{m}_3$. However, when noise is added to the logits $\hat{m}$ this solution is destabilized. Instead, in the presence of noise, the model can robustly select the middle element by making $\hat{m}_2$ much greater than $\hat{m}_1, \hat{m}_3$. We test this intuition by running this experiment for large values of N and find that with noise the average gradient is much larger for $\hat{m}_{N/2}$.

E.2. 2D Uniform Distribution

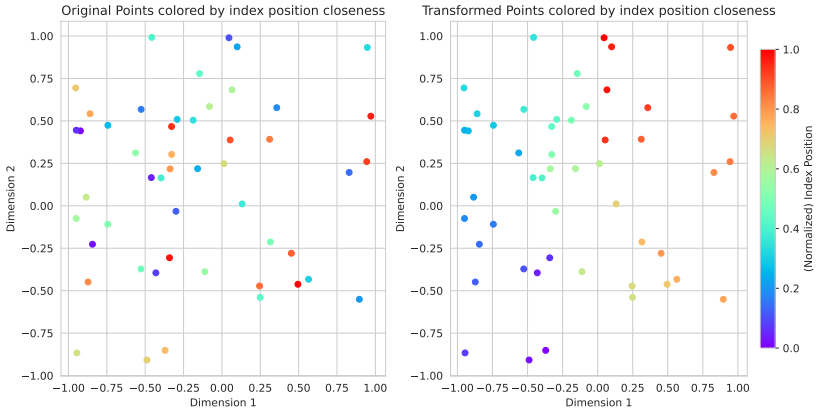


Figure 11. Our model's learned permutation on the 2D uniform distribution. The model puts elements that are close together in the Euclidean plane next to each other in the permutation.

E.3. N=8, M=1 30D Experiment

To determine if our model has learned an LSH-like solution, we try to reverse engineer the query model in a simple setting where $N = 8$ and $M = 1$. The query-execution model is only allowed one lookup. We fit 8 one-vs-rest logistic regression classifiers using queries sampled from the query distribution and the output of the query model (lookup position) as features and labels, respectively. We then do PCA on the set of 8 classifier coefficients. We find that the top 2 principal components explain all of the variance which suggests that the query model's mapping can be explained by the projection onto these two components. In Figure 13 we plot the projection of queries onto these components and color them based on the position they were assigned by the query model. We do the same for inputs $x_i \in D$ and color them by the position they were permuted to. The plot on the right suggests that the data-processing network permutes the input vectors based on their projection onto these two components. This assignment is noisy because there may be multiple inputs in a dataset that map to the same

