MULTISET-EQUIVARIANT SET PREDICTION WITH APPROXIMATE IMPLICIT DIFFERENTIATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Most set prediction models in deep learning use set-equivariant operations, but they actually operate on multisets. We show that set-equivariant functions cannot represent certain functions on multisets, so we introduce the more appropriate notion of *multiset-equivariance*. We identify that the existing Deep Set Prediction Network (DSPN) can be multiset-equivariant *without being hindered by set-equivariance* and improve it with approximate implicit differentiation, allowing for better optimization while being faster and saving memory. In a range of toy experiments, we show that the perspective of multiset-equivariance is beneficial and that our changes to DSPN achieve better results in most cases. On CLEVR object property prediction, we substantially improve over the state-of-the-art Slot Attention from 8% to 77% in one of the strictest evaluation metrics because of the benefits made possible by implicit differentiation.

1 INTRODUCTION

An example of *set prediction* is detecting the objects in an image: we want a collection of objects as output, but there is no inherent order to them. Multisets and sets are natural representations for such data due to their orderless nature, with the difference between them being whether duplicate elements are allowed or not. In general, they are useful for representing collections of *discrete and composable elements*, such as agents in an environment, instruments in a piece of music, or atoms in a molecule.

The usual approach to representing a multiset or set of vectors in memory is to store it as a list in which the elements are placed in an arbitrary order. To still treat the list as if it were a multiset or set, recent set prediction models avoid being affected by this arbitrary order by using list-to-list functions that are enforced to be *permutation-equivariant* (Zaheer et al., 2017): changing the order of the input elements should change the order of the output elements in the same way.

Now, consider a function that takes a multiset (represented as a list) of two scalars as input and "pushes them apart":

$$\texttt{push_apart}([a, b]) = \begin{cases} [a-1, b+1] & \text{if } a \leq b \\ [a+1, b-1] & \text{otherwise} \end{cases} \tag{1}$$

For example, $\texttt{push_apart}([1, 2]) = [0, 3]$ and $\texttt{push_apart}([2, 1]) = [3, 0]$. At first glance, it appears that this function is permutation-equivariant because swapping the inputs swaps the outputs. However, notice that $\texttt{push_apart}([0, 0]) = [-1, 1]$. Swapping the inputs has no effect in this case, so the outputs remain unchanged, thus $\texttt{push_apart}$ is not permutation-equivariant.

This is a bit strange – the function only violates the equivariance property for equal input elements, in which case the order of the equal input elements is irrelevant anyway. How can we reconcile this observation with the standard definition of permutation-equivariance? We aim to address this question in our paper and present two key ideas:

1. **Multiset-equivariance.** We begin Section 2 by making the important observation that many existing models for set prediction thought to operate on sets actually operate on multisets. We then show that these models are too restricted in the functions they can represent on multisets due to the limitation of being permutation-equivariant, which we call *set-equivariance*. This is a problem even when there are no exact duplicate elements. Thus, we develop a new notion of *multiset*-equivariance, which we argue is a more suitable property for operations on multisets.

Table 1: Difference between equivariance properties in the context of multisets. Only when satisfying *exclusive multiset-equivariance* (this paper) is it enforced that order does not matter while also allowing equal input elements to be mapped to different output elements like in `push_apart`.

f is set-equivariant	f is multiset-equivariant	$f([\boldsymbol{a}, \boldsymbol{b}]) = [\boldsymbol{c}, \boldsymbol{d}]$ $\Leftrightarrow f([\boldsymbol{b}, \boldsymbol{a}]) = [\boldsymbol{d}, \boldsymbol{c}]$	$f([\boldsymbol{a}, \boldsymbol{a}]) = [\boldsymbol{c}, \boldsymbol{d}]$
✓	✓	guaranteed ✓	not possible ✗
✗	✗	not guaranteed ✗	possible ✓
✗	✓	guaranteed ✓	possible ✓

To avoid the aforementioned limitation of set-equivariance, a crucial point is that it is possible to be multiset-equivariant without being set-equivariant, which we call *exclusive multiset-equivariance*. We identify and show that deep set prediction networks (DSPN) (Zhang et al., 2019; Huang et al., 2020) satisfy this property with a specific choice of set encoder. This makes it the only set prediction model we are aware of that can satisfy exclusive multiset-equivariance.

2. **Implicit DSPN.** Despite this beneficial property, DSPN is outperformed by the set-equivariant Slot Attention (Locatello et al., 2020), which motivates us to improve other aspects of DSPN. We propose *implicit DSPN* (iDSPN) in Section 3: a version of DSPN that uses approximate implicit differentiation. Implicit differentiation enables us to use better optimizers and more iterations at a constant memory cost and less computation time. The approximation makes this faster to run and easier to implement by avoiding the computation of a Hessian while still having the same benefits.

In our experiments in Section 4, we start by evaluating the first point by comparing exclusively multiset-equivariant models with both set-equivariant and non-equivariant models on synthetic data. The former completely fail on this simple toy task while the latter are significantly less sample-efficient. Next, we evaluate the second point by testing the modeling capacity on autoencoding random sets, where iDSPN performs similarly to DSPN at the same iteration count and much better for the same computational cost. Lastly, iDSPN significantly raises the bar on CLEVR object property prediction, outperforming the state-of-the-art Slot Attention (Locatello et al., 2020) by a large margin of 69 percentage points on the $\text{AP}_{0.125}$ metric while only training for 7% of the number of epochs.

2 Limitations of Set-Equivariance

In this section, we begin by identifying problems that limit the expressivity of a wide class of set prediction models satisfying a type of permutation-equivariance, *set-equivariance*. We argue that set-equivariance is too strict and instead propose a relaxation of set-equivariance that is more suitable for set prediction models: *multiset-equivariance*. We end by discussing how an existing method, deep set prediction networks (Zhang et al., 2019; Huang et al., 2020), is *multiset*-equivariant but not *set*-equivariant and the associated benefits thereof (Table 1).

2.1 Preliminaries

Sets in the context of deep learning are typically unordered collections of real-valued vectors. They are represented in memory as $n \times d$ matrices with the n set elements in an arbitrary order, each element being a d-dimensional vector. Since elements in the matrix representation are typically not restricted to be unique, the term *multiset* is more appropriate. To still treat these matrices as multisets, operations must not rely on the arbitrary order of the elements. The two main properties to accomplish this are permutation-*invariance* and permutation-*equivariance* (Zaheer et al., 2017). In short, when changing the order of the elements in the matrix, the output of a function should either not change at all (invariance) or change order similarly (equivariance).

2.2 Set-Equivariant Models

Recent set prediction models (Lee et al., 2019; Locatello et al., 2020; Carion et al., 2020; Kosiorek et al., 2020) make use of set-to-set (permutation-equivariant list-to-list) functions to refine an initial set $\boldsymbol{Y}_0$, which is usually a randomly generated or learnable matrix. Permutation-equivariance is

desirable when processing sets because it prevents a function from relying on the arbitrary order of the set in its matrix representation. Such functions can be easily composed to build larger models that remain equivariant, which fits well into deep learning architectures as building blocks.

However, without having any restrictions on duplicate elements, it is more accurate to think of set prediction models as using multiset-to-multiset functions. This difference subtly changes the appropriate notion of equivariance for sets versus multisets, which will become important for our discussion on their limitations.

Set-equivariance. First, we generalize the standard notion of permutation-equivariance for sets to *multiset inputs* and refer to it as *set-equivariance*. This directly matches the usual definition of permutation-equivariance in the context of *set inputs* in the literature. Therefore, many existing models already satisfy the following definition, including Slot Attention (Locatello et al., 2020), Set Transformers (Lee et al., 2019), TSPN (Kosiorek et al., 2020), and the equivariant version of DeepSets (Zaheer et al., 2017).

Definition 1 (Set-equivariance for multiset inputs)**.** A function $f : \mathbb{R}^{n \times d_1} \to \mathbb{R}^{n \times d_2}$ with multiset input $\boldsymbol{X}$ is set-equivariant iff for any permutation matrix $\boldsymbol{P}$,

$$f(\boldsymbol{P}\boldsymbol{X}) = \boldsymbol{P}f(\boldsymbol{X}) \tag{2}$$

A consequence is that elements in a multiset that are equal before a set-equivariant function must remain equal after applying the function, as shown by the following:

Proposition 1. *For all permutation matrices $\boldsymbol{P}_1, \boldsymbol{P}_2$ such that $\boldsymbol{P}_1\boldsymbol{X} = \boldsymbol{P}_2\boldsymbol{X}$, a set-equivariant function f satisfies:*

$$\boldsymbol{P}_1 f(\boldsymbol{X}) = \boldsymbol{P}_2 f(\boldsymbol{X}) \tag{3}$$

Proof.

$$\boldsymbol{P}_1 f(\boldsymbol{X}) = f(\boldsymbol{P}_1\boldsymbol{X}) = f(\boldsymbol{P}_2\boldsymbol{X}) = \boldsymbol{P}_2 f(\boldsymbol{X}) \tag{4}$$

□

$\boldsymbol{P}_1\boldsymbol{X} = \boldsymbol{P}_2\boldsymbol{X}$ can be satisfied by $\boldsymbol{P}_1 \neq \boldsymbol{P}_2$ when there are duplicate elements in $\boldsymbol{X}$, so $\boldsymbol{P}_1 f(\boldsymbol{X}) = \boldsymbol{P}_2 f(\boldsymbol{X})$ means that they must remain duplicates. This inability of separating equal elements demonstrates that the classical definition of permutation-equivariance is overly restrictive when directly applied to multisets as in Definition 1. Therefore, we seek a more appropriate definition of permutation-equivariance for multiset inputs.

Multiset-equivariance. We now consider a slightly different notion of permutation-equivariance that no longer forces equal elements to remain the same: *multiset-equivariance*. Recall the `push_apart` function from Equation 1, which we suggest should be a valid equivariant function for multisets, but cannot be represented by a set-equivariant function. Therefore, we propose that there should be a distinction between set-equivariance and *multiset*-equivariance, the latter corresponding to the appropriate equivariance property for the typical multiset-to-multiset mappings used in set prediction. The key difference of multiset-equivariance is that *the output order for equal elements in the input multiset can be arbitrary*. This leads us to the following definition:

Definition 2 (Multiset-equivariance for multiset inputs)**.** A function $f : \mathbb{R}^{n \times d_1} \to \mathbb{R}^{n \times d_2}$ with multiset input $\boldsymbol{X}$ is multiset-equivariant iff for any permutation matrix $\boldsymbol{P}_1$, there exists a permutation matrix $\boldsymbol{P}_2$ such that $\boldsymbol{P}_1\boldsymbol{X} = \boldsymbol{P}_2\boldsymbol{X}$ and:

$$f(\boldsymbol{P}_1\boldsymbol{X}) = \boldsymbol{P}_2 f(\boldsymbol{X}) \tag{5}$$

Any function that is set-equivariant is also multiset-equivariant (since we can set $\boldsymbol{P}_2 = \boldsymbol{P}_1$), but the converse is not true.

The problem with set-equivariance. We argue that set-equivariance is *too strong* of a property for set prediction. As we have shown in Proposition 1, it prevents equal elements from being processed differently. These equal elements could for example arise from ReLU nonlinearities zeroing out elements, the model naturally tending towards equal elements (perhaps from oversmoothing as is often seen in graph neural networks (Li et al., 2018)), or simply from being part of the dataset used.

