What Can Neural Networks Reason About?

Keyulu Xu* Jingling Li† Mozhi Zhang† Simon S. Du ‡
Ken-ichi Kawarabayashi§ Stefanie Jegelka*

* Massachusetts Institute of Technology (MIT)
† University of Maryland ‡ Carnegie Mellon University
§ National Institute of Informatics

Abstract

Neural networks have successfully been applied to solving reasoning tasks, ranging from learning simple concepts like "close to", to intricate questions whose reasoning procedures resemble algorithms. Empirically, not all network structures work equally well for reasoning. For example, Graph Neural Networks have achieved impressive empirical results, while the less structured neural networks may fail to learn to reason. Theoretically, there is currently limited understanding of the interplay between reasoning tasks and network learning.

In this paper, we develop a framework to characterize which tasks a neural network can learn well, by studying how well its structure aligns with the algorithmic structure of the relevant reasoning procedure. This suggests that Graph Neural Networks can learn dynamic programming, a powerful algorithmic strategy that solves a broad class of reasoning problems, such as relational question answering, sorting, intuitive physics, and shortest paths. Our perspective also implies strategies to design neural architectures for complex reasoning. On several abstract reasoning tasks, we see empirically that our theory aligns well with practice.

Correspondence to: keyulu@mit.edu

1 Introduction

Reasoning is about grasping the relations between objects in the world, and from that deriving sophisticated conclusions and predictions [47, 42, 31]. Recently, interest has been resurging to build neural networks that can learn to reason [43, 20, 42]. To that end, a broad class of reasoning tasks have been designed, including relational visual question answering [24, 22, 16, 2, 32], intuitive physics, *i.e.*, predicting the time evolution of physical objects [7, 53, 54, 17, 12], more abstract mathematical reasoning [43, 11] and visual IQ tests [42, 59].

Curiously, neural networks that perform well in reasoning tasks usually possess specific, explicit *structure* in their computation graph [41, 8, 51], while less structured networks, e.g., fully connected architectures, often fail [42, 41]. Many empirically successful models for reasoning follow the Graph Neural Network (GNN) framework [8, 56, 45, 29]. These models consider pairwise relations, and recursively update each object's representation by aggregating its relations with other objects [7, 34, 23, 37, 46, 40]. Other computational structures, *e.g.*, symbolic program execution [25, 57], have been effective for other tasks.

However, there is currently limited understanding of how the reasoning ability and the structure of a neural network relate. *What tasks can a neural network learn to reason about? When and why is a network structure more effective than the others?* Answering these questions can be crucial for building better neural networks for complex reasoning.

This paper is an initial work towards answering these fundamental questions. We develop a theoretical framework to characterize what tasks a neural network can reason about well. Our framework is motivated by a seemingly simple observation: many reasoning procedures resemble algorithms. Hence, we study how well a reasoning algorithm "aligns" with the computation graph of the network. Intuitively, if the structures align, the network can easily learn to simulate the reasoning procedure.

We formalize this intuition of alignment, and show initial support for our hypothesis that alignment facilitates learning. First, we provide an example how the alignment can affect a theoretical analysis: in a sequential setting, when the structures of network and reasoning algorithm align, the network can learn more sample-efficiently, since the sub-modules it needs to learn are simpler.

Next, we study what algorithms a few example neural architectures structurally align with. In particular, we highlight that GNNs structurally match the broadly applicable algorithmic paradigm of dynamic programming (DP) [10]. We illustrate that we can solve a broad range of reasoning problems with DP, and thus, GNNs. Our results offer an explanation for the effectiveness (and hence popularity) of GNNs for reasoning, and are reflected empirically.

Our algorithmic structural condition differs from structural assumptions common in learning theory [49, 6, 5, 36, 19, 15, 4, 1] and specifically aligns with reasoning. Our main contribution is to introduce and formalize this *structural alignment perspective*, along with implications. This new perspective also implies strategies for designing architectures for complex reasoning. Our main results are summarized as follows:

1. We introduce the perspective of algorithmic alignment to analyze learning for reasoning.
2. Our initial theoretical results suggest that structural alignment is desirable for generalization.
3. We apply our perspective to analyze what we expect some popular networks to learn well, and show that Graph Neural Networks align with dynamic programming.
4. On a test suite of reasoning tasks, our empirical results support our theoretical findings.

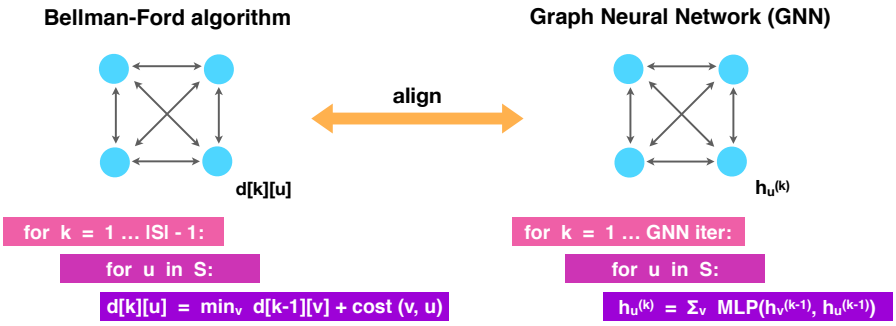


Figure 1: **Algorithmic alignment.** Our framework suggests that algorithmic alignment implies generalization. The algorithm (bottom left) and the neural network (bottom right) structurally align: the network can simulate the algorithm by filling in (learning) *simple functions* via the MLP modules (bottom purple blocks). An MLP, however, does not well align with the algorithm, because it needs to learn to simulate the entire for loop. The top row illustrates the structures: nodes are variables in an algorithm or representation vectors in a network; arrows refer to an algorithm step or an MLP taking in end nodes as input.

2 Preliminaries: Abstract Reasoning

We begin by summarizing our setting and common models for reasoning and, along the way, introduce our notation. Let S denote the universe, *i.e.*, a configuration/set of objects to reason on. The object representation vectors, $X_s \in \mathbb{R}^n$ for $s \in S$, could be state descriptions [57, 41] or high-level features learned from raw data [30, 41]. Information about the question can be included in the object representations. Given a set of universes $\{S_1, ..., S_M\}$ and answer labels $\{y_1, ..., y_M\} \subseteq \mathcal{Y}$, we aim to learn a function g, potentially parameterized by a neural network, that can answer questions about unseen universes, $y = g(S)$.

If we reason about a single-object universe, *e.g.*, classification, applying a multilayer perceptron (MLP) on the object representation usually works well [30]. But, for multiple-object universes, simply applying an MLP to the concatenated object representations often leads to poor generalization [41]. To better learn functions of a set of objects, Zaheer et al. [58] propose *Deep Sets*. Deep Sets' structure, $y = \text{MLP}_2\left(\sum_{s\in S} \text{MLP}_1(X_s)\right)$ encodes *permutation-invariant* functions.

Graph Neural Networks (GNNs). GNNs too are permutation invariant, but focus on pairwise relations. Their structure follows a message passing scheme [18, 55], where the representation $h_s^{(k)}$ of each object s (in layer k) is recursively updated by aggregating its pairwise interactions with other objects:

$$h_s^{(k)} = \sum_{t\in S} \text{MLP}_1^{(k)}\left(h_s^{(k-1)}, h_t^{(k-1)}\right), \quad h_S = \text{MLP}_2\left(\sum_{s\in S} h_s^{(K)}\right), \tag{2.1}$$

where h_S is the answer/output and K is the number of GNN layers. We initialize $h_s^{(0)} = X_s$. Instead of the sum, other aggregation functions are used, too.

Originally proposed for learning on graphs [45], GNNs have become a widely used model for reasoning too [8]. Relation Networks [41] and Interaction Networks [7] resemble one-layer GNNs, Recurrent Relational Networks [37] apply LSTMs [21] after aggregation, and Transformers [50] aggregate with an attention mechanism. While, in graphs, one aggregates over neighboring nodes, GNNs for reasoning typically use all pairs, corresponding to message passing on a complete graph.

3 Theoretical Framework: Algorithmic Structural Alignment

Curiously, many models for reasoning are structured neural networks. Next, we study how the network structure and task may interact. To do so, we observe that the answer to many reasoning tasks may be derived by following an algorithm; we further illustrate this in Section 4. For example, the answer to the shortest paths problem can be computed by the Bellman-Ford algorithm [9], shown in Fig. 1. Intuitively, if a network can learn the algorithm, it can learn to answer the task.

In principle, many neural networks can *represent* algorithms. For example, DeepSets can universally represent permutation-invariant set functions [58, 52]. This also holds for GNNs and MLPs (our setting differs from [44, 56], who study functions on graphs):

Proposition 3.1. Let $f : \mathbb{R}^{d\times N} \to \mathbb{R}$ be any continuous functions over sets S of bounded cardinality $|S| \leq N$. If f is permutation-invariant to the elements in S and the elements are in a compact set in $\mathbb{R}^d$, then f can be approximated arbitrarily closely by a GNN (of any depth).

Proposition 3.2. For any GNN $\mathcal{N}$, there is an MLP that can represent all functions $\mathcal{N}$ can represent.