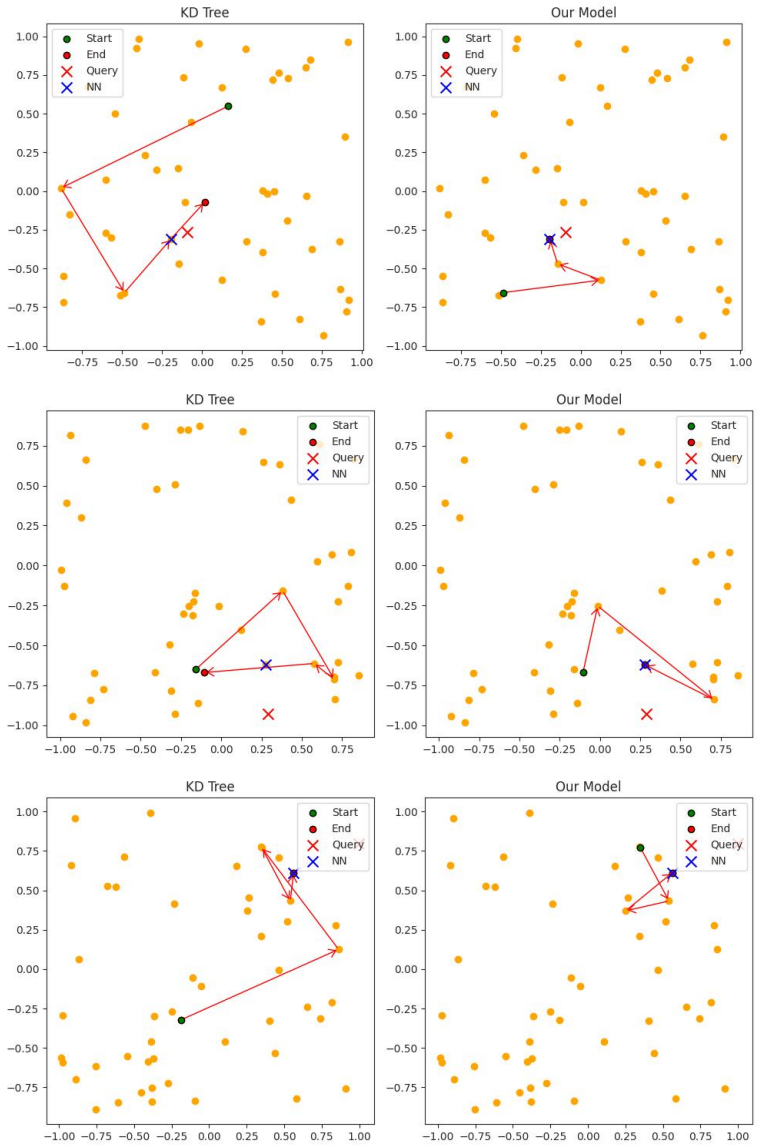


Figure 12. k-d search vs. our model on the uniform distribution in 2D. Unlike the k-d tree, our model has a stronger prior over where to begin its search.

bucket and because the model can only store a permutation, some buckets experience overflow. Similarly, the query model does a lookup in the position that corresponds to the query vector's bucket. This behaviour suggests the model has learned a locality-sensitive hashing type solution!

E.4. 1D Extra Space

E.4.1. Bucket Baseline

We create a simple bucket baseline that partitions $[-1, 1]$ into T evenly sized buckets. In each bucket b_i we store $argmin_{x_j \in D} ||x_j - l_i||$ where l_i is the midpoint of the segment partitioned in b_i. This baseline maps a query to its corresponding bucket and predicts the input stored in that bucket as the nearest-neighbor. As $T \to \infty$ this becomes an optimal hashing-like solution.

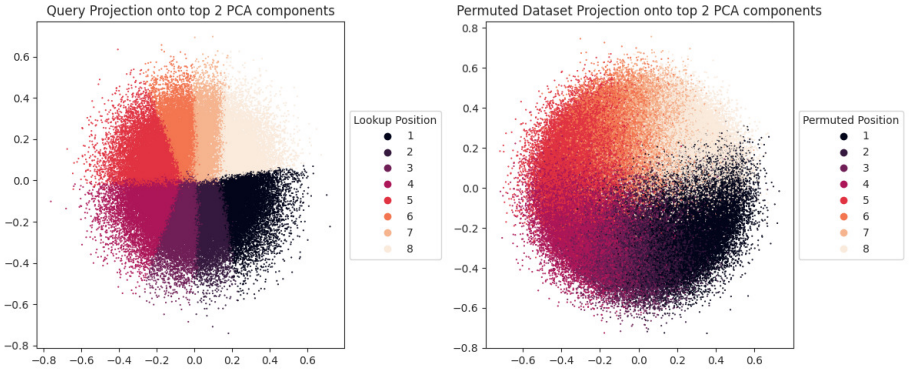
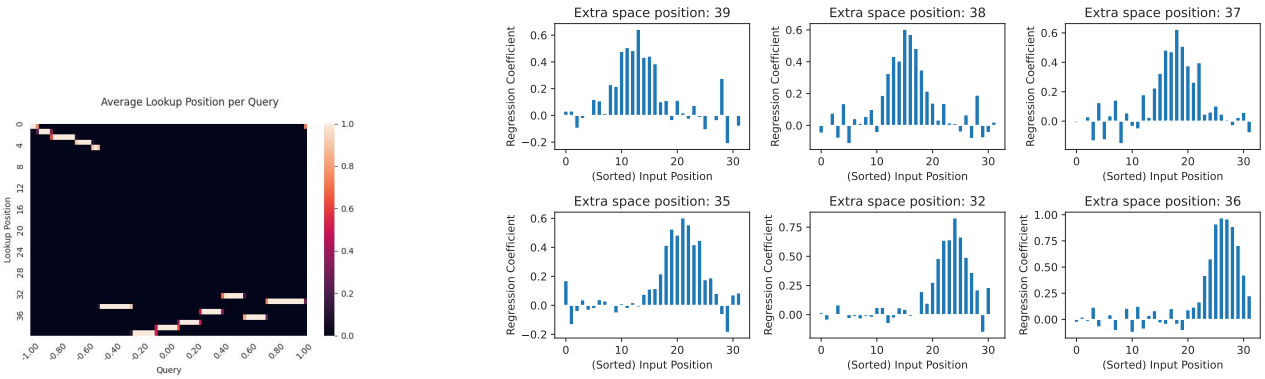