Even without exact duplicates in the input, very similar elements can still cause problems for set-equivariant models. For example, in order to learn the `push_apart` function, it is necessary for a model to decide which element is larger. This decision is fundamentally discontinuous, but most models in deep learning cannot represent discontinuous jumps. A continuous model must approximate the discontinuity, which means that close to the discontinuity (where elements are similar) the modeling error will likely be high. This is a general problem when trying to map similar elements in the input (e.g. due to random initialization of $\boldsymbol{Y}_0$) to dissimilar elements in the output.

Summary. It is important to make the distinction between set-equivariance – as has traditionally been considered in the literature – and multiset-equivariance. Deep learning typically operates on multisets rather than sets, so the less restrictive multiset-equivariance *without set-equivariance* should be the desirable property. Most existing set prediction models are set-equivariant, which places a limitation on their modeling capacity even if exact duplicates are never encountered. We refer to models that are multiset-equivariant but not set-equivariant as *exclusively multiset-equivariant*.

2.3 Deep Set Prediction Networks

Here, we highlight the only set prediction model we are aware of that *can* be exclusively multiset-equivariant: deep set prediction networks (DSPN) (Zhang et al., 2019; Huang et al., 2020). The core idea of DSPN is to use backpropagation through a permutation-invariant encoder g (multiset-to-vector) to iteratively update a multiset $\boldsymbol{Y}$. The objective is to find a $\boldsymbol{Y}$ that minimizes the difference between its vector encoding and the input vector $\boldsymbol{z}$ (this vector could for example come from an input image being encoded). This is expressed as the following optimization problem:

$$L(\boldsymbol{Y}, \boldsymbol{z}, \boldsymbol{\theta}) = ||g(\boldsymbol{Y}, \boldsymbol{\theta}) - \boldsymbol{z}||^2 \tag{6}$$

$$\texttt{DSPN}(\boldsymbol{z}) = \arg\min_{\boldsymbol{Y}} L(\boldsymbol{Y}, \boldsymbol{z}, \boldsymbol{\theta}) \tag{7}$$

Because g is permutation-invariant, any ordering for the elements in $\boldsymbol{Y}$ has the same value for L. In the forward pass of the model, the $\arg\min$ is approximated by running a fixed number of gradient descent steps. In the backward pass, the goal is to differentiate Equation 7 with respect to the input vector $\boldsymbol{z}$ and the parameters $\boldsymbol{\theta}$ of the encoder. To do this, Zhang et al. (2019) unroll the gradient descent applied in the forward pass and backpropagate through each gradient descent step.

Equivariance of DSPN. We now discuss the particular form of equivariance that DSPN takes, which has not been done before in the context of multisets. The gradient of the permutation-invariant encoder g with respect to the set input $\boldsymbol{Y}$ is always multiset-equivariant, but depending on the encoder, it is not necessarily set-equivariant. Zhang et al. (2019) find that FSPool-based encoders (Zhang et al., 2020) achieved by far the best results among the encoders they have tried. With FSPool, *DSPN becomes exclusively multiset-equivariant* to its initialization $\boldsymbol{Y}_0$. This is due to the use of numerical sorting in FSPool: the Jacobian of sorting is exclusively multiset-equivariant (Appendix A).

Note that DSPN is not always exclusively multiset-equivariant, but it depends on the choice of encoder. A DeepSets encoder (Zaheer et al., 2017) – which is based on sum pooling – has the same gradients for equal elements, which would make DSPN set-equivariant. It is specifically the use of the exclusively multiset-equivariant gradient of sorting that makes DSPN exclusively multiset-equivariant.

3 Implicit Deep Set Prediction Networks

Despite being exclusively multiset-equivariant, DSPN is outperformed by the set-equivariant Slot Attention (Locatello et al., 2020). We begin by highlighting some issues in DSPN that might be the cause of this, then propose *approximate implicit differentiation* as a solution.

Problems in DSPN. Increasing the number of optimization steps for solving Equation 7 generally results in a better solution (Zhang et al., 2019), but requires significantly more memory and computation time and can lead to training problems (Belanger et al., 2017). To be able to backpropagate through Equation 7, the activations of every intermediate gradient descent step have to be kept in memory. Each additional optimization step in the forward pass also requires backpropagating that step in the backward pass. These issues limit the number of iterations that are computationally

feasible (DSPN uses only 10 steps), which can have a negative effect on the modeling capacity due to insufficient minimization of Equation 7. We aim to address these problems in the following.

Implicit differentiation. Implicit differentiation is a technique that allows efficient differentiation of certain types of optimization problems (Krantz & Parks, 2012; Blondel et al., 2021). We replace the normal backward pass of DSPN with implicit differentiation and call our method *implicit deep set prediction networks* (**iDSPN**). Instead of differentiating through the entire forward optimization, implicit differentiation allows us to only differentiate the (local) optimum obtained at the end of the forward optimization in Equation 7. The minimizer $\boldsymbol{Y}^*(\boldsymbol{z}, \boldsymbol{\theta})$ is implicitly defined by the input vector $\boldsymbol{z}$ and the encoder parameters $\boldsymbol{\theta}$ through the fact that the gradient of L at an optimum is 0:

$$\nabla_{\boldsymbol{Y}} L(\boldsymbol{Y}^*, \boldsymbol{z}, \boldsymbol{\theta}) = \boldsymbol{0} \tag{8}$$

The implicit function theorem allows us to differentiate Equation 8 and obtain the Jacobians $\partial \boldsymbol{Y}^*(\boldsymbol{z}, \boldsymbol{\theta}) / \partial \boldsymbol{z}$ and $\partial \boldsymbol{Y}^*(\boldsymbol{z}, \boldsymbol{\theta}) / \partial \boldsymbol{\theta}$, which is all we need in order for iDSPN to fit into an autodiff framework. Appendix B contains the full details on how this works.

In the implicit differentiation, we need to solve a linear system involving the Hessian $\boldsymbol{H} = \partial(\nabla_{\boldsymbol{Y}} L)/\partial \boldsymbol{Y}$ evaluated at $\boldsymbol{Y}^*$ (also explained in Appendix B). This can be problematic when $\boldsymbol{H}$ is ill-conditioned or singular, so we regularize it with the identity matrix through $\boldsymbol{H}' = \boldsymbol{H}/\gamma + \boldsymbol{I}$ with $\gamma > 0$ (Martens, 2010; Rajeswaran et al., 2019). In the limit of $\gamma \to \infty$, the Hessian is simply replaced by the identity matrix. The limit can be interpreted as pretending that the inexact optimization of L found an exact (local) minimum and that L curves up equally in all directions around that point. Setting the Hessian equal to the identity matrix results in the following Jacobians, which are equivalent to backpropagating through a single gradient descent step at the optimum:

$$\frac{\partial \boldsymbol{Y}^*(\boldsymbol{z}, \boldsymbol{\theta})}{\partial \boldsymbol{z}} = -\frac{\partial \left(\nabla_{\boldsymbol{Y}} L(\boldsymbol{Y}^*, \boldsymbol{z}, \boldsymbol{\theta})\right)}{\partial \boldsymbol{z}} \qquad \frac{\partial \boldsymbol{Y}^*(\boldsymbol{z}, \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = -\frac{\partial \left(\nabla_{\boldsymbol{Y}} L(\boldsymbol{Y}^*, \boldsymbol{z}, \boldsymbol{\theta})\right)}{\partial \boldsymbol{\theta}} \tag{9}$$

Other motivations for this approximation are discussed in Fung et al. (2022), where they call this Jacobian-free backpropagation. Matching their results, we find that this is not only faster to run and easier to implement (no need to solve a linear system involving $\boldsymbol{H}$), but also leads to better results than standard implicit differentiation in preliminary experiments. Hence, we apply this approximation to differentiate Equation 7 with respect to $\boldsymbol{z}$ and $\boldsymbol{\theta}$ in all of our experiments.

Benefits. Implicit differentiation saves a significant amount of memory in training because memory no longer scales with the number of iterations; only the size of the set and its optimizer determine the memory cost since updates to the set can be performed in-place. Because we only take the derivative at the final set, we decrease the computational cost by a factor of around $\frac{T-1}{2T}$, reducing it by nearly a half. The combined improvements mean that we can minimize the objective in Equation 7 better by increasing the number of optimization steps and using more sophisticated optimizers in the inner loop. The optimization procedure is treated as a black box now, so it does not even need to be differentiable.

Further improvements. A simple improvement enabled by implicit differentiation is the use of Nesterov's Accelerated Gradient for estimating $\boldsymbol{Y}^*$. This has a better convergence rate on convex problems than standard gradient descent (Nesterov, 1983) and has found wide applicability in deep learning (Sutskever et al., 2013). Since the computation graph of the optimization is no longer kept in memory, it only has a minor impact on memory and computation: the momentum buffer is the same size as $\boldsymbol{Y}$, updates can be performed in-place, and the additional momentum calculation does not need to be differentiated. We have also tried gradient descent with line search, but found that the slowdown due to additional evaluations of L was not worth the small improvement in performance.

By default, we sample a random initial set $\boldsymbol{Y}_0 \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}/10)$ to start the optimization with. Similar to DSPN, we can also use a learned initial set $\boldsymbol{Y}_0$ to start closer to a solution. However, implicit differentiation treats the optimizer of Equation 7 as a black box, so there is no gradient signal for $\boldsymbol{Y}_0$. We therefore need to include a regularizer in Equation 6 to give us a gradient for $\boldsymbol{Y}_0$, for example by adding the regularizer from Rajeswaran et al. (2019): $\frac{\lambda}{2}||\boldsymbol{Y} - \boldsymbol{Y}_0||^2$. We use this only in Section 4.2 to make the forward passes of iDSPN and DSPN the same so that we can compare them fairly.

Incorporating constraints. We would like to have the ability to impose constraints on the outputs of iDSPN. For example, when each set element corresponds to a class label, we would like to

Table 2: Sample efficiency of equivariance properties on a task that requires separating equal elements. Accuracy in % (mean ± standard deviation) over 6 seeds, higher is better. PE refers to positional encoding, random PE refers to randomly sampled elements like in TSPN (Kosiorek et al., 2020).