But, empirically, not all network structures work equally well when *learning* these algorithms. Intuitively, a network may learn well if it can represent a function "more easily". We formalize this idea by our notion of *algorithmic alignment*. Indeed, not only the reasoning procedure has an algorithmic structure: the neural network's architecture induces a computational structure on the function it computes. This corresponds to an algorithm that prescribes how the network combines computations from modules. Fig. 1 illustrates this idea for a GNN, where the modules are the MLPs applied to pairs of objects. The GNN "matches" the algorithmic structure of the Bellman-Ford algorithm: it can simulate the algorithm if each of its modules learns a *simple* function (sum or min).

We could simulate the Bellman-Ford algorithm with an MLP too. But, a match of network modules to algorithmic steps would e.g. need to learn sparse MLPs that mimic the pairwise operations despite all objects as input, or an MLP that simulates the *entire for loop* as a whole, which is a much more complex function and, hence, would presumably need *many more* samples to learn. In short, without aligning well, the network may have to infer more of the algorithmic structure from data.

This perspective suggests that whether a neural network can reason about a task may depend on whether *there exists* an algorithmic solution that the network structurally aligns with.

3.1 Formalization

We formalize the above intuition about simple modules in a PAC learning framework [48]. PAC learnability formalizes *simplicity* as sample complexity, i.e., the number of samples needed to ensure low test error with high probability. It refers to a learning algorithm $\mathcal{A}$ that, conditioned on training samples $\{x_i, y_i\}_{i=1}^M$, outputs a function $f = \mathcal{A}(\{x_i, y_i\}_{i=1}^M)$. The learning algorithm here is the training method for the neural network, which is different from the reasoning algorithm.

Definition 3.3. (Learnability). Fix an error parameter $\epsilon > 0$ and failure probability $\delta \in (0, 1)$. Suppose $\{x_i, y_i\}_{i=1}^M$ are i.i.d. samples from some distribution $\mathcal{D}$, and the data satisfies $y_i = g(x_i)$ for some underlying function g. Let $f = \mathcal{A}(\{x_i, y_i\}_{i=1}^M)$ be the function generated by a learning algorithm $\mathcal{A}$. Then g is (M, ϵ, δ)-*learnable* with $\mathcal{A}$ if

$$\mathbb{P}_{x\sim\mathcal{D}}\left[\|f(x) - g(x)\| \leq \epsilon\right] \geq 1 - \delta. \tag{3.1}$$

The *sample complexity* $\mathcal{C}_{\mathcal{A}}(g, \epsilon, \delta)$ is the minimum M so that g is (M, ϵ, δ)-learnable with $\mathcal{A}$.

We then say that a network architecture aligns well with an algorithm if it can simulate the algorithm via a limited number of (types of) modules, and each module is simple, i.e., efficiently learnable.

Definition 3.4. (Algorithmic structural alignment). Let g be a reasoning function and $\mathcal{N}$ a neural network with n modules $\mathcal{N}_i$. The module functions $f_1, ..., f_n$ generate g for $\mathcal{N}$ if, by replacing $\mathcal{N}_i$ with f_i, the network $\mathcal{N}$ simulates g. Then $\mathcal{N}$ (M, ϵ, δ)-aligns with g if (1) $f_1, ..., f_n$ generate g and (2) there are learning algorithms $\mathcal{A}_i$ for the $\mathcal{N}_i$'s such that $n \cdot \max_i C_{\mathcal{A}_i}(f_i, \epsilon, \delta) \leq M$.

The structures align well if the sample complexity M is small. This implies all algorithm steps f_i are *easy to learn*. The number of modules n can be kept small via weight sharing.

Next, we show an initial result demonstrating that structural alignment is desirable for generalization. Theorem 3.5 states that, in a setting where we sequentially train modules of a network with auxiliary labels, alignment implies generalization: if the network (M, ϵ, δ)-aligns with an algorithm, then the algorithm is $(M, O(\epsilon), O(\delta))$-learnable by the network. In Section 5 we will see that, empirically, the same pattern holds for end-to-end learning. We prove Theorem 3.5 in the Appendix.

Theorem 3.5. (Structural alignment implies learnability). Fix ϵ and δ. Suppose $\{S_i, y_i\}_{i=1}^M \sim \mathcal{D}$, where $|S_i| < N$, and $y_i = g(S_i)$ for some g. Suppose $\mathcal{N}_1, ..., \mathcal{N}_n$ are network $\mathcal{N}$'s MLP modules in a sequential order. Under the following assumptions, g is $(M, O(\epsilon), O(\delta))$-learnable by $\mathcal{N}$.

a) Structural alignment. $\mathcal{N}$ and g (M, ϵ, δ)-align via functions $f_1, ..., f_n$.

b) Algorithm stability. Let $\mathcal{A}$ be the learning algorithm for the $\mathcal{N}_i$'s. Suppose $f = \mathcal{A}(\{x_i, y_i\}_{i=1}^M)$, and $\hat{f} = \mathcal{A}(\{\hat{x}_i, y_i\}_{i=1}^M)$. For any x, $\|f(x) - \hat{f}(x)\| \leq L_0 \cdot \max_i \|x_i - \hat{x}_i\|$, for some L_0.

c) Sequential training. We train $\mathcal{N}_i$'s sequentially: $\mathcal{N}_1$ has input samples $\{\hat{x}_i^{(1)}, f_1(\hat{x}_i^{(1)})\}_{i=1}^N$, with $\hat{x}_i^{(1)}$ obtained from S_i. For $j > 1$, the input $\hat{x}_i^{(j)}$ for $\mathcal{N}_j$ are the outputs from the previous modules, but labels are generated by the correct functions $f_{j-1}, ..., f_1$ on $\hat{x}_i^{(1)}$.

d) Lipschitzness. The learned functions $\hat{f}_j$ satisfy $\|\hat{f}_j(x) - \hat{f}_j(\hat{x})\| \leq L_1 \|x - \hat{x}\|$, for some L_1.

In our analysis, the Lipschitz constants, the universe size, and number of MLP modules are constants going into $O(\epsilon)$ and $O(\delta)$. While a fine-grained analysis is possible, we leave it for future work.

3.2 MLPs and Sample Efficiency Gap

The generalization bound via alignment in Theorem 3.5 depends on the sample complexity of the MLP modules. Hence, next, we study learnability with MLPs. Recent work shows sample complexity bounds for overparameterized two or three-layer MLPs by analyzing their gradient descent trajectories [4, 1]. Theorem 3.6, proved in the Appendix, summarizes and extends Theorem 6.1 of Arora et al. [4] to vector-valued functions.

Theorem 3.6. (Sample Complexity for MLPs). Let $\mathcal{A}$ be an overparameterized and randomly initialized two-layer MLP trained with gradient descent for a sufficient number of iterations. Suppose $g : \mathbb{R}^d \to \mathbb{R}^m$ with components $g(x)^{(i)} = \sum_j \alpha_j^{(i)} \big(\beta_j^{(i)\top} x\big)^{p_j^{(i)}}$, where $\beta_j^{(i)} \in \mathbb{R}^d$, $\alpha \in \mathbb{R}$, and $p_j^{(i)} = 1$ or $p_j^{(i)} = 2l$ $(l \in \mathbb{N}_+)$. The sample complexity $\mathcal{C}_{\mathcal{A}}(g, \epsilon, \delta)$ is

$$\mathcal{C}_{\mathcal{A}}(g, \epsilon, \delta) = O\Big(\frac{\max_i \sum_{j=1}^K p_j^{(i)} |\alpha_j^{(i)}| \cdot \|\beta_j^{(i)}\|_2^{p_j^{(i)}} + \log(m/\delta)}{(\epsilon/m)^2}\Big). \tag{3.2}$$

Theorem 3.6 says that if a function is "simple" when expressed as a polynomial, *e.g.*, via a Taylor expansion, it is learnable by an MLP. By this notion, complex interactions that involve *many objects* may require many samples for MLPs to learn, since the number K of polynomials or $\|\beta_j^{(i)}\|$ may increase in (3.2). Although Theorem 3.6 only gives an upper bound on the sample complexity, it might still provide a plausible explanation for why MLPs fail to learn.

Corollary 3.7. Suppose the universe S contains ℓ objects $X_1, ..., X_\ell$, and we have $g(S) = \sum_{i,j}(X_i - X_j)^2$. In the sequential learning setting, the upper bound on sample complexity for MLP is $O(\ell^2)$ times higher for GNN.

The example in Corollary 3.7 illustrates how neural networks with a *matching structure*, *e.g.*, GNNs, may get a polynomial improvement in sample complexity over MLPs. In practice, an $O(\ell^2)$ efficiency gap can be serious with many objects to reason about.

Our framework is general and can work with other sample complexity bounds for MLPs too. Theorem 3.6 is an illustrative example. Next, we apply our algorithmic alignment perspective to *foresee* which tasks some popular networks can reason about.

4 Predicting What Neural Networks Can Reason About

To explore some implications of our perspective, we next study the alignment of popular architectures to reasoning tasks with increasing complexity. Simpler tasks may serve as modules for more complex tasks. If an architecture aligns well with a task, we expect it to learn the task well. We then introduce a neural architecture design strategy for complex reasoning. The experiments in Section 5 empirically validate the findings in this section.