Figure 13. **Left** Projection of queries onto top two PCA components of the decision boundaries of the query model, colored by the lookup position the query is mapped to. **Right** Projection of inputs onto the same PCA components colored by the position the data-processing model places them in. Both the data-processing and query models map similar regions to the same positions, suggesting an LSH-like bucketing solution has been learned.



(a) (b)

Figure 14. (a) Decision boundary of the first query model. (b) The regression coefficients of the values stored in extra positions as a linear function of the (sorted) inputs.

E.4.2. Understanding Extra Space Usage

By analyzing the lookup patterns of the first query model, we can better understand how the model uses extra space. In Figure 14(a) we plot the decision boundary of the first query model. The plot demonstrates that the model chunks the query space ($[-1, 1]$) into different buckets. To get a sense of what the model stores in the extra space, we fit a linear function on the sorted inputs and regress the values stored in each of the extra space tokens b_i and plot the coefficients for several of the extra spaces in Figure 14(b). For a given subset of the query range, the value stored at its corresponding extra space is approximately a weighted sum of the values stored at the indices that correspond to the percentile of that query range subset. This is useful information as it tells the model for a given query percentile how 'shifted' the values in the current dataset stored in the corresponding indices are from model's prior.

F. Related Work

To the best of our knowledge there is no prior work on using machine learning to design efficient data structures and query algorithms end-to-end from scratch. However, we discuss several related works below and highlight the connections to our work.

Learning-Augmented Algorithms Recent work has shown that traditional data structures and algorithms can be made more efficient by learning properties of the underlying data distribution. For example, Lykouris and Vassilvitskii [13] proposed learning-augmented algorithms for online caching and paging problems, where predictions about future requests are used to improve cache replacement policies. Kraska, Beutel, Chi, *et al.* [1] introduced the concept of learned index structures, which use machine learning models to replace traditional index structures in databases, resulting in significant performance improvements for certain query workloads. By learning the cumulative distribution function of the data distribution the model has a stronger prior over where to start the search for a record. Other works augment the data structure with predictions instead of the query algorithm. For example, Lin, Luo, and Woodruff [14] use learned frequency estimation oracles to estimate the priority in which elements should be stored in a treap. Perhaps most relevant to the theme of our work is [15], which trains neural networks to learn a partitioning of the space for efficient nearest neighbor search using locality sensitive hashing.

In much of these works, the goal of the learning algorithm is only to learn certain properties of the data distribution while most components of the underlying data structure/query algorithm remain fixed. While this line of work clearly demonstrates the potential in leveraging distributional information, it still relies on expert knowledge to design and integrate learning into such structures. The goal of our work is to push this idea even further by learning the data structure as well as the query algorithm together from scratch, allowing much greater adaptability to the underlying data distribution.

Neural Algorithmic Learners Neural algorithmic learners focus on embedding traditional algorithms into neural network frameworks, allowing these models to learn and execute algorithmic tasks. One of the pioneering works in this area is the Neural Turing Machine (NTM) proposed by Graves, Wayne, and Danihelka [16], which combines a neural network with an external memory, enabling it to learn and perform algorithmic tasks such as sorting and copying . More recent approaches, such as those by Veličković, Ying, Padovano, *et al.* [17], have employed graph neural networks (GNNs) to learn to perform classical algorithms such as breadth-first search (BFS) and shortest path algorithms, leveraging their ability to handle structured data efficiently.

While these works share a similar motivation of training neural networks to execute algorithms, they are trained with a much greater degree of supervision than our model and focus on embedding known algorithms into neural networks. For instance, Graves, Wayne, and Danihelka [16] use the ground truth sorted list as supervision to train the model to learn to sort and other works even use intermediate computations of algorithms as additional supervision [17]. Instead, we are interested in discovering efficient solutions to more general tasks (e.g. nearest-neighbor search). Typically, such solutions require learning algorithmic primitives (such as sorting) and so our model is indirectly encouraged to learn these. In this sense, we take a more top-down approach. Instead of training neural networks to learn specific (known) primitives that can be manually combined to solve a given task, we instead train models to solve the task directly. This approach allows the model more freedom to discover solutions adapted to the specific task distribution. However, this is also more challenging as our model needs to learn multiple algorithms in tandem, e.g. learning to both sort and execute binary search with only the desired output as supervision.