Model	Set-equivariant	Multiset-equivariant	Training samples		
			1×	10×	100×
DeepSets	✓	✓	$0.0_{\pm 0}$	$0.0_{\pm 0}$	$0.0_{\pm 0}$
Transformer without PE	✓	✓	$0.0_{\pm 0}$	$0.0_{\pm 0}$	$0.0_{\pm 0}$
Transformer with random PE	✗	✗	$28.5_{\pm 8.1}$	$22.6_{\pm 5.2}$	$22.7_{\pm 7.7}$
Transformer with PE	✗	✗	$0.0_{\pm 0}$	$46.4_{\pm 26.2}$	$94.4_{\pm 1.8}$
BiLSTM	✗	✗	$0.0_{\pm 0}$	$25.1_{\pm 20.2}$	$\mathbf{97.9_{\pm 2.1}}$
DSPN	✗	✓	$89.4_{\pm 2.4}$	$88.9_{\pm 4.9}$	$90.8_{\pm 4.2}$
iDSPN (ours)	✗	✓	$\mathbf{96.8_{\pm 0.6}}$	$\mathbf{98.1_{\pm 0.3}}$	$\mathbf{97.9_{\pm 0.6}}$

enforce that every vector is in the probability simplex. As suggested by Blondel et al. (2021), we can incorporate such constraints through constrained optimization in implicit differentiation by changing Equation 8 to the following and then following the standard derivation in Appendix B:

$$\texttt{proj}(\boldsymbol{Y}^* - \nabla_{\boldsymbol{Y}} L(\boldsymbol{Y}^*, \boldsymbol{z}, \boldsymbol{\theta})) - \boldsymbol{Y}^* = \boldsymbol{0} \tag{10}$$

`proj` is a function that projects its input onto the feasible region. We can then use projected gradient descent with this projection to ensure that the constraints are always satisfied. For example, `proj` could map each element onto the closest point in the probability simplex, which is what we use in Section 4.1. Note that projections can map different elements to the same value, which can cause problems when $\nabla_{\boldsymbol{Y}} L$ is set-equivariant. By using an exclusively multiset-equivariant $\nabla_{\boldsymbol{Y}} L$, we avoid this problem because equal elements after a projection can be separated again.

When we already know parts of the desired $\boldsymbol{Y}$ (e.g. specific elements in the set or the values along certain dimensions), we can help the model by keeping those parts fixed and not optimizing them in Equation 7. For example, in Section 4.1 we know that the first few dimensions in the output multiset should be the exactly same as in the input multiset. If we fix these dimensions during optimization, the model only needs to learn the remaining dimensions that we do not already know. We use this in Section 4.1 (but not the other experiments) to make iDSPN exclusively multiset-equivariant to the input multiset, not just to the initialization $\boldsymbol{Y}_0$.

4 Experiments

In this section, we evaluate three different aspects of our contributions: the usefulness of exclusive multiset-equivariance (Section 4.1), the differences between automatic and our approximate implicit differentiation (Section 4.2), and the applicability of iDSPN to a larger-scale dataset (Section 4.3). We provide detailed descriptions of the experimental procedure in Appendix D, show example inputs and outputs in Appendix E, and open-source the code to reproduce all experiments at `https://github.com/<redacted>/<redacted>` and in the supplementary material.

4.1 Difference between equivariances on class-specific numbering

First, we propose a task that highlights the difference between models that are set-equivariant, exclusively multiset-equivariant, and not multiset-equivariant at all. Given a multiset like $[a, a, b, b, b]$, the aim is to number the elements for each class separately: $[(a, 0), (a, 1), (b, 0), (b, 1), (b, 2)]$. Since these are multisets, the numbered elements do not need to be in any particular order. Equal elements in the input are mapped to different elements in the output, so this is a task where being able to separate equal elements is directly necessary.

Setup. Each element in the input set of size 64 is generated by randomly picking a class out of 4 choices and one-hot encoding it. We have also considered different set sizes and number of classes, but varying these only affected the number of training steps and model parameters required for similar

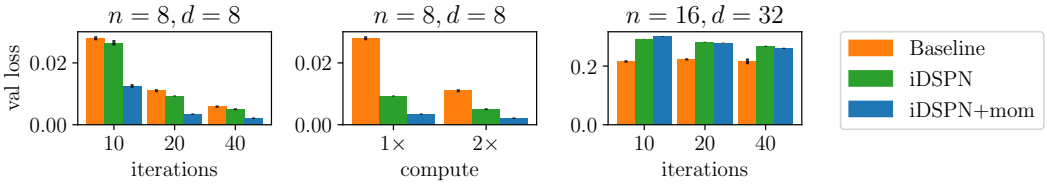


Figure 1: Implicit versus automatic differentiation for different dataset difficulties, reconstruction losses, lower is better. `iDSPN+mom` stands for iDSPN with Nesterov momentum. (left) and (right) compare models at equal iterations, (middle) compares models at roughly the same amount of computation. We select (right) as an atypical example where DSPN outperforms iDSPN.

results. For the target output, a one-hot incremental ID that is separate for each class is concatenated to each element. We evaluate the accuracy of predicting every element in the multiset correctly.

Results. Table 2 shows our results. The two exclusively multiset-equivariant models DSPN and iDSPN perform well at any training set size. As expected, the set-equivariant models are unable to solve this task because they cannot map the equal elements in the input to different elements in the output. This applies even to transformers with random "position" embeddings, which similarly to TSPN (Kosiorek et al., 2020) and Slot Attention (Locatello et al., 2020) use noise to make elements not exactly equal. Meanwhile, BiLSTM and transformers with normal positional encoding require at least 100× more training samples to come close to the performance of iDSPN. This is because they lack the correct structural bias of not relying on the order of the elements, which makes them less sample-efficient. Note that the non-equivariant models are unlikely to benefit significantly from even more training data because they overfit on 1× and 10×, but no longer on 100×.

4.2 Automatic versus implicit differentiation on random multisets

Next, we aim to evaluate our contributions in Section 3. To do this, we compare: 1) baseline DSPN, 2) iDSPN without momentum and 3) iDSPN with Nesterov's Accelerated Gradient. Comparing 1 and 2 allows us to see the difference between backpropagation through the optimizer and the approximate implicit differentiation (Equation 9) when using the same forward pass, while comparing 2 and 3 allows us to see the effects of using a better optimizer for the inner optimization.

We propose the task of autoencoding random multisets: every element is sampled i.i.d. from $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$. The goal is to reconstruct the input with a permutation-invariant latent vector as bottleneck. Varying the set size n and dimensionality of set elements d allows us to control the difficulty of the task. Note that while this appears like a toy task, it is also likely harder than many real-world datasets with similar n and d. This is because the independent nature of the elements means that there is no global structure for the model to rely on; the only way to solve this task is to memorize every element exactly, which is especially difficult when we want the latent space to be permutation-invariant. This task is also challenging because it indirectly requires a version of `push_apart`: similar elements in the initialization $\boldsymbol{Y}_0$ might need to be moved far away from each other in the output.

Setup. All models use the same encoder, so DSPN and iDSPN without momentum have exactly the same forward pass. We train each model on every combination of number of set elements $n \in \{2, 4, 8, 16, 32\}$, set element dimensionality $d \in \{2, 4, 8, 16, 32\}$, and optimization steps $\in \{10, 20, 40\}$. We evaluate the final validation reconstruction losses over 8 random seeds.

Results. In Figure 1, we show a selection of representative results. We show the full results in Appendix C due to the large number of dataset and model combinations. When comparing the models at the same number of iterations (Figure 1, left), iDSPN performs similarly to DSPN, but iDSPN + momentum is a major improvement. This shows that the iDSPN gradients are of similar quality despite only using the approximate implicit differentiation, and that the improved optimizer leads to consistently better results. If we instead compare the models at roughly the same amount of computation (Figure 1, middle), it is clear that iDSPN has significant practical benefits over DSPN regardless of momentum. These results apply to most dataset configurations (see Appendix C).

The only settings where iDSPN + momentum performed worse than DSPN is when $n \times d$ approaches the dimensionality of the latent vector, 512 (Figure 1, right). In these cases, the set *encoder* runs into

Table 3: Performance on a larger-scale set prediction task, average precision (AP) in % (mean ± standard deviation) over 8 random seeds for our results, higher is better. Slot MLP and Slot Attention results are from Locatello et al. (2020), DSPN results are from Zhang et al. (2019). DSPN is trained at 10 iterations and evaluated at 30 iterations. For Slot Attention* and †, Locatello et al. (2020) supervise the model at each iteration and scale up the loss on the coordinates by either 2 or 6. We do not mark the iDSPN 256x256 image results in bold because of the difference in dataset setup.

Model	Image size	AP_{∞}	AP_{1}	$AP_{0.5}$	$AP_{0.25}$	$AP_{0.125}$	$AP_{0.0625}$
Slot MLP	128x128	$19.8_{\pm 1.6}$	$1.4_{\pm 0.3}$	$0.3_{\pm 0.2}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	—
DSPN	128x128	$85.2_{\pm 4.8}$	$81.1_{\pm 5.2}$	$47.4_{\pm 17.6}$	$10.8_{\pm 9.0}$	$0.6_{\pm 0.7}$	—
Slot Attention	128x128	$94.3_{\pm 1.1}$	$86.7_{\pm 1.4}$	$56.0_{\pm 3.6}$	$10.8_{\pm 1.7}$	$0.9_{\pm 0.2}$	—
Slot Attention*	128x128	$96.4_{\pm 0.5}$	$93.6_{\pm 0.8}$	$80.4_{\pm 2.1}$	$26.5_{\pm 2.7}$	$2.6_{\pm 0.3}$	—
Slot Attention†	128x128	$90.6_{\pm 1.8}$	$89.1_{\pm 2.1}$	$84.4_{\pm 2.5}$	$50.3_{\pm 3.8}$	$7.9_{\pm 1.0}$	—
iDSPN	128x128	$\mathbf{98.8}_{\pm 0.5}$	$\mathbf{98.5}_{\pm 0.6}$	$\mathbf{98.2}_{\pm 0.6}$	$\mathbf{95.8}_{\pm 0.7}$	$\mathbf{76.9}_{\pm 2.5}$	$\mathbf{32.3}_{\pm 3.9}$
iDSPN	256x256	$\mathit{99.4}_{\pm 0.3}$	$\mathit{99.2}_{\pm 0.4}$	$\mathit{99.0}_{\pm 0.5}$	$\mathit{97.8}_{\pm 0.7}$	$\mathit{86.9}_{\pm 1.0}$	$47.2_{\pm 3.8}$

fundamental representation limits of set functions (Wagstaff et al., 2019). With performance being limited by the encoder, it is natural that better optimization yields little to no benefit. In Appendix E, it becomes clear that no model performs acceptably on these dataset configurations.

4.3 CLEVR Object Property Prediction

Finally, we evaluate iDSPN on the CLEVR (Johnson et al., 2017) object property prediction task that was used in DSPN (Zhang et al., 2019) and Slot Attention (Locatello et al., 2020). Given an image of a synthetic 3d scene containing up to ten objects, the goal is to predict the set of their properties: 3d coordinate, size, color, and material (18 dimensions in total). To account for different set sizes in the dataset, a 19th dimension is added as a binary indicator for whether an element is present or not.

Setup. We mostly follow the experimental setup of DSPN where a ResNet (He et al., 2016) encodes the image into the input vector $\boldsymbol{z}$, which is decoded into a set by iDSPN. We also include a few modifications, which we elaborate on in Appendix D. Most notably, we use Nesterov's Accelerated Gradient as iDSPN optimizer and increase the number of optimization steps, both changes that would make DSPN even slower and use more memory. Aside from these, most changes we make are simplifications that speed up iDSPN slightly without improving results, so they are also not likely to improve the DSPN baseline results. The evaluation metric is a modified average precision ($AP_{\text{threshold}}$) where a predicted object is considered correct if every property is correct and the 3d coordinates are within the given Euclidean distance threshold of the ground-truth element.

Results. Table 3 shows our results. iDSPN significantly improves over any configuration of Slot Attention on all AP metrics. It is better at attribute classification when ignoring the 3d coordinates (AP_{∞}, 96.4% $\rightarrow$ 98.8%) and improves especially for the metrics with stricter 3d coordinate thresholds ($AP_{0.125}$, 7.9% $\rightarrow$ 76.9%). This is despite Slot Attention† using a three times higher weight on the loss for the coordinates than iDSPN. Note that a stricter AP threshold is always upper bounded by a looser AP threshold, so iDSPN is guaranteed to be better than Slot Attention† on $AP_{0.0625}$.