Single-object feature extraction. Given "disentangled" object representations $X = [h_1, h_2, ..., h_k]$, where each $h_i \in \mathbb{R}^{d_i}$ is a feature like color or shape, by Theorem 3.6, an MLP can provably learn to extract, and answer questions about, relevant features, *e.g.*, "What is the color of the cat?". Disentangled representations have also empirically shown good generalization [57, 41].

Corollary 4.1. Feature extraction functions $g_i : \mathbb{R}^d \to \mathbb{R}^{d_i}$, $g_i(X) = h_i$ are learnable with $O(\log(1/\delta)/\epsilon^2)$ samples by an MLP.

Summary statistics. Deep Sets, i.e., $\text{MLP}_2(\sum_{s\in S} \text{MLP}_1(X_s))$, align with summary statistics, because the MLP_1 can extract object features. In particular, they can *count*, *e.g.*, "How many white cats with blue eyes are there?", and learn *max or min* statistics by using smooth approximations like the softmax $\max_{s\in S} X_s \approx \log(\sum_{s\in X_s} \exp(X_s))$. For maxima, max pooling likely works better [38], because it aligns with the max even better than sum.

Pairwise relations. Deep Sets, however, do not align well with *pairwise* object relations, since those are not easily encoded by summing over all individual objects, i.e., via $\sum_{s\in S} \text{MLP}_1(X_s)$.

Claim 4.2. Suppose $g(x, y) = 0$ if and only if $x = y$, *e.g.*, $g(x, y) = (x - y)^2$. There is no f such that $g(x, y) = f(x) + f(y)$.

Hence, the MLP_2 in Deep Sets must learn the pairwise relations. Corollary 3.7 suggests this may indeed be more difficult for MLPs, as no network structure can be exploited. In contrast, GNNs model pairwise relations with $\sum_{s,t} \text{MLP}(X_s, X_t)$. By Corollary 4.1, $\text{MLP}(X_s, X_t)$ can learn to extract relevant features in $[X_s, X_t]$. Hence, *each GNN iteration* can reason about simple functions, *e.g.*, sum/max/min, over pairwise relations, *e.g.*, "Which two cats are the farthest apart?". In Section 5, we will see that indeed, Deep Sets fail to learn answering such questions, but GNNs learn well.

Higher-order relations. For relations between more than two objects, one may use MLPs that take in multiple objects [8, 35], generalizing from graphs to (directed) hypergraphs. We refer to such networks as *Hypergraph Relation Network* (HRN). An one-layer HRN with triplet input is defined as

$$\text{MLP}_2\big(\sum\nolimits_{s,t,l\in S} \text{MLP}_1\big(h_s^{(k-1)}, h_t^{(k-1)}, h_l^{(k-1)}\big)\big). \tag{4.1}$$

HRNs, however, are computationally expensive (e.g., $O(|S|^3)$ for triplets). Often, this expense is not needed: many complex relations can be *recursively reduced* to pairwise relations as we see next.

4.1 GNNs Can Perform Complex Reasoning through Dynamic Programming

GNNs structurally align with the powerful algorithm paradigm *dynamic programming (DP)* [10, 13]. DP solves a complicated problem by *recursively* breaking it down into simpler sub-problems, i.e.,

$$\text{Answer}[k][i] = \text{DP}(\{\text{Answer}[k-1][j]\}, j = 1...n), \tag{4.2}$$

where $\text{Answer}[k][i]$ denotes the answer to the sub-problem indexed by k and i, and DP is an algorithm-dependent update rule which obtains $\text{Answer}[k][i]$ by reducing it to $\text{Answer}[k-1][j]$'s. $\text{Answer}[k][i]$ can also depend on answers from step $k-2, k-3, ...$ by remembering answers from previous steps. Often, the updates $\text{DP}(\cdot)$ are fairly simple, *e.g.*, min/max/sum.

The structure of GNNs naturally matches that of DP. The GNN representation vectors $h_i^{(k)}$ correspond to $\text{Answer}[k][i]$, and the message passing update corresponds to the DP update. Theorem 3.5 suggests that a GNN can learn to simulate the underlying DP algorithm if it learns the DP update rules in each recursive step, and it has at least the same number of steps (depth) as the DP algorithm. Next, we show examples of reasoning tasks that can be solved by DP, and thus, by GNNs.

Relational question answering. A complex relational question can be answered through DP if we can recursively break it down to simpler relational questions. An example is "Which cat is the closest to the cat which is the closest to ...?" [37]. A DP solution would reduce the question of finding the k-hop closest cat to finding the cat that is closest to the $(k-1)$-hop closest cat as follows:

$$\text{closest}[1][i] = \arg\min_j \text{d}(i,j), \quad \text{closest}[k][i] = \text{closest}[k-1]\Big[\text{closest}[1][i]\Big] \text{ for } k > 1, \tag{4.3}$$

where $\text{closest}[k][i]$ is cat i's k-hop closest cat. If a GNN maintains $h_i^{(k)} = [\text{closest}[k][i], \text{closest}[1][i]]$, then the DP update rules, *i.e.*, finding the argmin of pairwise distances and conditionally copying another object's features, are learnable by an iteration of GNN.

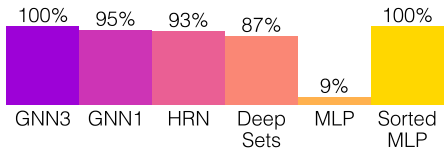
Shortest paths. Many known algorithms for the (single-source) shortest paths problem are DP in nature [14, 9]. An example is the Bellman-Ford algorithm [9], which recursively updates the minimum distance between each object u and the source object s within k steps:

$$\text{distance}[1][u] = \text{cost}(s,u), \quad \text{distance}[k][u] = \min_v \big\{\text{distance}[k-1][v] + \text{cost}(v,u)\big\}, \tag{4.4}$$

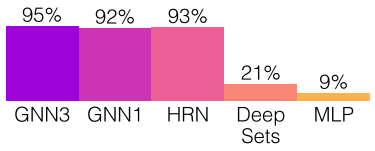
If a GNN maintains $h_u^{(k)} = [\text{distance}[k][u], X_u]$, where we need X_u if $\text{cost}(v,u) = \text{cost}(X_v, X_u)$ needs to be learned too, then the sum $\text{distance}[k-1][v] + \text{cost}(v,u)$ is simple to be learned by $\text{MLP}(h_v^{(k-1)}, h_u^{(k-1)})$. Moreover, since min pooling can be approximated by sum pooling as we have discussed, GNNs can provably learn the DP update, and thus, can reason about shortest paths.

Divide and conquer, greedy, sorting, and intuitive physics. We can formulate many more reasoning problems as roughly DP and solve them with GNNs. Divide and conquer and greedy algorithms [13] fall under the general framework of DP. Divide and conquer also breaks a problem into sub-problems, but it only combines answers to *non-overlapping* sub-problems. This simpler algorithmic paradigm already solves many challenging reasoning problems, such as the classic Tower of Hanoi puzzle. Greedy algorithms maintain a *single* optimum, whereas DP/GNNs maintain answers to many sub-problems. GNNs can learn to sort [8], because we can count how many objects are "smaller than" an object through pairwise comparisons. Moreover, a GNN can predict the trajectories of physical objects if it learns a simple law of physics as the DP update rule [40].

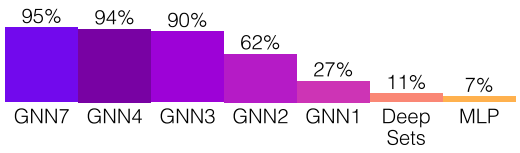
Discrete mathematics and theory. In fact, we could even reason about many profound questions in discrete mathematics and theoretical computer science (TCS) via DP, and thus, GNNs. Graph minor theory via DP gives one of the *deepest results* in discrete mathematics [27, 39]. While in TCS, DP is one of the best tools to obtain a polynomial time *approximation algorithm* for problems known to be NP-hard. For



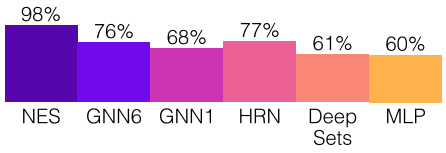
(a) Maximum value difference.



(b) Furthest pair.



(c) Dragon trainer.



(d) Subset sum. Random is 50%.

Figure 2: **Test accuracies on four abstract reasoning tasks with increasing complexity.** GNNk is GNN with k iterations. (a) Computing a summary statistic of the universe. All models except MLP generalize. Sorted MLP refers to an MLP with input treasures sorted by value. (b) Simple relational reasoning. Deep Sets and MLP fail. (c) Dynamic programming. Only GNNs with at least three iterations generalize. (d) An NP-hard problem. Even GNNs fail, and only NES generalizes.

example, DP for the traveling salesman problem (TSP) in Euclidean space gives one of the best known approximation algorithms [3, 33]. Thus, our theory also provides an explanation for the effectiveness (and hence popularity) of applying GNNs to solve NP-hard combinatorial problems [46, 28].