There has also been work on learning end-to-end algorithms. Selsam, Lamm, Bünz, *et al.* [18] train neural networks to solve SAT. Garg, Tsipras, Liang, *et al.* [19] and Akyürek, Schuurmans, Andreas, *et al.* [20] show that transformers can be trained to encode learning algorithms for function classes such as linear functions and decision trees, and Fu, Chen, Jia, *et al.* [21] observe that trained transformers discover algorithms resembling higher-order optimization methods.

Differentiable Algorithms Differentiable algorithms are traditional algorithms that have been modified to allow gradient-based optimization, making them compatible with neural network training. This approach enables the integration of algorithmic components directly into end-to-end machine learning models.

For instance, Grover, Wang, Zweig, *et al.* [7] proposed differentiable sorting and ranking functions, which approximate traditional sorting algorithms with differentiable counterparts, allowing them to be used in gradient-based optimization frameworks. In our work, we make use of this differentiable sorting function to reorder the input dataset into a data structure. More recent works have proposed other differentiable sorting algorithms that use optimal transport or enforce monotonicity [22], [23].

Xie, Dai, Chen, *et al.* [24] introduced differentiable top-k selection algorithms which enable the integration of top-k selection within neural network architectures. Operating on pairwise distances, these top-k algorithms can be used to retrieve k-nearest neighbors in a differentiable manner. However, this requires computing distances between the query and every item in the

dataset (i.e. N lookups for a dataset of size N). Instead, our work focuses on learning data structures and algorithms such that nearest-neighbor queries can be executed with $M << N$ lookups.

At a high-level, both components of our model can be regarded as differentiable algorithms themselves. For example, in the context of high-dimensional NN search, the trained data-processing network buckets the input dataset based on hash codes. The query-processing network searches in these buckets based on the query's hash code. Both of these operations are fully differentiable. Thus, our model can also be integrated into a larger pipeline that requires a learned differentiable algorithm.

G. Limitations and Future Work

There are several limitations to our current model that we plan to address in future work. One limitation is scaling. Our current model can find sparse solutions up to $N = 50$ and non-sparse solutions up to $N = 150$ (Figure 15). While we demonstrate that useful data structures can still be learned at this scale, it is possible that other classes of structures only emerge for larger datasets. It is likely that significantly scaling up the parameter count can help scale up N. Complementary to this, it would be worthwhile to explore better inductive biases for the query and data-processing networks, and other methods to ensure sparse solutions, enabling smaller models to scale to larger datasets. While we intentionally refrained from introducing additional inductive bias in this work in order to give both models more degrees of freedom, there are several modifications that are likely helpful such as shared weights among query-models and using relative lookups (similar to relative position encoding in transformers [25]) as opposed to absolute lookups.

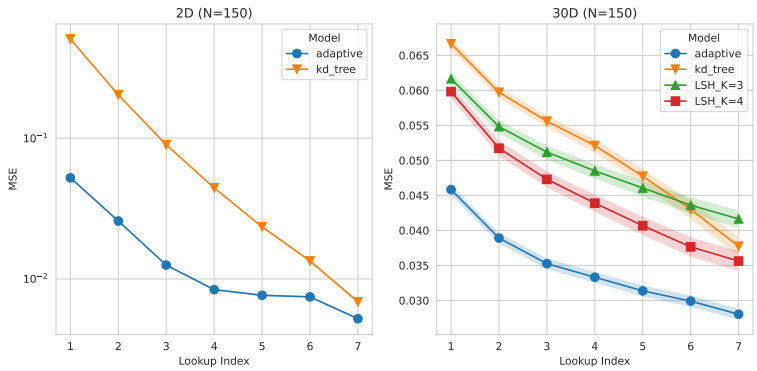


Figure 15. 2D and 30D experiments with $N = 150$ and $M = 7$. Our model can learn competitive solutions at this scale however they are not fully sparse.