We observe some overfitting with 128x128 image inputs (1.6e-4 train loss, 5.4e-4 validation loss), which did not appear in preliminary experiments when training an autoencoder with the ground-truth set as input. We reduce this generalization gap by increasing the image size to 256x256 while keeping the latent vector size the same, which results in further performance improvements (1.1e-4 train loss, 2.5e-4 validation loss). We thus believe that the overfitting is due to the ResNet18 image encoder rather than iDSPN.

Efficiency. iDSPN also compares favorably against Slot Attention when considering training times. Locatello et al. (2020) train Slot Attention for 200k steps with batch size 512 while we only train for 100 epochs with batch size 128, which is equivalent to 55k steps. Their model takes around 51 hours to train on a V100 GPU, while our model only takes 2 hours and 40 minutes on a V100 GPU (though with different infrastructure around it). Ignoring possible speed-ups from increasing the batch size for our model, the per-sample iteration speed is similar.

5 Related work

Set prediction is ubiquitous due to the versatility of representing data as sets. Example applications include speaker diarization (Fujita et al., 2019), vector graphics generation (Carlier et al., 2020), object detection (Carion et al., 2020) and point cloud generation (Achlioptas et al., 2018). Since sets are closely related to graphs, there are also natural extensions to molecule generation (Simonovsky & Komodakis, 2018) and scene graph prediction (Krishna et al., 2017; Xu et al., 2017). As we have covered in this paper, most existing set predictors are either not multiset-equivariant (Vinyals et al., 2016; Stewart & Andriluka, 2016; Vaswani et al., 2017; Achlioptas et al., 2018; Li et al., 2018; Rezatofighi et al., 2020) or set-equivariant (Lee et al., 2019; Locatello et al., 2020; Kosiorek et al., 2020). The only set prediction model we are aware of that can be exclusively multiset-equivariant is DSPN (Zhang et al., 2019; Huang et al., 2020). Note that Zhang et al. (2021a) also make use of the exclusively multiset-equivariant Jacobian of sorting, but instead focus on learning multimodal densities over sets, which is tangential to our work.

One topic we have not covered in detail so far is the use of noise to separate similar or identical elements. For example, TSPN (Kosiorek et al., 2020) copies the same latent vector z over all elements and adds Gaussian noise to still be able to separate them with set-equivariant self-attention. While this makes the elements different, the other problems we point out in Section 2.2 still apply, such as the fact that elements can become similar in later parts of the network and that there can be modeling problems with similar (not necessarily equal) elements. Indeed, in Section 4.1 the transformer with random position embeddings (similar application of noise as in TSPN) does not perform well.

The difference between operating on sets versus multisets has also been studied in the context of graph neighborhood aggregation (Xu et al., 2018). There, the concern is about set and multiset-*invariance* rather than equivariance: mean and max pooling lose information about duplicate elements while sum pooling does not. Equal elements easily arise in graphs (e.g. in molecules, the multiset of neighbors often contains multiple atoms of the same chemical element), so making a clear distinction between sets and multisets for invariance and equivariance properties is especially important in such contexts.

6 Conclusion and Future Work

We have established a more appropriate property for set prediction (exclusive multiset-equivariance) and that DSPN satisfies this property. We showed that after improving DSPN with approximate implicit differentiation, iDSPN can outperform a state-of-the-art set predictor like Slot Attention significantly. However, there are still some open problems.

Lack of inductive bias to spatial inputs. Most structured data can be encoded into a vector, which makes iDSPN with its vector input versatile. However, it also limits the inductive bias towards spatial domains. When working on CLEVR with image inputs and set outputs, we had to compress all spatial information into a single vector before decoding it into a set. The benefit of methods like Slot Attention is that they take another set as input, possibly including positional encoding. This allows them to work with feature map inputs rather than only feature vectors, which we believe is especially helpful with tasks like object discovery where having a spatial bias is crucial.

In preliminary experiments on the Tetrominoes object discovery task from Greff et al. (2019), we only inconsistently reached above 98% on the ARI evaluation metric, unable to match the 5-run average of 99.5% ARI for Slot Attention (Locatello et al., 2020) and 99.2% ARI for IODINE (Greff et al., 2019). We believe that in order for iDPSN to scale to larger and more complex tasks, an architecture that makes better use of spatial information is crucial. We also propose that better set prediction is a requirement for better object discovery; if a model cannot predict a set with direct supervision well, why should we expect it to be able to infer a similar one without direct supervision?

Multiset prediction is fundamentally discontinuous. Lastly, we made the point in Section 2 that multiset-equivariant functions like `push_apart` can require discontinuous changes when elements are equal. This is a concrete example of the responsibility problem (Zhang et al., 2020; Huang et al., 2020), which shows that certain set prediction tasks require discontinuous changes. It is an open problem on how to best incorporate such discontinuities into a model while still being trainable. Optimization as is used in iDSPN is a promising step towards dealing with this problem, but there is certainly plenty of room for improvement in both theory and practice.

REPRODUCIBILITY STATEMENT

Proofs and derivations are contained in Appendix A and Appendix B. Detailed descriptions of experiments, including how to generate these datasets in the case for Section 4.1 and Section 4.2, are given in Appendix D. We provide all the code, scripts to reproduce the experiments with the exact hyperparameters, and scripts to calculate the evaluation results. In the de-anonymized version, we intend to make every training run and the corresponding pre-trained weights used to produce the results in this paper available through public Weights & Biases tables (Biewald, 2020).

ETHICS STATEMENT

We propose improvements to a general vector-to-set model and evaluate it largely on synthetic datasets. We have not yet shown the applicability of iDSPN on realistic large-scale datasets. The most immediate potential application is in object detection, which can be used for improving safety (better understanding of the world in self-driving cars) but also for surveillance (face detection and tracking). This is shared with all the general set prediction models. We showed that the main improvement on CLEVR property prediction for iDSPN is the better localization of the objects in the 3d world coordinates. This should be more useful for self-driving cars to locate objects and predict their trajectories more accurately, rather than surveillance where presence is likely more important than the precise position in 3d space.

In general, it is important to be mindful of biases when using such machine learning systems (Torralba & Efros, 2011; Mehrabi et al., 2021). For example, models trained on general object recognition datasets can be discriminatory towards lower-income households (DeVries et al., 2019). It is of course especially important to be careful when the system concerns humans, for example with biases relating to human ethnicity and gender in facial recognition systems (Klare et al., 2012; Robinson et al., 2020).

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Panos Achlioptas, Olga Diamanti, Ioannis Mitliagkas, and Leonidas J Guibas. Learning representations and generative models for 3D point clouds. In *International Conference on Machine Learning (ICML)*, 2018.

David Belanger, Bishan Yang, and Andrew. McCallum. End-to-end learning for structured prediction energy networks. In *International Conference on Machine Learning (ICML)*, 2017.

Lukas Biewald. Experiment tracking with weights and biases, 2020. URL `https://www.wandb.com/`. Software available from wandb.com.

Mathieu Blondel, Quentin Berthet, Marco Cuturi, Roy Frostig, Stephan Hoyer, Felipe Llinares-López, Fabian Pedregosa, and Jean-Philippe Vert. Efficient and modular implicit differentiation. *arXiv preprint arXiv:2105.15183*, 2021.

Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *European Conference on Computer Vision (ECCV)*, pp. 213–229, 2020.

Alexandre Carlier, Martin Danelljan, Alexandre Alahi, and Radu Timofte. Deepsvg: A hierarchical generative network for vector graphics animation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Terrance DeVries, Ishan Misra, Changhan Wang, and Laurens van der Maaten. Does object recognition work for everyone. In *Computer Vision and Pattern Recognition (CVPR)*, 2019.

Yusuke Fujita, Naoyuki Kanda, Shota Horiguchi, Kenji Nagamatsu, and Shinji Watanabe. End-to-End Neural Speaker Diarization with Permutation-free Objectives. In *Interspeech*, 2019.

Samy Wu Fung, Howard Heaton, Qiuwei Li, Daniel McKenzie, Stanley Osher, and Wotao Yin. JFB: Jacobian-free backpropagation for implicit networks. In *AAAI Conference on Artificial Intelligence*, 2022.

Klaus Greff, Raphaël Lopez Kaufmann, Rishab Kabra, Nick Watters, Chris Burgess, Daniel Zoran, Loic Matthey, Matthew Botvinick, and Alexander Lerchner. Multi-object representation learning with iterative variational inference. In *International Conference on Machine Learning (ICML)*, 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Computer Vision and Pattern Recognition (CVPR)*, 2016.

Qian Huang, Horace He, Abhay Singh, Yan Zhang, Ser-Nam Lim, and Austin Benson. Better set representations for relational reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Peter J. Huber. Robust Estimation of a Location Parameter. *The Annals of Mathematical Statistics*, 35(1):73–101, 1964.

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 *Computer Vision and Pattern Recognition (CVPR)*, 2017.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

Brendan F. Klare, Mark J. Burge, Joshua C. Klontz, Richard W. Vorder Bruegge, and Anil K. Jain. Face recognition performance: Role of demographic information. *IEEE Transactions on Information Forensics and Security*, 7(6), 2012.

Adam R Kosiorek, Hyunjik Kim, and Danilo J Rezende. Conditional set generation with transformers. *Workshop on Object-Oriented Learning at ICML*, 2020.

Steven G Krantz and Harold R Parks. *The implicit function theorem: history, theory, and applications*. Springer Science & Business Media, 2012.

Ranjay Krishna, Yuke Zhu, Oliver Groth, Justin Johnson, Kenji Hata, Joshua Kravitz, Stephanie Chen, Yannis Kalantidis, Li-Jia Li, David A Shamma, Michael Bernstein, and Li Fei-Fei. Visual genome: Connecting language and vision using crowdsourced dense image annotations. *International Journal of Computer Vision*, 123(1):32–73, 2017.

Harold W Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning (ICML)*, 2019.

Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper Insights into Graph Convolutional Networks for Semi-Supervised Learning. In *AAAI Conference on Artificial Intelligence*, 2018.

Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. In *International Conference on Machine Learning (ICML)*, 2018.

Francesco Locatello, Dirk Weissenborn, Thomas Unterthiner, Aravindh Mahendran, Georg Heigold, Jakob Uszkoreit, Alexey Dosovitskiy, and Thomas Kipf. Object-centric learning with slot attention. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

James Martens. Deep learning via hessian-free optimization. In *International Conference on Machine Learning (ICML)*, 2010.

Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. A survey on bias and fairness in machine learning. *ACM Comput. Surv.*, 54(6), 2021.

Yurii E. Nesterov. A method for solving the convex programming problem with convergence rate $o(1/k^2)$. *Dokl. Akad. Nauk SSSR*, 269:543–547, 1983.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS) 32*, 2019.

Aravind Rajeswaran, Chelsea Finn, Sham M Kakade, and Sergey Levine. Meta-learning with implicit gradients. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Hamid Rezatofighi, Roman Kaskman, Farbod T Motlagh, Qinfeng Shi, Anton Milan, Daniel Cremers, Laura Leal-Taixé, and Ian Reid. Learn to predict sets using feed-forward neural networks. *arXiv preprint arXiv:2001.11845*, 2020.