4.2 Neural Algorithms: a Neural Architecture Design Strategy

To reason about tasks that involve *more complicated operations*, *e.g.*, computing the maximum flow or solving a linear program [13], our framework suggests we shall *instantiate a similar structure in the network*. We name neural networks that follow this architecture design strategy *Neural Algorithms*. As one example, we apply the neural algorithm strategy to an NP-hard reasoning problem.

Neural Exhaustive Search. Given a set of numbers, the subset sum problem asks whether there exists a subset that sums to 0. Subset sum is NP-hard [26]. Although there is a pseudo-polynomial time DP solution [13], it requires maintaining all possible sum so far at each step, so a GNN would need *many more* representation vectors to store these answers. This may be difficult if a GNN always retains the same number of representation vectors. Otherwise, we can consider a simple exhaustive search strategy, where we compute the sum for all $2^{|S|}$ possible subsets τ and decide whether any of them is 0. This leads to a neural algorithm we name Neural Exhaustive Search (NES).

$$\text{MLP}_2(\max_{\tau \subseteq S} \text{MLP}_1 \circ \text{LSTM}(X_1, ..., X_{|\tau|} : X_1, ..., X_{|\tau|} \in \tau)). \tag{4.5}$$

NES structurally aligns with the exhaustive search algorithm if the LSTM learns to sum, MLP_1 learns to check whether its input is zero, and the max pooling layer summarizes this test for all subsets. In Section 5, we will see NES indeed solves subset sum, despite the exponential time complexity.

5 Experiments

We design four abstract reasoning tasks with increasing complexity. To separate reasoning from representation learning, we use disentangled object representations. **Data and training details are in Appendix H.** We test the following hypotheses derived from our framework:

MLPs do not align well with reasoning tasks (Corollary 3.7) and will not generalize well. **Deep Sets** can reason about summary statistics but not relations. **GNNs** generalize well on many reasoning tasks and can learn dynamic programming (§4.1), but may fail on NP-hard problems. **Hypergraph Relation Network (HRN)** (4.1) has similar generalization power as GNNs on many tasks, despite its extra time complexity. **Neural Exhaustive Search (NES)** (4.5) can solve NP-hard problems.

5.1 Fantastic Treasure: Relational Question Answering

The universe S has 25 fantastic treasures. We have the magic power to check the location $h_1 \in [-20..20]^8$, value $h_2 \in [0..100]$, and color $h_3 \in [1..6]$ of each treasure $X = [h_1, h_2, h_3]$. Treasures change over time, and we train networks on snapshots of the universe S_i to answer two questions.

Maximum value difference. The first question asks the difference in value between the most and the least valuable treasure. Formally, the answer is $y(S) = \max_{s\in S} h_2(X_s) - \min_{s\in S} h_2(X_s)$.

This question asks a summary statistic of the universe. We expect MLP to fail (Corollary 3.7) and other models to succeed. Fig. 2a confirms our prediction. Interestingly, MLP achieves perfect test accuracy when the input treasures are sorted by value (Sorted MLP in Fig. 2a). This observation is in line with our theory—when the treasures are sorted, the answer is reduced to a *simple* subtraction: $y(S) = h_2(X_{|S|}) - h_2(X_1)$, which MLP can learn (Theorem 3.6).

Our theory also explains why Deep Sets have lower test accuracy than GNNs. We can rewrite the answer as $y(S) = \max_{s_1,s_2}\{h_2(X_{s_1}) - h_2(X_{s_2})\}$. GNNs align with this equation and only need to learn two operations: max and subtraction. Deep Sets do not loop over pairs of objects and, therefore, must find the answer through more operations: max, min, and then subtraction. Finally, HRN is slightly behind GNNs, showing that ternary relations do not give additional gain for this task.

Furthest pair. Our second question asks the colors of the two treasures with the largest distance. The answer is a pair of colors, encoded as an integer category:

$$y(S) = (h_3(X_{s_1}), h_3(X_{s_2})) \quad \text{s.t. } \{X_{s_1}, X_{s_2}\} = \arg\max_{s_1,s_2\in S} \|h_1(X_{s_1}) - h_1(X_{s_2})\|_{\ell_1}$$

Unlike values, locations are *not totally ordered*, so the answer is not just a summary statistic of the universe and requires reasoning over pairwise relations. MLP and Deep Sets fail to generalize on this relational reasoning task (Fig. 2b), confirming Claim 4.2. HRN and GNNs work well with similar accuracies, suggesting again that ternary relations do not give additional gain for this task.

5.2 Dragon Trainer: Dynamic Programming

A dragon trainer lives in a world S with 10 dragons. Each dragon $X = [h_1, h_2]$ has a location $h_1 \in [0..10]^2$ and a unique combat level $h_2 \in [1..10]$. In each game, the trainer starts at a random location with level zero, $X_{\text{trainer}} = [p_0, 0]$, and receives a quest to defeat the level-k dragon. At each time step, the trainer can challenge any *more powerful* dragon X, with a cost equal to the product of the travel distance and the level difference $c(X_{\text{trainer}}, X) = \|h_1(X_{\text{trainer}}) - h_1(X)\|_{\ell_1} \times (h_2(X) - h_2(X_{\text{trainer}}))$. After defeating dragon X, the trainer's level upgrades to $h_2(X)$, and the trainer moves to $h_1(X)$. We ask the minimum cost of completing the quest, *i.e.*, defeating the level-k dragon.

We can solve the game with a DP algorithm similar to shortest paths (4.4), where the source is the trainer's starting location, and the target is the quest dragon. Our game is *more challenging than vanilla shortest paths* because the model also needs to learn the cost function c.

We train models on $200,000$ games to predict minimum cost in $[0..200]$. To make games challenging, we sample games whose optimal solution involves defeating three to seven non-quest dragons.

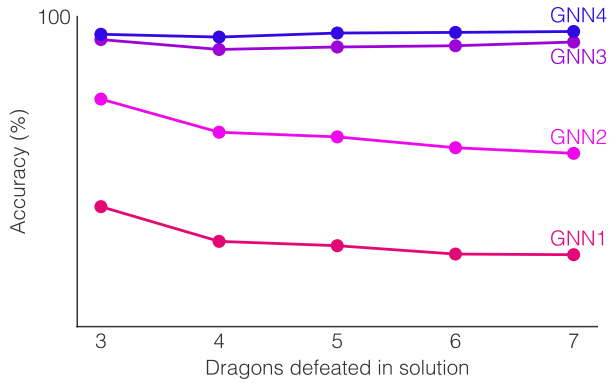


Figure 3: **Accuracy breakdown on dragon trainer.** Each dot indicates the accuracy of a model (y-axis) on games categorized by the number of defated dragons (x-axis) in the optimal strategy.

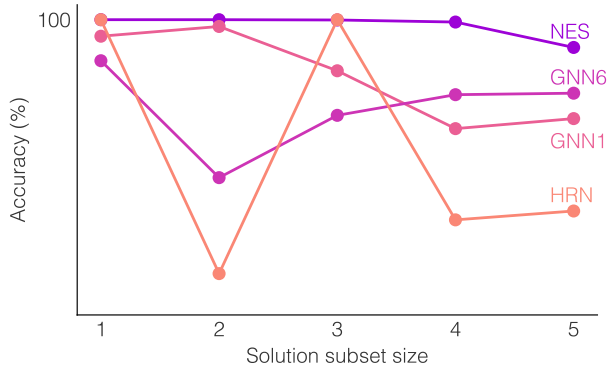


Figure 4: **Accuracy breakdown on subset sum.** Each dot indicates the accuracy of a model (y-axis) on questions where there exists a subset of (x-axis) elements which sum to zero.

Results. As expected, MLP, DeepSets, and one/two-iteration GNNs fail this complex game (Fig. 2c).

Surprisingly, a GNN with four iterations has almost the same test accuracy as a GNN with seven iterations. In our dataset, the optimal strategies for some games require defeating seven dragons, and the Bellman-Ford algorithm (4.4) needs at least seven iterations to solve these games. Therefore, the four-iteration GNN must have discovered a solution to shortest-paths that requires fewer iterations.

One possible DP solution is the following. To compute a shortest-path from a source object s to a target object t with at most seven stops, we run the following updates for four iterations:

$$\text{distance}_s[1][u] = \text{cost}(s,u), \qquad \text{distance}_s[k][u] = \min_v \left\{\text{distance}_s[k-1][v] + \text{cost}(v,u)\right\}, \tag{5.1}$$

$$\text{distance}_t[1][u] = \text{cost}(u,t), \qquad \text{distance}_t[k][u] = \min_v \left\{\text{distance}_t[k-1][v] + \text{cost}(u,v)\right\}. \tag{5.2}$$

Update (5.1) is identical to the Bellman-Ford algorithm (4.4), and $\text{distance}_s[k][u]$ is the shortest distance from s to u with at most k stops. Update (5.2) is a *reverse* Bellman-Ford algorithm, and $\text{distance}_t[k][u]$ is the shortest distance from u to t with at most k stops. After running (5.1) and (5.2) for k iterations, we can compute a shortest path with at most $2k$ stops by enumerating a mid-point and aggregating the results of the two Bellman-Ford algorithms:

$$\min_u \left\{\text{distance}_s[k][u] + \text{distance}_t[k][u]\right\}. \tag{5.3}$$

Thus, this alternative algorithm only needs *half of the iterations of Bellman-Ford*. Its structure also aligns with GNN—(5.1) and (5.2) are similar to GNN updates, and the final enumeration step (5.3) can be learned by GNN's last pooling layer. The interesting empirical finding aligns with our theory: a neural network can reason about a task if *there exists* a structurally matching algorithmic solution.

In Fig. 3, we compare the performance of GNNs on games of different complexity. The accuracies of single/two-iteration GNNs drop dramatically as the games become more challenging, *i.e.*, as the optimal strategy involves defeating more non-quest dragons. Indeed, as algorithmic alignment suggests, more than two GNN iterations are necessary to align with the iterations of a correct reasoning algorithm and find an optimal strategy for the game.

5.3 Subset Sum

Finally, we consider a classic NP-hard problem: our universe S has six integers $X_1, ..., X_6 \in [-200..200]$. We ask whether there is a subset of S that sums up to 0.

Results. The *neural algorithm design strategy* from §4.2 proves useful in this challenging task: Only Neural Exhaustive Search (NES) achieves a nearly perfect test accuracy (Fig. 2d), confirming again that neural networks generalize better when the network structure matches the algorithmic structure of a correct reasoning procedure. MLP and Deep Sets barely outperform random guessing (50%). GNNs and HRN generalize better, suggesting that they can learn to inspect a small number of subsets.

Fig. 4 shows the fine-grained accuracies for subset sum questions whose answers are yes, *i.e.*, there exists a solution subset whose elements sum to zero. Indeed, if there exist solution subsets of two elements, single-iteration GNN always identifies them and can perfectly answer these questions (Fig. 4). In contrast, HRN fails to identify solution subsets of size two, but succeeds if the solution contains three elements instead (Fig. 4). This empirical finding can be explained via our theory: Single-iteration GNN considers pairwise relations, and HRN considers ternary relations. Thus, by algorithmic alignment, they can easily learn to inspect the sum of every pair (subsets of size two) and triplet (subsets of size three), respectively.

6 Conclusion

This paper is an initial step towards formally understanding how neural networks can learn abstract reasoning. We introduce an algorithmic alignment perspective that may inspire architecture design, and opens up theoretical avenues. An interesting future direction is to design, e.g. via algorithmic alignment, networks that can learn more general abstract reasoning than GNNs.

Acknowledgments

This research was supported by NSF CAREER award 1553284, DARPA DSOs Lagrange program under grant FA86501827838 and a Chevron-MIT Energy Fellowship. This research was also supported by JST ERATO JPMJER1201 and JSPS Kakenhi JP18H05291. MZ was supported by DARPA award HR0011-15-C-0113 under subcontract to Raytheon BBN Technologies. The views, opinions, and/or findings contained in this article are those of the author and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

References

[1] Zeyuan Allen-Zhu, Yuanzhi Li, and Yingyu Liang. Learning and generalization in overparameterized neural networks, going beyond two layers. *arXiv preprint arXiv:1811.04918*, 2018.

[2] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *Proceedings of the IEEE international conference on computer vision*, pages 2425–2433, 2015.

[3] Sanjeev Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *Journal of the ACM (JACM)*, 45(5):753–782, 1998.

[4] Sanjeev Arora, Simon S Du, Wei Hu, Zhiyuan Li, and Ruosong Wang. Fine-grained analysis of optimization and generalization for overparameterized two-layer neural networks. In *International Conference on Machine Learning*, 2019.

[5] Peter L Bartlett and Shahar Mendelson. Rademacher and gaussian complexities: Risk bounds and structural results. *Journal of Machine Learning Research*, 3(Nov):463–482, 2002.

[6] Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. Spectrally-normalized margin bounds for neural networks. In *Advances in Neural Information Processing Systems*, pages 6240–6249, 2017.

[7] Peter Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, et al. Interaction networks for learning about objects, relations and physics. In *Advances in Neural Information Processing Systems*, pages 4502–4510, 2016.

[8] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

[9] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[10] Richard Bellman. Dynamic programming. *Science*, 153(3731):34–37, 1966.

[11] Michael Chang, Abhishek Gupta, Sergey Levine, and Thomas L. Griffiths. Automatically composing representation transformations as a means for generalization. In *International Conference on Learning Representations*, 2019.

[12] Michael B Chang, Tomer Ullman, Antonio Torralba, and Joshua B Tenenbaum. A compositional object-based approach to learning physical dynamics. In *International Conference on Learning Representations*, 2017.

[13] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[14] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1): 269–271, 1959.

[15] Gintare Karolina Dziugaite and Daniel M Roy. Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data. *arXiv preprint arXiv:1703.11008*, 2017.

[16] François Fleuret, Ting Li, Charles Dubout, Emma K Wampler, Steven Yantis, and Donald Geman. Comparing machines and humans on a visual categorization test. *Proceedings of the National Academy of Sciences*, 108(43):17621–17625, 2011.

[17] Katerina Fragkiadaki, Pulkit Agrawal, Sergey Levine, and Jitendra Malik. Learning visual predictive models of physics for playing billiards. In *International Conference on Learning Representations*, 2016.

[18] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pages 1273–1272, 2017.

[19] Noah Golowich, Alexander Rakhlin, and Ohad Shamir. Size-independent sample complexity of neural networks. In *Conference On Learning Theory*, pages 297–299, 2018.

[20] Felix Hill, Adam Santoro, David Barrett, Ari Morcos, and Timothy Lillicrap. Learning to make analogies by contrasting abstract relational structure. In *International Conference on Learning Representations*, 2019.

[21] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[22] Ronghang Hu, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko. Learning to reason: End-to-end module networks for visual question answering. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 804–813, 2017.

[23] Michael Janner, Sergey Levine, William T. Freeman, Joshua B. Tenenbaum, Chelsea Finn, and Jiajun Wu. Reasoning about physical interactions with object-centric models. In *International Conference on Learning Representations*, 2019.

[24] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2901–2910, 2017.

[25] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Judy Hoffman, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Inferring and executing programs for visual reasoning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2989–2998, 2017.

[26] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[27] Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Bruce Reed. The disjoint paths problem in quadratic time. *Journal of Combinatorial Theory, Series B*, 102(2):424–435, 2012.

[28] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

[29] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[31] Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40, 2017.

[32] Jiayuan Mao, Chuang Gan, Pushmeet Kohli, Joshua B. Tenenbaum, and Jiajun Wu. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. In *International Conference on Learning Representations*, 2019.

[33] Joseph SB Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric tsp, k-mst, and related problems. *SIAM Journal on computing*, 28(4):1298–1309, 1999.

[34] Damian Mrowca, Chengxu Zhuang, Elias Wang, Nick Haber, Li F Fei-Fei, Josh Tenenbaum, and Daniel L Yamins. Flexible neural representation for physics prediction. In *Advances in Neural Information Processing Systems*, pages 8799–8810, 2018.

[35] Ryan L Murphy, Balasubramaniam Srinivasan, Vinayak Rao, and Bruno Ribeiro. Janossy pooling: Learning deep permutation-invariant functions for variable-size inputs. In *International Conference on Learning Representations*, 2019.

[36] Behnam Neyshabur, Ryota Tomioka, and Nathan Srebro. Norm-based capacity control in neural networks. In *Conference on Learning Theory*, pages 1376–1401, 2015.

[37] Rasmus Palm, Ulrich Paquet, and Ole Winther. Recurrent relational networks. In *Advances in Neural Information Processing Systems*, pages 3368–3378, 2018.

[38] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 1 (2):4, 2017.

[39] Neil Robertson and Paul D Seymour. Graph minors. xiii. the disjoint paths problem. *Journal of combinatorial theory, Series B*, 63(1):65–110, 1995.

[40] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin Riedmiller, Raia Hadsell, and Peter Battaglia. Graph networks as learnable physics engines for inference and control. In *International Conference on Machine Learning*, pages 4467–4476, 2018.

[41] Adam Santoro, David Raposo, David G Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy Lillicrap. A simple neural network module for relational reasoning. In *Advances in neural information processing systems*, pages 4967–4976, 2017.

[42] Adam Santoro, Felix Hill, David Barrett, Ari Morcos, and Timothy Lillicrap. Measuring abstract reasoning in neural networks. In *International Conference on Machine Learning*, pages 4477–4486, 2018.

[43] David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. Analysing mathematical reasoning abilities of neural models. In *International Conference on Learning Representations*, 2019.

[44] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. Computational capabilities of graph neural networks. *IEEE Transactions on Neural Networks*, 20(1):81–102, 2009.

[45] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[46] Daniel Selsam, Matthew Lamm, Benedikt Bunz, Percy Liang, Leonardo de Moura, and David L Dill. Learning a SAT solver from single-bit supervision. In *International Conference on Learning Representations*, 2019.

[47] Elizabeth S Spelke and Katherine D Kinzler. Core knowledge. *Developmental science*, 10(1):89–96, 2007.

[48] Leslie G Valiant. A theory of the learnable. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 436–445. ACM, 1984.

[49] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013.

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[51] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. In *International Conference on Learning Representations*, 2015.