Joseph P. Robinson, Gennady Livitz, Yann Henon, Can Qin, Yun Fu, and Samson Timoner. Face recognition: Too bias, or not too bias? In *Computer Vision and Pattern Recognition (CVPR)*, 2020.

Adam Santoro, David Raposo, David G Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Tim Lillicrap. A simple neural network module for relational reasoning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Martin Simonovsky and Nikos Komodakis. GraphVAE: Towards generation of small graphs using variational autoencoders. In *International Conference on Artificial Neural Networks (ICANN)*, 2018.

Russell Stewart and Mykhaylo Andriluka. End-to-end people detection in crowded scenes. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning (ICML)*, 2013.

Antonio Torralba and Alexei A. Efros. Unbiased look at dataset bias. In *Computer Vision and Pattern Recognition (CVPR)*, 2011.

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 (NeurIPS)*, 2017.

Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. In *International Conference on Learning Representations (ICLR)*, 2016.

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

Danfei Xu, Yuke Zhu, Christopher Choy, and Li Fei-Fei. Scene graph generation by iterative message passing. In *Computer Vision and Pattern Recognition (CVPR)*, 2017.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2018.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

David W Zhang, Gertjan J Burghouts, and Cees GM Snoek. Set prediction without imposing structure as conditional density estimation. In *International Conference on Learning Representations (ICLR)*, 2021a.

Miao Zhang, Steven W. Su, Shirui Pan, Xiaojun Chang, Ehsan M Abbasnejad, and Reza Haffari. idarts: Differentiable architecture search with stochastic implicit gradients. In *International Conference on Machine Learning (ICML)*, 2021b.

Yan Zhang, Jonathon Hare, and Adam Prügel-Bennett. Deep Set Prediction Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Yan Zhang, Jonathon Hare, and Adam Prügel-Bennett. FSPool: Learning set representations with featurewise sort pooling. In *International Conference on Learning Representations (ICLR)*, 2020.

A THE JACOBIAN OF SORTING IS EXCLUSIVELY MULTISET-EQUIVARIANT

First, let us clarify what we mean by the Jacobian of sorting. We use $\texttt{sort} : \mathbb{R}^{n\times 1} \to \mathbb{R}^{n\times 1}$ to denote sorting of a multiset with scalar elements in ascending order. To be precise, we define $\texttt{sort}(\boldsymbol{X}) = \boldsymbol{S}_{\boldsymbol{X}}\boldsymbol{X}$ where $\boldsymbol{S}_{\boldsymbol{X}}$ is the permutation matrix that sorting $\boldsymbol{X}$ would apply. With Jacobian of sorting, we mean:

$$\frac{\partial \texttt{sort}(\boldsymbol{X})}{\partial \boldsymbol{X}} = \frac{\partial \boldsymbol{S}_{\boldsymbol{X}}\boldsymbol{X}}{\partial \boldsymbol{X}} = \boldsymbol{S}_{\boldsymbol{X}}^{\top} \tag{11}$$

In the last step, we treat $\boldsymbol{S}_{\boldsymbol{X}}$ as independent of $\boldsymbol{X}$ because $\partial \boldsymbol{S}_{\boldsymbol{X}}/\partial \boldsymbol{X}$ is 0 almost everywhere and undefined on a zero measure set. The definition of sorting can be extended to vector elements (the setting in FSPool (Zhang et al., 2020)) by independently sorting each dimension across the multiset so that we have a different $\boldsymbol{S}_{\boldsymbol{X}}$ for each dimension.

No set-equivariance. To show that the Jacobian of numerical sorting is not set-equivariant, a simple counterexample is to realize that the Jacobian does not change between $\texttt{sort}\left(\left[\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right][0,0]^{\top}\right)$ and $\texttt{sort}\left(\left[\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right][0,0]^{\top}\right)$. The input does not change regardless of how we permute $[0,0]^{\top}$, therefore the Jacobian of sorting does not change.

To make this point more concrete, we implement `push_apart` from the main text using the Jacobian of sorting. We omit the transposes in the following for brevity. We begin by defining the function $g([a,b]) = \texttt{sort}([a,b]) \cdot [-1, 1]$. This multiplies the smaller value with -1 and the larger with 1 (ties broken arbitrarily), then adds them together. Taking the gradient of g with respect to the input set means following the permutation that the sort applied in reverse ($\boldsymbol{S}_{\boldsymbol{X}}^{\top}$), so the -1 gets propagated to the smaller element and 1 to the larger one. In other words, $\nabla_{[a,b]} g = [-1, 1]$ if $a \leq b$ (alternatively $a < b$), else $[1, -1]$. Now we can define $\texttt{push_apart}([a,b]) = [a,b] + \nabla_{[a,b]} g$ to obtain exactly the same function as in the main text.

Multiset-equivariance. We now show that the Jacobian of sorting is multiset-equivariant.

Proposition 2. *The Jacobian of sorting $f(\boldsymbol{X}) = \partial \texttt{sort}(\boldsymbol{X})/\partial \boldsymbol{X} = \boldsymbol{S}_{\boldsymbol{X}}^{\top}$ is multiset-equivariant.*

Proof. To show that for all $\boldsymbol{P}_1$, there exists a $\boldsymbol{P}_2$ that satisfies the two conditions of multiset-equivariance, we set $\boldsymbol{P}_2 = \boldsymbol{S}_{\boldsymbol{P}_1\boldsymbol{X}}^{\top}\boldsymbol{S}_{\boldsymbol{X}}$. We use the fact that $\boldsymbol{S}$ is a permutation matrix so $\boldsymbol{S}^{\top}\boldsymbol{S} = \boldsymbol{S}\boldsymbol{S}^{\top} = \boldsymbol{I}$. To show that the first condition is true:

$$\begin{aligned}
\boldsymbol{P}_2 f(\boldsymbol{X}) &= \boldsymbol{P}_2 \boldsymbol{S}_{\boldsymbol{X}}^{\top} && (12)\\
&= \boldsymbol{S}_{\boldsymbol{P}_1\boldsymbol{X}}^{\top}\boldsymbol{S}_{\boldsymbol{X}}\boldsymbol{S}_{\boldsymbol{X}}^{\top} && (13)\\
&= \boldsymbol{S}_{\boldsymbol{P}_1\boldsymbol{X}}^{\top} && (14)\\
&= f(\boldsymbol{P}_1\boldsymbol{X}) && (15)
\end{aligned}$$

We now use the fact that sorting is permutation-invariant, so for any $\boldsymbol{P}$, $\boldsymbol{S}_{\boldsymbol{X}}\boldsymbol{X} = \boldsymbol{S}_{\boldsymbol{P}\boldsymbol{X}}\boldsymbol{P}\boldsymbol{X}$. To show that the second condition is true:

$$\begin{aligned}
\boldsymbol{P}_2\boldsymbol{X} &= \boldsymbol{S}_{\boldsymbol{P}_1\boldsymbol{X}}^{\top}\boldsymbol{S}_{\boldsymbol{X}}\boldsymbol{X} && (16)\\
&= \boldsymbol{S}_{\boldsymbol{P}_1\boldsymbol{X}}^{\top}\boldsymbol{S}_{\boldsymbol{P}_1\boldsymbol{X}}\boldsymbol{P}_1\boldsymbol{X} && (17)\\
&= \boldsymbol{P}_1\boldsymbol{X} && (18)
\end{aligned}$$

We showed the existence of a $\boldsymbol{P}_2$ that fulfills both conditions for all $\boldsymbol{P}_1$, so f is multiset-equivariant. □

Note that in the definition of multiset-equivariance, it is important that the existential quantifier is on $\boldsymbol{P}_2$ acting on the output of f rather than $\boldsymbol{P}_1$ acting on the input. If the quantifiers were the other way around ($\forall \boldsymbol{P}_2 \exists \boldsymbol{P}_1$), then the multiset-equivariance definition would not be useful: we have seen that permuting equal input elements does not affect the Jacobian of sorting, so there must be some $\boldsymbol{P}_2$ for which no choice of $\boldsymbol{P}_1$ works.

B DERIVATION OF IMPLICIT DSPN

The goal of differentiating Equation 7 is to obtain the gradient of the training objective with respect to $\boldsymbol{z}$ and $\boldsymbol{\theta}$. If we can do that, then iDSPN fits into an autodiff framework like a normal differentiable module. We start by deriving the Jacobians for $\boldsymbol{z}$ and $\boldsymbol{\theta}$ with implicit differentiation and present the approximation for the Hessian afterwards. Blondel et al. (2021) describe a straightforward and efficient recipe for applying implicit differentiation to optimization problems. We follow their approach for the standard implicit differentiation derivation and efficiency concerns. We highly recommend their paper for further background and possible extensions that are made possible by implicit differentiation.

To simplify notation, we combine the target vector with the encoder parameters into $\Theta = (\boldsymbol{z}, \boldsymbol{\theta})$ and restate the optimality condition:

$$\nabla_{\boldsymbol{Y}} L(\boldsymbol{Y}^*, \Theta) = \boldsymbol{0} \tag{19}$$

This implicitly defines a minimizer function $\boldsymbol{Y}^*(\Theta)$. The implicit function theorem (further background in Krantz & Parks (2012)) states that its Jacobian $\partial \boldsymbol{Y}^*(\Theta)/\partial\Theta$ exists if $\nabla_{\boldsymbol{Y}} L$ is continuously differentiable and the Hessian $\partial(\nabla_{\boldsymbol{Y}} L(\boldsymbol{Y}^*, \Theta))/\partial \boldsymbol{Y}^*$ is a square invertible matrix. Using $\nabla_{\boldsymbol{Y}} L$ rather than L lets us apply the implicit function theorem: its Jacobian is a square matrix instead of a vector and the optimality condition is fulfilled even when the root of $\nabla_{\boldsymbol{Y}} L$ only corresponds to a local minimum of L. In practice, we will estimate $\boldsymbol{Y}^*$ with an optimizer (e.g. gradient descent) that approximately minimizes L.

Using the chain rule, we differentiate both sides of Equation 19 with respect to Θ and reorder to obtain:

$$\frac{\partial \nabla_{\boldsymbol{Y}} L(\boldsymbol{Y}^*, \Theta)}{\partial \boldsymbol{Y}^*} \frac{\partial \boldsymbol{Y}^*(\Theta)}{\partial \Theta} = -\frac{\partial \nabla_{\boldsymbol{Y}} L(\boldsymbol{Y}^*, \Theta)}{\partial \Theta} \tag{20}$$

This is a linear system of the form $\boldsymbol{H}\boldsymbol{X} = \boldsymbol{B}$ where we can use autodiff to compute $\boldsymbol{H}$ and $\boldsymbol{B}$. Solving this system for $\boldsymbol{X} = \partial \boldsymbol{Y}^*(\Theta)/\partial\Theta$ lets us compute the gradients for $\boldsymbol{z}$ and $\boldsymbol{\theta}$ with $\nabla_\Theta \mathcal{L} = (\nabla_{\boldsymbol{Y}^*} \mathcal{L})^\top \boldsymbol{X}$, where $\mathcal{L}$ is the loss of the prediction (as opposed to L).

Efficiency. In order to incorporate iDSPN into an autodiff framework, we only need the vector-Jacobian product (VJP) $\boldsymbol{v}^T \boldsymbol{X}$ for some vector $\boldsymbol{v}$. Following Blondel et al. (2021), We can efficiently compute this by solving the much smaller linear system $\boldsymbol{H}\boldsymbol{u} = \nabla_{\boldsymbol{Y}^*} \mathcal{L}$ for $\boldsymbol{u}$. This allows us to compute the desired gradient with $\nabla_\Theta \mathcal{L} = (\nabla_{\boldsymbol{Y}^*} \mathcal{L})^\top \boldsymbol{X} = \boldsymbol{u}^\top \boldsymbol{H}\boldsymbol{X} = \boldsymbol{u}^\top \boldsymbol{B}$.

Instead of solving the linear system exactly, it is more efficient to use the conjugate gradient method (Blondel et al., 2021). Rajeswaran et al. (2019) show that a small number of conjugate gradient steps (5 in their experiment) can be enough to obtain a sufficiently good solution. Another advantage of conjugate gradient is that we only need to compute the Hessian-vector product $\boldsymbol{H}\boldsymbol{u}$ instead of the full Hessian $\boldsymbol{H}$. Not only does this avoid storing $\boldsymbol{H}$ explicitly (decreasing memory requirements), with reverse-mode autodiff like in PyTorch (Paszke et al., 2019) and Tensorflow (Abadi et al., 2015) it also improves parallelization: to compute the full Hessian explicitly we would have to differentiate each entry in $\nabla_{\boldsymbol{Y}} L$ individually. The same advantage applies to $\boldsymbol{B}$ since we can also use a VJP to compute $\boldsymbol{u}^\top \boldsymbol{B}$.

Previous uses of implicit differentiation in meta-learning (Rajeswaran et al., 2019) and neural architecture search (Zhang et al., 2021b) also involve solving a linear system, but their $\boldsymbol{Y}$ corresponds to the neural network parameters and thus usually has millions of entries. In contrast, in our setting we work with a much smaller $\boldsymbol{Y}$, which for example only contains 190 entries in Section 4.3.

Regularization. In Rajeswaran et al. (2019), the regularization of the Hessian like we described in Section 3 appears due to regularizing L by adding $\frac{\lambda}{2}||\boldsymbol{Y} - \boldsymbol{Y}_0||^2$. In preliminary experiments, we observed that even if L is not regularized, solving for the regularized Hessian $(\boldsymbol{H}/\gamma + \boldsymbol{I})$ yielded better performance at the same number of conjugate gradient steps. Adding the identity matrix can be seen as adding 1 to the eigenvalues to improve the conditioning of $\boldsymbol{H}$.

Of course, in the main text the efficiency improvements are irrelevant because we take $\gamma \to \infty$. Due to that, $\boldsymbol{H} = \boldsymbol{I}$ so Equation 20 simplifies to Equation 9. Fung et al. (2022) provide different motivations for this approximation.

C RANDOM SETS FULL RESULTS

We had to exclude 10 runs of DSPN due to significantly worse results making their average uninformative. We did not observe any stability issues with iDSPN, so we did not have to exclude any iDSPN runs. The following excluded DSPN runs were all using 40 iterations:

- 6 runs of $n = 2, d = 32$ (out of 8 seeds),
- 1 run of $n = 2, d = 16$,
- 2 runs of $n = 4, d = 16$,
- 1 run of $n = 4, d = 32$.

Figure 2 and Figure 3 provide different views of the same underlying data in Table 4: each subplot in Figure 2 keeps n fixed and varies d, while each subplot in Figure 3 keeps d fixed and varies n. Since the scale of the losses varies across the different dataset configurations, we normalize each (n, d) combination by dividing the loss by the loss of the `it=10 Baseline` (DSPN) run. That is why `it=10 Baseline` always has a relative loss of 1.

The runs are re-ordered from the main text to be in the order `iDSPN with momentum`, `iDSPN`, `Baseline` to make visual comparison for equal computation easier. The correspondence between color and model is still the same. To make a comparison for equal iterations, compare the baseline DSPN (orange bar) to the two bars to the left of it (in the table, the two rows above). To make a comparison for similar computation, compare the baseline DSPN (orange bar) to the two bars to the right of it (in the table, the two rows below), but only if within the same n, d dataset configuration.

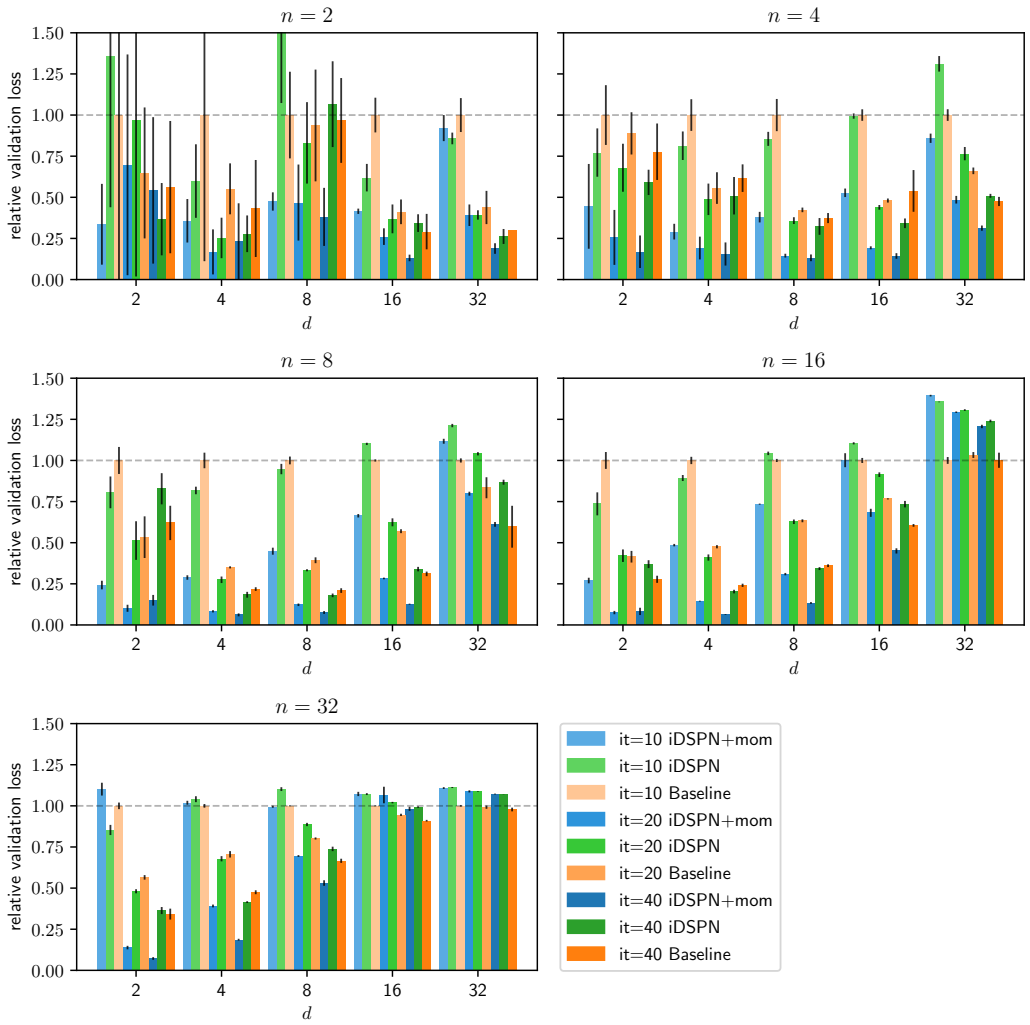


Figure 2: Relative reconstruction loss for autoencoding sets when fixing n and varying d, loss relative to `it=10 DSPN`, lower is better.

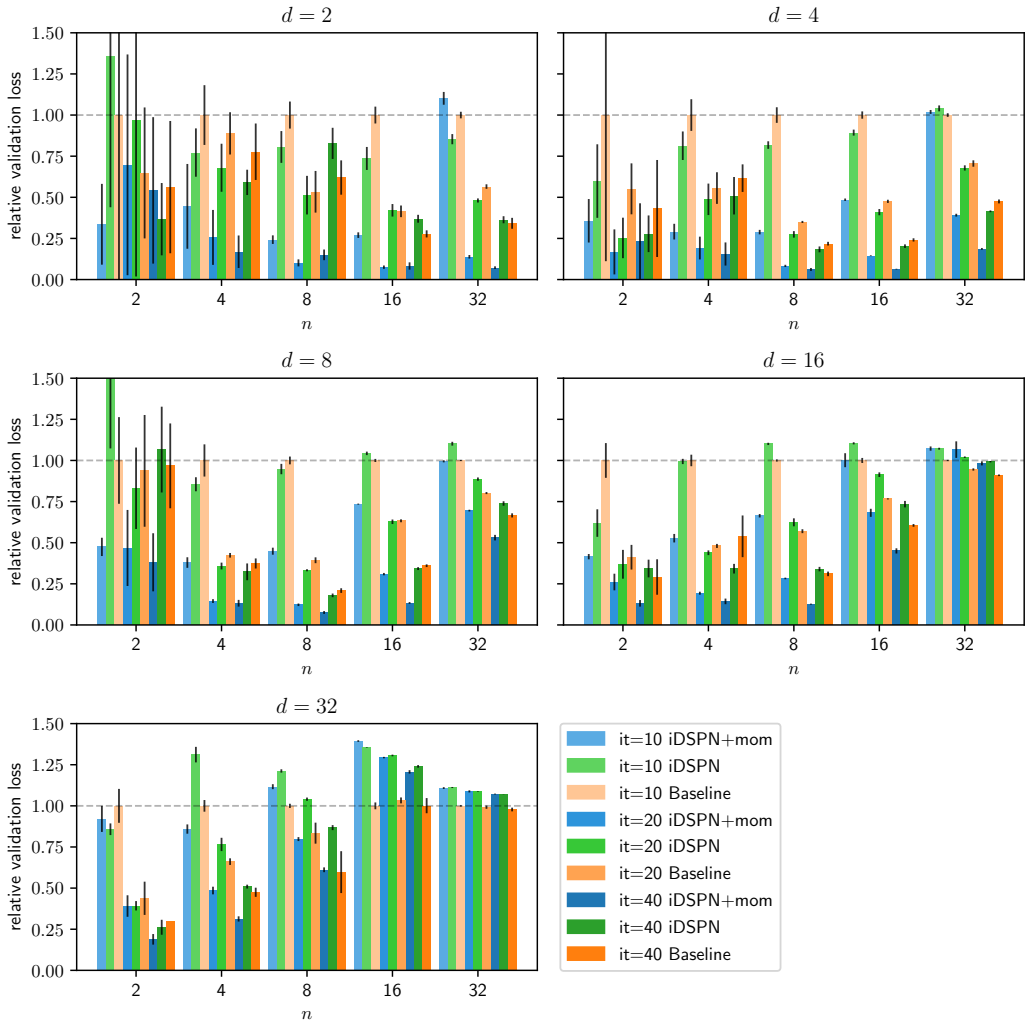


Figure 3: Relative reconstruction loss for autoencoding sets when fixing d and varying n, loss relative to `it=10 DSPN`, lower is better.