[52] Edward Wagstaff, Fabian B Fuchs, Martin Engelcke, Ingmar Posner, and Michael Osborne. On the limitations of representing functions on sets. In *International Conference on Machine Learning*, 2019.

[53] Nicholas Watters, Daniel Zoran, Theophane Weber, Peter Battaglia, Razvan Pascanu, and Andrea Tacchetti. Visual interaction networks: Learning a physics simulator from video. In *Advances in neural information processing systems*, pages 4539–4547, 2017.

[54] Jiajun Wu, Erika Lu, Pushmeet Kohli, Bill Freeman, and Josh Tenenbaum. Learning to see physics via visual de-animation. In *Advances in Neural Information Processing Systems*, pages 153–164, 2017.

[55] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning*, pages 5453–5462, 2018.

[56] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

[57] Kexin Yi, Jiajun Wu, Chuang Gan, Antonio Torralba, Pushmeet Kohli, and Josh Tenenbaum. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. In *Advances in Neural Information Processing Systems*, pages 1031–1042, 2018.

[58] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan R Salakhutdinov, and Alexander J Smola. Deep sets. In *Advances in Neural Information Processing Systems*, pages 3391–3401, 2017.

[59] Chi Zhang, Feng Gao, Baoxiong Jia, Yixin Zhu, and Song-Chun Zhu. Raven: A dataset for relational and analogical visual reasoning. *arXiv preprint arXiv:1903.02741*, 2019.

A Proof for Proposition 3.1

We will prove the universal approximation of GNNs by showing that GNNs have at least the same expressive power as Deep Sets, and then apply the universal approximation of Deep Sets for permutation invariant continuous functions.

Zaheer et al. [58] prove the universal approximation of Deep Sets under the restriction that the set size is fixed and the hidden dimension is equal to the set size plus one. Wagstaff et al. [52] extend the universal approximation result for Deep Sets by showing that the set size does not have to be fixed and the hidden dimension is only required to be at least as large as the set size. The results for our purposes can be summarized as follows.

Universal approximation of Deep Sets. Assume the elements are from a compact set in $\mathbb{R}^d$. Any continuous function on a set S of size bounded by N, *i.e.*, $f : \mathbb{R}^{d\times N} \to \mathbb{R}$, that is permutation invariant to the elements in S can be approximated arbitrarily close by some Deep Sets model with sufficiently large width and output dimension for its MLPs.

Next we show any Deep Sets can be expressed by some GNN with one message passing iteration. The computation structure of one-layer GNNs is shown below.

$$h_s = \sum_{t\in S} \phi\left(X_s, X_t\right), \quad h_S = g\left(\sum_{s\in S} h_s\right), \tag{A.1}$$

where ϕ and g are parameterized by MLPs. If ϕ is a function that ignores X_t so that $\phi\left(X_s, X_t\right) = \rho(X_s)$ for some ρ, *e.g.*, by letting part of the weight matricies in ϕ be 0, then we essentially get a Deep Sets in the following form.

$$h_s = \rho\left(X_s\right), \quad h_S = g\left(\sum_{s\in S} h_s\right). \tag{A.2}$$

For any such ρ, we can get the corresponding ϕ via the construction above. Hence for any Deep Sets, we can express it with an one-layer GNN. The same result applies to GNNs with multiple layers (message passing iterations), because we can express a function $\rho(X_s)$ by the composition of multiple $\rho^{(k)}$'s, which we can express with a GNN layer via our construction above. It then follows that GNNs are universal approximators for permutation invariant continuous functions.

B Proof for Proposition 3.2

For any GNN $\mathcal{N}$, we construct an MLP that is able to do the exact same computation as $\mathcal{N}$. It will then follow that the MLP can represent any function $\mathcal{N}$ can represent. Suppose the computation structure of $\mathcal{N}$ is the following.

$$h_s^{(k)} = \sum_{t\in S} f^{(k)}\left(h_s^{(k-1)}, h_t^{(k-1)}\right), \quad h_S = g\left(\sum_{s\in S} h_s^{(K)}\right), \tag{B.1}$$

where f and g are parameterized by MLPs. Suppose the set size is bounded by M (the expressive power of GNNs also depend on M [52]). We first show the result for a fixed size input, *i.e.*, MLPs can simulate GNNs if the input set has a fixed size, and then apply an ensemble approach to deal with variable sized input.

Let the input to the MLP be a vector concatenated by $h_s^{(0)}$'s, in some arbitrary ordering. For each message passing iteration of $\mathcal{N}$, any $f^{(k)}$ can be represented by an MLP. Thus, for each pair of $(h_t^{(k-1)}, h_s^{(k-1)})$, we

can set weights in the MLP so that the the concatenation of all $f(h_t^{(k-1)}, h_s^{(k-1)})$ become the hidden vector after some layers of the MLP. With the vector of $f(h_t^{(k-1)}, h_s^{(k-1)})$ as input, in the next few layers of the MLP we can construct weights so that we have the concatenation of $h_s^{(k)} = \sum_{t \in S} f^{(k)} \left(h_s^{(k-1)}, h_t^{(k-1)} \right)$ as the result of the hidden dimension, because we can encode summation with weights in MLPs. So far, we can simulate an iteration of GNN $\mathcal{N}$ with layers of MLP. We can repeat the process for K times by stacking the similar layers. Finally, with a concatenation of $h_s^{(K)}$ as our hidden dimension in the MLP, similarly, we can simulate $h_S = g\left(\sum_{s \in S} h_s^{(K)} \right)$ with layers of MLP. Stacking all layers together, we have obtained an MLP that can simulate $\mathcal{N}$.

To deal with variable sized inputs, we construct M MLPs that can simulate the GNN for each input set size $1, ..., M$. Then we construct a meta-layer, whose weights represent (universally approximate) the summation of the output of M MLPs multiplied by an indicator function of whether each MLPs has the same size as the set input (these need to be input information). The meta layer weights on top can then essentially select the output from of MLP that has the same size as the set input and then exactly simulate the GNN. Note that the MLP we construct here has the requirement for how we input the data and the information of set sizes etc. In practice, we can have M MLPs and decide which MLP to use depending on the input set size.

C Proof for Theorem 3.5

We will show the learnability result by an inductive argument. Specifically, we will show that under our setting and assumptions, the error between the learned function and correct function on the test set will not blow up after the transform of another learned function $\hat{f}_j$, assuming learnability on previous $\hat{f}_1, ..., \hat{f}_{j-1}$ by induction. Thus, we can essentially provably learn at all layers/iterations and eventually learn g.

Suppose we have performed the sequential learning. Let us consider what happens at the test time. Let f_j be the *correct functions* as defined in the match of structure assumption. Let $\hat{f}_j$ be the functions learned by algorithm $\mathcal{A}_j$ and MLP $\mathcal{N}_j$. We have input $S \sim \mathcal{D}$, and our goal is to bound $\|g(S) - \hat{g}(S)\|$ with high probability. To show this, we bound the error of the intermediate representation vectors, *i.e.*, the output of $\hat{f}_j$ and f_j, and thus, the input to $\hat{f}_{j+1}$ and f_{j+1}.

Let us first consider what happens for the first MLP $\mathcal{N}_1$. f_1 and $\hat{f}_1$ have the same input distribution $x \sim \mathcal{D}$, where x are obtained from S, *e.g.*, the pairwise object representations as in (2.1). Hence, by the learnability assumption on $\mathcal{A}_1$ (match of structures assumption), $\|f_1(x) - \hat{f}_1(x)\| < \epsilon$ with probability at least $1 - \delta$. The error for the input of $\mathcal{N}_2$ is then $O(\epsilon)$ with failure probability $O(\delta)$, because there are a constant number of terms of aggregation of f_1's output, and we can apply union bound to upper bound the failure probability.

Next, we proceed by induction. Let us fix a k. Let z denote the input for f_k, which are generated by the previous f_j's, and let $\hat{z}$ denote the input for $\hat{f}_k$, which are generated by the previous $\hat{f}_j$'s. Assume $\|z - \hat{z}\| \leq O(\epsilon)$ with failure probability at most $O(\delta)$. We aim to show that this holds for $k + 1$. For the simplicity of notation, let f denote the correct function f_k and let $\hat{f}$ denote the learned function $\hat{f}_k$. Since there are a constant number of terms for aggregation, our goal is then to bound $\|\hat{f}(\hat{z}) - f(z)\|$. By triangle inequality, we have

$$\|\hat{f}(\hat{z}) - f(z)\| = \|\hat{f}(\hat{z}) - \hat{f}(z) + \hat{f}(z) - f(z)\| \tag{C.1}$$

$$\leq \|\hat{f}(\hat{z}) - \hat{f}(z)\| + \|\hat{f}(z) - f(z)\| \tag{C.2}$$

We can bound the first term with the Lipschitzness assumption of $\hat{f}$ as the following.

$$\|\hat{f}(\hat{z}) - \hat{f}(z)\| \leq L_1 \|\hat{z} - z\| \tag{C.3}$$

To bound the second term, our key insight is that f is a *learnale correct function*, so by the learnability assumption (match of structures assumption), it is close to the function $\tilde{f}$ learned by the MLP learning algorithm $\mathcal{A}$ on the *correct samples*, *i.e.*, f is close to $\tilde{f} = \mathcal{A}(\{z_i, y_i\})$. Moreover, $\hat{f}$ is generated by the MLP learning algorithm $\mathcal{A}$ on the perturbed samples, *i.e.*, $\hat{f} = \mathcal{A}(\{\hat{z}_i, y_i\})$. By the algorithm stability assumption, $\hat{f}$ and $\tilde{f}$ should be close if the input samples are only slightly perturbed. It then follows that

$$\|\hat{f}(z) - f(z)\| = \|\hat{f}(z) - \tilde{f}(z) + \tilde{f}(z) - f(z)\| \tag{C.4}$$

$$\leq \|\hat{f}(z) - \tilde{f}(z)\| + \|\tilde{f}(z) - f(z)\| \tag{C.5}$$

$$\leq L_0 \max_i \|z_i - \hat{z}_i\| + \epsilon \quad \text{w.p. } \geq 1 - \delta \tag{C.6}$$

where z_i and $\hat{z}_i$ are the training samples at the same layer k. Here, we apply the same induction condition as what we had for z and $\hat{z}$: $\|z_i - \hat{z}_i\| \leq O(\epsilon)$ with failure probability at most $O(\delta)$. We can then apply union bound to bound the probability of any bad event happening. Here, we have 3 bad events each happening with probability at most $O(\delta)$. Thus, with probability at least $1 - O(\delta)$, we have

$$\|\hat{f}(\hat{z}) - f(z)\| \leq L_1 O(\epsilon) + L_0 O(\epsilon) + \epsilon = O(\epsilon) \tag{C.7}$$

This completes the proof.

D Proof for Theorem 3.6

Theorem 3.6 is a generalization of Theorem 6.1 in [4], which addresses the scalar case. See [4] for a complete list of assumptions.

Theorem D.1. [4] Suppose we have $g : \mathbb{R}^d \to \mathbb{R}$, $g(x) = \sum_j \alpha_j \left(\beta_j^\top x\right)^{p_j}$, where $\beta_j \in \mathbb{R}^d$, $\alpha \in \mathbb{R}$, and $p_j = 1$ or $p_j = 2l\,(l \in \mathbb{N}_+)$. Let $\mathcal{A}$ be an overparameterized two-layer MLP that is randomly initialized and trained with gradient descent for a sufficient number of iterations. The sample complexity $\mathcal{C}_\mathcal{A}(g, \epsilon, \delta)$ is $O\Big(\frac{\sum_j p_j |\alpha_j| \cdot \|\beta_j\|_2^{p_j} + \log(1/\delta)}{\epsilon^2}\Big)$.

To extend the sample complexity bound to vector-valued functions, we view each entry/component of the output vector as an independent scalar-valued output. We can then apply a union bound to bound the error rate and failure probability for the output vector, and thus, bound the overall sample complexity.

Let ϵ and δ be the given error rate and failure probability. Moreover, suppose we choose some error rate ϵ_0 and failure probability δ_0 for the output/function of each entry. Applying Theorem D.1 to each component

$$g(x)^{(i)} = \sum_j \alpha_j^{(i)} \left(\beta_j^{(i)\top} x\right)^{p_j^{(i)}} =: g_i(x) \tag{D.1}$$

yields a sample complexity bound of

$$\mathcal{C}_\mathcal{A}(g_i, \epsilon_0, \delta_0) = O\left(\frac{\sum_j p_j^{(i)} |\alpha_j^{(i)}| \cdot \|\beta_j^{(i)}\|_2^{p_j^{(i)}} + \log(1/\delta_0)}{\epsilon_0^2}\right) \tag{D.2}$$

for each $g_i(x)$. Now let us bound the overall error rate and failure probability given ϵ_0 and δ_0 for each entry. The probability that we fail to learn each of the g_i is at most δ_0. Hence, by a union bound, the probability that we fail to learn any of the g_i is at most $m \cdot \delta_0$. Thus, with probability at least $1 - m\delta_0$, we successfully learn all g_i for $i = 1, ..., m$, so the error for every entry is bounded by ϵ_0. The error for the vector output is then at most $\sum_{i=1}^m \epsilon_0 = m\epsilon_0$.

Setting $m\delta_0 = \delta$ and $m\epsilon_0 = \epsilon$ gives us $\delta_0 = \frac{\delta}{m}$ and $\epsilon_0 = \frac{\epsilon}{m}$. Thus, if we can successfully learn the function for each output entry independently with error ϵ/m and failure rate δ/m, we can successfully learn the entire vector-valued function with rate ϵ and δ. This yields the following overall sample complexity bound:

$$\mathcal{C}_{\mathcal{A}}(g, \epsilon, \delta) = O\left(\frac{\max_i \sum_j p_j^{(i)} |\alpha_j^{(i)}| \cdot \|\beta_j^{(i)}\|_2^{p_j^{(i)}} + \log(m/\delta)}{(\epsilon/m)^2}\right) \tag{D.3}$$

Regarding m as a constant, we can further simplify the sample complexity to

$$\mathcal{C}_{\mathcal{A}}(g, \epsilon, \delta) = O\left(\frac{\max_i \sum_j p_j^{(i)} |\alpha_j^{(i)}| \cdot \|\beta_j^{(i)}\|_2^{p_j^{(i)}} + \log(1/\delta)}{\epsilon^2}\right). \tag{D.4}$$

E Proof for Corollary 3.7

Our main insight is that a giant MLP learns the same function $(X_i - X_j)^2$ for ℓ^2 times and encode them in the weights. This leads to the $O(\ell^2)$ extra sample complexity through Theorem 3.6, because the number of polynomial terms $(X_i - X_j)^2$ is of order ℓ^2.

First of all, the function $f(x, y) = (x - y)^2$ can be expressed as the following polynomial.

$$(x - y)^2 = \left([1 \ -1]^\top [x \ y]\right)^2 \tag{E.1}$$

We have $\beta = [1 - 1]$, so $p \cdot \|\beta\|^p = 4$. Hence, by Theorem 3.6, it takes $O(\frac{\log(1/\delta)}{\epsilon^2})$ samples for an MLP to learn $f(x, y) = (x - y)^2$. Under the sequential training setting, an one-layer GNN applies an MLP to learn f, and then sums up the outcome of $f(X_i, X_j)$ for all pairs X_i, X_j. Here, we essentially get the aggregation error $O(\ell^2 \cdot \epsilon)$ from ℓ^2 pairs. However, we will see that applying an MLP to learn g will also incur the same aggregation error. Hence, we do not need to consider the aggregation error effect when we compare the sample complexities.

Now we consider using MLP to learn the function g. No matter in what order the objects X_i are concatenated, we can express g with the sum of polynomials as the following.

$$g(S) = \sum_{ij} (\beta_{ij}^\top [X_1, ..., X_n])^2, \tag{E.2}$$

where β_{ij} has 1 at the i-th entry, -1 at the j-th entry and 0 elsewhere. Hence $\|\beta_{ij}\|^p \cdot p = 4$. It then follows from Theorem 3.6 and union bound that it takes $O((\ell^2 + \log(1/\hat{\delta}))/\hat{\epsilon}^2)$ to learn g, where $\hat{\epsilon} = \ell^2\epsilon$ and $\hat{\delta} = \ell^2\delta$. Here, as we have discussed above, the same aggregation error $\hat{\epsilon}$ occurs in the aggregation process of f, so we can simply consider $\hat{\epsilon}$ for both. Thus, comparing $O(\log(1/\hat{\delta})/\hat{\epsilon}^2)$ and $O((\ell^2 + \log(1/\hat{\delta}))/\hat{\epsilon}^2)$ gives us the $O(\ell^2)$ difference.

F Proof for Corollary 4.1

The main proof idea is that, any object feature $h_i = X_{S_i}$ is embedded in a subspace indexed by a subset S_i. Hence, the feature extractor function $g_i(X) = X_{S_i}$ can be represented as a linear function of X, whose coefficients depend on S_i, *i.e.*, which coordinates of X encode the feature h_i.

We can obtain each output coordinate of $g_i(X)^{(j)}$ with the following function.

$$g_i(X)^{(j)} = X_{S_i^{(j)}} = \beta_i^{(j)\top} X, \tag{F.1}$$

where $\beta_i^{(j)} \in \mathbb{R}^d$ has 1 at the $S_i^{(j)}$-th entry and 0 otherwise, and $S_i^{(j)}$ is the j-th element in S_i. Thus, we have $\|\beta_i^{(j)}\|_2 = 1$ for any i and j. Suppose the length of each object feature $d_i = |S_i|$ is some constant. It then follows from Theorem 3.6 that

$$\mathcal{C}_{\mathcal{A}}(g_i, \epsilon, \delta) = O\left(\frac{\log\left(d_i/\delta\right)}{\left(\epsilon/d_i\right)^2}\right) = O\left(\frac{\log\left(1/\delta\right)}{\epsilon^2}\right) \tag{F.2}$$

G Proof for Claim 4.2

We prove the claim by contradiction. Suppose there exists f such that $f(x) + f(y) = g(x, y)$ for any x and y. This implies that for any x, we have $f(x) + f(x) = g(x, x) = 0$. It follows that $f(x) = 0$ for any x. Now consider some x and y so that $x \neq y$. We must have $f(x) + f(y) = 0 + 0 = 0$. However, $g(x, y) \neq 0$ because $x \neq y$. Hence, there exists x and y so that $f(x) + f(y) \neq g(x, y)$. We have reached a contradiction.