Table 4: Full results for autoencoding sets on every dataset configuration with set size n and set element dimensionality d, every model, and varying number of iterations. Reconstruction Huber loss (mean $\pm$ standard deviation) over 8 runs for most results, lower is better. Some dataset configurations with DSPN use fewer runs, see Appendix C for the explanation.

Model	Iterations	$d = 2$	$d = 4$	$d = 8$	$d = 16$	$d = 32$
		$n = 2$				
iDSPN+mom	10	6.0e-5$\pm$4e-5	1.8e-4$\pm$6e-5	2.3e-4$\pm$3e-5	1.0e-3$\pm$4e-5	9.6e-3$\pm$8e-4
iDSPN	10	2.4e-4$\pm$2e-4	2.9e-4$\pm$1e-4	7.8e-4$\pm$2e-4	1.5e-3$\pm$2e-4	9.0e-3$\pm$4e-4
Baseline	10	1.8e-4$\pm$2e-4	4.9e-4$\pm$4e-4	4.9e-4$\pm$1e-4	2.5e-3$\pm$3e-4	1.0e-2$\pm$1e-3
iDSPN+mom	20	1.2e-4$\pm$1e-4	8.3e-5$\pm$7e-5	2.3e-4$\pm$1e-4	6.5e-4$\pm$1e-4	4.1e-3$\pm$7e-4
iDSPN	20	1.7e-4$\pm$2e-4	1.2e-4$\pm$6e-5	4.1e-4$\pm$1e-4	9.1e-4$\pm$2e-4	4.1e-3$\pm$3e-4
Baseline	20	1.2e-4$\pm$7e-5	2.7e-4$\pm$8e-5	4.6e-4$\pm$2e-4	1.0e-3$\pm$2e-4	4.6e-3$\pm$1e-3
iDSPN+mom	40	9.6e-5$\pm$8e-5	1.1e-4$\pm$1e-4	1.9e-4$\pm$9e-5	3.3e-4$\pm$5e-5	2.0e-3$\pm$3e-4
iDSPN	40	6.5e-5$\pm$4e-5	1.4e-4$\pm$5e-5	5.3e-4$\pm$1e-4	8.5e-4$\pm$1e-4	2.7e-3$\pm$5e-4
Baseline	40	1.0e-4$\pm$7e-5	2.1e-4$\pm$1e-4	4.8e-4$\pm$1e-4	7.2e-4$\pm$3e-4	3.2e-3$\pm$nan
		$n = 4$				
iDSPN+mom	10	3.3e-4$\pm$2e-4	5.7e-4$\pm$9e-5	2.2e-3$\pm$2e-4	9.9e-3$\pm$5e-4	5.8e-2$\pm$2e-3
iDSPN	10	5.7e-4$\pm$1e-4	1.6e-3$\pm$2e-4	4.9e-3$\pm$2e-4	1.9e-2$\pm$3e-4	8.8e-2$\pm$3e-3
Baseline	10	7.4e-4$\pm$1e-4	2.0e-3$\pm$2e-4	5.8e-3$\pm$6e-4	1.9e-2$\pm$7e-4	6.7e-2$\pm$2e-3
iDSPN+mom	20	1.9e-4$\pm$1e-4	3.8e-4$\pm$1e-4	8.4e-4$\pm$7e-5	3.6e-3$\pm$2e-4	3.3e-2$\pm$2e-3
iDSPN	20	5.1e-4$\pm$1e-4	9.6e-4$\pm$2e-4	2.1e-3$\pm$1e-4	8.3e-3$\pm$3e-4	5.2e-2$\pm$3e-3
Baseline	20	6.6e-4$\pm$1e-4	1.1e-3$\pm$2e-4	2.4e-3$\pm$8e-5	9.0e-3$\pm$2e-4	4.5e-2$\pm$1e-3
iDSPN+mom	40	1.3e-4$\pm$7e-5	3.1e-4$\pm$1e-4	7.6e-4$\pm$1e-4	2.7e-3$\pm$3e-4	2.1e-2$\pm$1e-3
iDSPN	40	4.4e-4$\pm$6e-5	1.0e-3$\pm$2e-4	1.9e-3$\pm$3e-4	6.4e-3$\pm$6e-4	3.4e-2$\pm$8e-4
Baseline	40	5.8e-4$\pm$1e-4	1.2e-3$\pm$2e-4	2.2e-3$\pm$2e-4	1.0e-2$\pm$2e-3	3.2e-2$\pm$2e-3
		$n = 8$				
iDSPN+mom	10	4.2e-4$\pm$5e-5	2.5e-3$\pm$1e-4	1.3e-2$\pm$6e-4	5.0e-2$\pm$8e-4	2.1e-1$\pm$3e-3
iDSPN	10	1.4e-3$\pm$2e-4	7.0e-3$\pm$2e-4	2.7e-2$\pm$9e-4	8.2e-2$\pm$5e-4	2.2e-1$\pm$2e-3
Baseline	10	1.7e-3$\pm$1e-4	8.5e-3$\pm$4e-4	2.8e-2$\pm$7e-4	7.5e-2$\pm$5e-4	1.9e-1$\pm$2e-3
iDSPN+mom	20	1.8e-4$\pm$4e-5	7.1e-4$\pm$6e-5	3.4e-3$\pm$2e-4	2.1e-2$\pm$4e-4	1.5e-1$\pm$2e-3
iDSPN	20	8.9e-4$\pm$2e-4	2.3e-3$\pm$2e-4	9.3e-3$\pm$2e-4	4.7e-2$\pm$2e-3	1.9e-1$\pm$2e-3
Baseline	20	9.3e-4$\pm$2e-4	3.0e-3$\pm$5e-5	1.1e-2$\pm$5e-4	4.3e-2$\pm$9e-4	1.5e-1$\pm$1e-2
iDSPN+mom	40	2.6e-4$\pm$6e-5	5.2e-4$\pm$8e-5	2.1e-3$\pm$2e-4	9.4e-3$\pm$2e-4	1.1e-1$\pm$3e-3
iDSPN	40	1.4e-3$\pm$2e-4	1.6e-3$\pm$2e-4	5.0e-3$\pm$3e-4	2.5e-2$\pm$1e-3	1.6e-1$\pm$3e-3
Baseline	40	1.1e-3$\pm$2e-4	1.9e-3$\pm$9e-5	5.8e-3$\pm$4e-4	2.3e-2$\pm$1e-3	1.1e-1$\pm$2e-2
		$n = 16$				
iDSPN+mom	10	1.0e-3$\pm$7e-5	1.1e-2$\pm$2e-4	4.4e-2$\pm$2e-4	1.5e-1$\pm$7e-3	3.0e-1$\pm$1e-3
iDSPN	10	2.8e-3$\pm$3e-4	2.0e-2$\pm$4e-4	6.2e-2$\pm$6e-4	1.7e-1$\pm$1e-3	2.9e-1$\pm$4e-4
Baseline	10	3.9e-3$\pm$2e-4	2.2e-2$\pm$5e-4	6.0e-2$\pm$6e-4	1.5e-1$\pm$2e-3	2.2e-1$\pm$4e-3
iDSPN+mom	20	2.9e-4$\pm$4e-5	3.2e-3$\pm$6e-5	1.8e-2$\pm$4e-4	1.0e-1$\pm$4e-3	2.8e-1$\pm$1e-3
iDSPN	20	1.6e-3$\pm$1e-4	9.0e-3$\pm$4e-4	3.7e-2$\pm$8e-4	1.4e-1$\pm$2e-3	2.8e-1$\pm$1e-3
Baseline	20	1.6e-3$\pm$1e-4	1.0e-2$\pm$2e-4	3.8e-2$\pm$5e-4	1.2e-1$\pm$5e-4	2.2e-1$\pm$4e-3
iDSPN+mom	40	3.2e-4$\pm$9e-5	1.4e-3$\pm$5e-5	8.0e-3$\pm$3e-4	6.9e-2$\pm$2e-3	2.6e-1$\pm$2e-3
iDSPN	40	1.4e-3$\pm$9e-5	4.5e-3$\pm$2e-4	2.0e-2$\pm$5e-4	1.1e-1$\pm$3e-3	2.7e-1$\pm$2e-3
Baseline	40	1.1e-3$\pm$8e-5	5.3e-3$\pm$2e-4	2.2e-2$\pm$5e-4	9.3e-2$\pm$1e-3	2.2e-1$\pm$1e-2
		$n = 32$				
iDSPN+mom	10	6.1e-3$\pm$2e-4	3.0e-2$\pm$3e-4	9.3e-2$\pm$6e-4	2.3e-1$\pm$3e-3	3.3e-1$\pm$2e-3
iDSPN	10	4.8e-3$\pm$2e-4	3.0e-2$\pm$5e-4	1.0e-1$\pm$1e-3	2.3e-1$\pm$1e-3	3.3e-1$\pm$1e-3
Baseline	10	5.6e-3$\pm$1e-4	2.9e-2$\pm$3e-4	9.3e-2$\pm$3e-4	2.2e-1$\pm$8e-4	3.0e-1$\pm$1e-3
iDSPN+mom	20	7.7e-4$\pm$6e-5	1.1e-2$\pm$2e-4	6.5e-2$\pm$5e-4	2.3e-1$\pm$1e-2	3.2e-1$\pm$2e-3
iDSPN	20	2.7e-3$\pm$7e-5	2.0e-2$\pm$5e-4	8.3e-2$\pm$9e-4	2.2e-1$\pm$1e-3	3.2e-1$\pm$6e-4
Baseline	20	3.2e-3$\pm$8e-5	2.1e-2$\pm$6e-4	7.5e-2$\pm$6e-4	2.0e-1$\pm$1e-3	3.0e-1$\pm$3e-3
iDSPN+mom	40	4.0e-4$\pm$5e-5	5.4e-3$\pm$2e-4	4.9e-2$\pm$2e-3	2.1e-1$\pm$3e-3	3.2e-1$\pm$8e-4
iDSPN	40	2.0e-3$\pm$1e-4	1.2e-2$\pm$2e-4	6.9e-2$\pm$1e-3	2.1e-1$\pm$9e-4	3.2e-1$\pm$2e-4
Baseline	40	1.9e-3$\pm$2e-4	1.4e-2$\pm$3e-4	6.2e-2$\pm$1e-3	2.0e-1$\pm$1e-3	2.9e-1$\pm$3e-3

D DETAILED EXPERIMENTAL DESCRIPTION

In each of the experiments, we make use of FSPool (Zhang et al., 2020) in the set encoder g for iDSPN and DSPN, which allows them to be exclusively multiset-equivariant.

Multiset loss. When the task is to predict sets, we need to compute a loss between a predicted and a ground-truth set. Throughout this paper, we will make use of the Hungarian loss (Kuhn, 1955):

$$\mathcal{L}(\boldsymbol{Y}^{(1)}, \boldsymbol{Y}^{(2)}) = \min_{\pi \in \Pi} \sum_i ||\boldsymbol{y}_i^{(1)} - \boldsymbol{y}_{\pi(i)}^{(2)}||^2 \tag{21}$$

It finds the optimal 1-to-1 mapping from each element in the prediction $\boldsymbol{Y}^{(1)} = [\boldsymbol{y}_1^{(1)}, \ldots, \boldsymbol{y}_n^{(1)}]^\top$ to each element in the ground-truth $\boldsymbol{Y}^{(2)} = [\boldsymbol{y}_1^{(2)}, \ldots, \boldsymbol{y}_n^{(2)}]^\top$, minimizing the total pairwise losses. A pairwise loss is a standard loss like mean squared error and cross-entropy. The Hungarian loss is permutation-invariant in both its arguments and is able to map duplicate predicted elements to different ground-truth elements and vice versa, which makes it well-suited for multiset prediction.