H Experiments: Data and Training Details

H.1 Fantastic Treasure: Maximum Value Difference

Dataset generation. In the dataset, we sample $50,000$ training data, $5,000$ validation data, and $5,000$ test data. For each model, we report the test accuracy with the hyperparameter setting that achieves the best validation accuracy. In each training sample, the input universe consists of 25 treasures $X_1, ..., X_{25}$. For each treasure X_i, we have $X_i = [h_1, h_2, h_3]$, where the location h_1 is sampled uniformly from $[0..20]^8$, the value h_2 is sample uniformly form $[0..100]$, and the color h_3 is sampled uniformly from $[1..6]$. The task is to answer what the difference is in value between the most and least valuable treasure. We generate the answer label y for a universe S as follows: we find the the maximum difference in value among all treasures and set it to y. Then we make the label y into one-hot encoding with $100 + 1 = 101$ classes.

Hyperparameter setting. We train all models with the Adam optimizer, with learning rate from $1e-3, 5e-4$, and $1e-4$, and we decay the learning rate by 0.5 every 50 steps. We use cross-entropy loss. We train all models for 150 epochs. We tune batch size of 128 and 64. We apply weight decay of $1e-5$ for all models. For the MLP model, we choose the number of of hidden layers from 4 and 8. For models other than MLP, we set the number of hidden layers of the last MLP, *i.e.*, MLP_2, to 4. For models other than MLP, we choose the number of hidden layers of the MLPs prior to the last MLP, *i.e.*, MLP_1, from 3 and 4. For the MLP model, we set the hidden dimension to 256. For all models, we choose the hidden dimension of all MLPs from 128 and 256. Moreover, dropout with rate 0.5 is applied before the last two hidden layers of MLP_1, *i.e.*, the last MLP module in all models. Dropout with rate 0.5 is also applied before the last two hidden layers of the MLP model.

H.2 Fantastic Treasure: Furthest Pair

Dataset generation. In the dataset, we sample $60,000$ training data, $6,000$ validation data, and $6,000$ test data. For each model, we report the test accuracy with the hyperparameter setting that achieves the best validation accuracy. In each training sample, the input universe consists of 25 treasures $X_1, ..., X_{25}$. For each treasure X_i, we have $X_i = [h_1, h_2, h_3]$, where the location h_1 is sampled uniformly from $[0..20]^8$, the value h_2 is sample uniformly form $[0..100]$, and the color h_3 is sampled uniformly from $[1..6]$. The task is to answer what are the colors of the two treasure that are the most distant from each other. We generate the

answer label y for a universe S as follows: we find the pair of treasures that are the most distant from each other, say (X_i, X_j). Then we order the pair $(h_3(X_i), h_3(X_j))$ to obtain an ordered pair (a, b) with $a \leq b$ (aka. $a = \min\{h_3(X_i), h_3(X_j)\}$ and ($b = \max\{h_3(X_i), h_3(X_j)\}$), where $h_3(X_i)$ denotes the color of X_i. Then we compute the label y from (a, b) by counting how many valid pairs of colors are smaller than (a, b) (a pair (k, l) is smaller than (a, b) iff i). $k < a$ or ii). $k = a$ and $l < b$). The label y is one-hot encoding of the minimum cost with $6 \times (6-1)/2 + 6 = 21$ classes.

Hyperparameter setting. We train all models with the Adam optimizer, with learning rate from $1e-3, 5e-4$, and $1e-4$, and we decay the learning rate by 0.5 every 50 steps. We use cross-entropy loss. We train all models for 150 epochs. We tune batch size of 128 and 64. We apply weight decay of $1e-5$ for all models. For the MLP model, we choose the number of of hidden layers from 4 and 8. For models other than MLP, we set the number of hidden layers of the last MLP, *i.e.*, MLP_2, to 4. For models other than MLP, we choose the number of hidden layers of the MLPs prior to the last MLP, *i.e.*, MLP_1, from 3 and 4. For the MLP model, we set the hidden dimension to 256. For all models, we choose the hidden dimension of all MLPs from 128 and 256. Moreover, dropout with rate 0.5 is applied before the last two hidden layers of MLP_1, *i.e.*, the last MLP module in all models. Dropout with rate 0.5 is also applied before the last two hidden layers of the MLP model.

H.3 Dragon Trainer

Dataset generation. In the dataset, we sample $200,000$ training data, $6,000$ validation data, and $6,000$ test data. For each model, we report the test accuracy with the hyperparameter setting that achieves the best validation accuracy. In each training sample, the input universe consists of the trainer and 10 dragons $X_0, ..., X_{10}$, and the request level k, *i.e.*, we need to challenge dragon k. We have $X_i = [h_1, h_2]$, where $h_1 = i$ indicates the combat level, and the location $h_2 \in [0..10]^2$ is sampled uniformly from $[0..10]^2$. We generate the answer label y for a universe S as follows. We implement a shortest path algorithm to compute the minimum cost from the trainer to dragon k, where the cost is defined in Section 5. Then the label y is a one-hot encoding of minimum cost with 200 classes. Moreover, when we sample the data, we apply rejection sampling to ensure that the minimum cost's shortest path is of length $3, 4, 5, 6, 7$ with equal probability. That is, we eliminate the trivial questions.

Hyperparameter setting. We train all models with the Adam optimizer, with learning rate from $2e-4$ and $5e-4$, and we decay the learning rate by 0.5 every 50 steps. We use cross-entropy loss. We train all models for 300 epochs. We tune batch size of 128 and 64. We apply weight decay of $1e-5$ for all models. For the MLP model, we choose the number of layers from 4 and 8. For models other than MLP, we set the number of hidden layers of the last MLP, *i.e.*, MLP_2, to 4. For models other than MLP, we choose the number of hidden layers of the MLPs prior to the last MLP, *i.e.*, MLP_1, from 3 and 4. For the MLP model, we set the hidden dimension to 256. For all models, we choose MLP hidden dimensions from 128 and 256. Moreover, dropout with rate 0.5 is applied before the last two hidden layers of MLP_1, *i.e.*, the last MLP module in all models. Dropout with rate 0.5 is also applied before the last two hidden layers of the MLP model.

H.4 Subset Sum

Dataset generation. In the dataset, we sample $40,000$ training data, $4,000$ validation data, and $4,000$ test data. For each model, we report the test accuracy with the hyperparameter setting that achieves the best validation accuracy. In each training sample, the input universe S consists of 6 numbers $X_1, ..., X_6$, where each X_i is uniformly sampled from [-200..200]. The goal is to decide if there exists a subset that sums up to 0. In the data generation, we carefully decrease the number of questions that have trivial answers: 1)we control the number of samples where $0 \in \{X_1, ..., X_6\}$ to be around 1% of the total training data; 2) we further control the number of samples where $X_1 + ... + X_6 = 0$ or $\exists i, j \in [1..6]$ so that $X_i = -X_j$ to be around 1.5% of the total training data. In addition, we apply rejection sampling to make sure that the questions with answer yes (aka. such subset exists) and answer no (aka. no such subset exists) are balanced (*i.e.*, 20,000 samples for each class in the training data).

Hyperparameter setting. We train all models with the Adam optimizer, with learning rate from $1e-3, 5e-4$, and $1e-4$, and we decay the learning rate by 0.5 every 50 steps. We use cross-entropy loss. We train all models for 300 epochs. The batch size we use for all models is 64. We apply weight decay of $1e-5$ for all models. For the MLP model, we set the number of of hidden layers to 6. For DeepSets, we set the number of of hidden layers of the last MLP, *i.e.*, MLP_2, to 6 and the number of hidden layers of the MLPs prior to the last MLP, *i.e.*, MLP_1, to 4. For models other than MLP and DeepSets, we set the number of hidden layers of the last MLP, *i.e.*, MLP_2, to 4. For models other than MLP and DeepSets, we set the number of hidden layers of the MLPs prior to the last MLP, *i.e.*, MLP_1, to 3. For all models, we choose the hidden dimension of all MLPs from 128 and 256. Moreover, dropout with rate 0.5 is applied before the last two hidden layers of MLP_1, *i.e.*, the last MLP module in all models. Dropout with rate 0.5 is also applied before the last two hidden layers of the MLP model.

The model Neural Exhaustive Search (NES) enumerates all possible non-empty subsets τ of S, and passes the numbers of τ to an MLP, in a random order, to obtain the hidden feature. The hidden feature is then passed to a single-direction one-layer LSTM of hidden dimension 128. Afterwards, NES applies an aggregation function to these $2^6 - 1$ hidden states obtained by the LSTM to obtain the final output. For NES, we set the number of hidden layers of the last MLP, *i.e.*, MLP_2, to 4, the number of hidden layers of the MLPs prior to the last MLP, *i.e.*, MLP_1, to 3, and we choose the hidden dimension of all MLPs from 128 and 256.