Set refiner networks. Set refiner networks (SRN) (Huang et al., 2020) make a few minor changes to DSPN. When we mention DSPN, we use the following modifications:

- SRN found that a learning rate of 1 for the inner optimization (Equation 7) is sufficient. DSPN originally used a learning rate of 800 due to normalizing by the set size in the set encoder and a bug that made the effective learning rate depend on the batch size.
- SRN found that with the better learning rate, the additional supervised loss term that DSPN proposes is not necessary for successful training.

D.1 CLASS-SPECIFIC NUMBERING

The following description corresponds to Section 4.1.

Dataset. This experiment can be thought of as an elementwise classification that requires global information: each input element needs to be classified (predicting the numerical ID of the element), but the prediction needs to take the other predictions into account to avoid duplicate IDs.

We use a Hungarian loss to match the predictions and ground-truth within each class, because we do not care which instance of a class gets which ID as long as all the correct IDs are predicted. We use MSE as pairwise loss for DSPN/iDSPN and cross-entropy as pairwise loss for the other models. The baselines perform worse with MSE as pairwise loss.

For each example, the input multiset has a size of 64 with 4-dimensional elements corresponding to 4 classes. This is represented as a 64×4 matrix where each row is a one-hot vector that is sampled i.i.d. from the multinomial distribution over the equally-weighted 4 classes. We generate the target multiset of the same set size with 64-dimensional elements (64×64 matrix), each element being a one-hot vector that represents a number. For each class, we number the corresponding elements sequentially. The setting with $1\times$ samples has a training dataset of size 640. We additionally use a validation set of size 6,400 and a test set of size 64,000 for every run.

Models. We have the prior information that every instance of a class in the input will also appear in the desired output. To make the task easier for the models, we modify them to take this into account. This makes sure that a model does not need to learn to preserve the input class in the output; it only needs to predict the ID associated to the class.

Both the equivariant DeepSets version (as opposed to the more commonly-used invariant version) and BiLSTM naturally pair up each input with each output element, so they already satisfy the desired property. If the "first" element in the input is class c, the "first" predicted ID in the output will naturally correspond to class c as well.

For the Transformer baselines, we use an additional modification: the input set to the transformer decoder is based on an affine transformation $(\boldsymbol{Y}_0)_i = \boldsymbol{W}\boldsymbol{x}_i + \boldsymbol{b}$ of the $1 \leq i \leq n$ input elements. Through this, each input element is paired up with an output element as well. The Transformer

without PE becomes set-equivariant to the input this way. For the Transformers with PE and random PE, we provide the position encoding only to the transformer decoder because giving it to the transformer encoder significantly degraded the performance.

We adapt the iDSPN and DSPN models to this setting by fixing the first four dimensions of $\boldsymbol{Y}$ to the values in the input multiset as we discussed in Section 3, which makes them multiset-equivariant to the input. With the set encoder g architecture of Linear($4 + 64 \rightarrow 256$)–ReLU–Linear($256 \rightarrow 256$)–FSPool, they become exclusively multiset-equivariant. Since the initial set $\boldsymbol{Y}_0$ is sampled i.i.d., we do not need to pay special attention to which input element gets associated with which random initialization. During the inner optimization of iDSPN, we only update the 64 dimensions corresponding to the output set and keep the first four dimensions fixed. Additionally, we apply projected gradient descent with a `proj` that projects each element back onto a point in the probability simplex – the convex hull relaxation of the one-hot encoded IDs – after each gradient descent update. This ensures that the vectors in $\boldsymbol{Y}$ remain valid probability distributions after each update. We found that this significantly improved the performance over the unconstrained version.

Evaluation. We compute the accuracy at the sample-level, meaning a predicted set is considered correct only if every element is correct. The predicted ID for each predicted element is obtained by taking the argmax over the elements' dimensions in the output. The baselines are very volatile during training, which results in very large variances at the end of training. To reduce this variance, we pick the best model according to the validation accuracy, evaluated every 500 training steps. We report the accuracy on the test set for the best model. Each run samples new datasets based on the random seed, so we evaluate all models using the same set of random seeds.

D.2 Random Sets Autoencoding

The following description corresponds to Section 4.2.

Dataset. Every set has a fixed size with n elements of dimensionality d. Every element is sampled i.i.d. from $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$. We assign 64,000 samples to the training set and 6,400 samples to the test set. Changing the random seed also changes the dataset. As loss and evaluation metric, we use Hungarian matching with the Huber loss (Huber, 1964) (quadratic for distances below 1, linear for distances above 1) as pairwise loss. We always use the results for a model after the final epoch without any early-stopping (we did not observe overfitting) or selection based on best loss.

Models. We try to make the DSPN baseline and the iDSPN models as similar as possible. Since DSPN was intended to be used with a learned initial set $\boldsymbol{Y}_0$ (Zhang et al., 2019), we match this in iDSPN and add the regularizer $0.1||\boldsymbol{Y} - \boldsymbol{Y}_0||^2$ to Equation 8 as we discussed in Section 3. We apply the same regularization to the objective in DSPN to make the objectives match up. We initialize $\boldsymbol{Y}_0$ at the start of training with elements sampled from $\mathcal{N}(\mathbf{0}, \boldsymbol{I}/10)$ for all models. For the same seed, the weight initializations are the same for all models.

We use the same set encoder to encode the autoencoder input between the models. We use the same set encoder g between the models (but we do not share the weights with the encoder for the autoencoder input). The architecture of these set encoders is Linear($d \rightarrow 512$)–ReLU–Linear($512 \rightarrow 512$)–FSPool, where the linear layers are applied on each element in a multiset individually with shared weights across the elements.

Note that we do not compare against Slot Attention here due to the difference in setup. Slot Attention takes a set directly as input, so giving it the input set directly would not give us a permutation-invariant bottleneck, which somewhat defeats the point of the autoencoder. It only has to learn to produce the identity function, which we verified it can easily do.

Training. All models are trained for 40 epochs at batch size 128 with Adam (Kingma & Ba, 2015) with a learning rate of 1e-3 and default momentum parameters. This is enough to converge for almost every model and dataset combination. Only the hardest difficulty settings ($32 \times 16, 16 \times 32, 32 \times 32$) would show any benefit from more training.

D.3 CLEVR Object Property Prediction

The following description corresponds to Section 4.3. Where possible, we use exactly the same setup as DSPN.

Every image is resized to 128x128 (or 256x256) and scaled to be in the interval $[0, 1]$. For the ground-truth, we scale down the target 3d coordinates from the interval $[-3, 3]$ to $[-1, 1]$ and concatenate it to the attributes (each of which is one-hot encoded) and the binary mask feature (1 if element is present, 0 if not). The input image is processed by a ResNet encoder, which scales the input down to a 4x4x512 (8x8x512) feature map, on which we apply BatchNorm followed by a 2x2 convolution with stride 2 to obtain a 2x2x128 (4x4x32) feature map. This is flattened to a latent vector with 512 dimensions, which is the input $\boldsymbol{z}$ to iDSPN. We use the Hungarian loss with Huber loss (Huber, 1964) as pairwise loss. We train the model for 100 epochs with the Adam optimizer (Kingma & Ba, 2015), a learning rate of 1e-3, and default momentum hyperparameters.

For evaluation, we adapt the average precision evaluation code to PyTorch. It is calculated by sorting the predicted elements in descending order by the dimension corresponding to the binary mask feature (acting as confidence), then computing the precision-recall curve by including more and more predicted elements, starting from the most confident elements. A prediction is considered correct only if all attributes are correct (after taking the argmax for each attribute individually) and the 3d coordinate is within a given threshold.

We make some changes to the training procedure of DSPN:

- Instead of using a relation network (Santoro et al., 2017) – expanding the set into the set of all pairs first – paired with FSPool, we skip the relation network entirely. This improves the complexity of the set encoder g used in iDSPN from $O(n^2 \log n)$ to $O(n \log n)$. Using the relation network approach would improve our results slightly (e.g. 3.5 percentage points improvement on $\text{AP}_{0.125}$ for 128×128), but is also a bit slower. For simplicity, we therefore opted to not using relation networks.

  The architecture of the set encoder g in iDSPN is Linear($19 \to 512$)–ReLU–Linear($512 \to 512$)–FSPool. The main difference to DSPN is that since there is no concatenation of pairs, so the input dimensionality is 19 instead of 38 and everything is applied on sets of size n rather than sets of size n^2.
- Instead of ResNet34 to encode the input image, we use the smaller ResNet18. This did not appear to affect results.
- Instead of using a learned initial set $\boldsymbol{Y}_0$ as in DSPN, we find that it makes no difference to randomly sample the initial set for every example. We therefore use the latter for simplicity. In initial experiments we found that even initializing every element to $\mathbf{0}$ causes no problems.
- We increase the batch size from 32 to 128. There appeared to be no difference in results between the two, with 128 being faster by making better use of parallelization.
- We use Nesterov's Accelerated Gradient (Nesterov, 1983) with a momentum parameter of 0.9 instead of standard gradient descent without momentum.
- Instead of fixing the number of iterations at 10 like DSPN, we set the number of iterations to 20 at the start of training and change it to 40 after 50 epochs. This had slightly better training loss than starting training with 40 iterations. We have tried a few other ways of increasing the number of iterations throughout training (going from 10 to 20 to 30 to 40 iterations, smooth increase from 1 to 40 over the epochs, randomly sampling an iteration between 20 and 40 every batch), which had little impact on results. iDSPN training was stable in all of these configurations.
- We drop the learning rate after 90 epochs from 1e-3 to 1e-4 for the last 10 epochs. This slightly improved training loss while also reducing variance in epoch-to-epoch validation loss.
- In preliminary experiments, we rarely observed spikes in the training loss. Clipping the gradients in the inner optimization to a maximum L2 norm of 10 seemed to help.

Note that we use a ResNet18 image encoder, while Slot Attention uses a simpler 4-layer convolutional neural network with 5×5 kernels. This difference is not so easily compared: we compress the image into a 512d vector while they operate on a feature map. Their final feature map for CLEVR has 32x32 spatial positions with 64d each, so it can be argued that their latent space is 65536-dimensional. It

is natural that a tighter bottleneck requires more processing. ResNet18 also applies several strided convolutions early, so the amount of processing between it and the Slot Attention image encoder are not too dissimilar. This is reflected by the fact that the time taken to process a sample is similar between ResNet18 + iDSPN and CNN + Slot Attention, even though we use a smaller batch size and could gain a speed-up from better parallelization for higher batch sizes.

A small difference between the default Slot Attention setup and our setup is that they normalize the 3d coordinates to be within [0, 1] while we use an interval of [-1, 1]. In Table 3, Slot Attention* uses an interval of [-1, 1] (the same as iDSPN) while Slot Attention† uses the interval of [-3, 3] (the default coordinate range for this dataset). This increases the weight on the coordinates versus the other attributes, so it improves AP for strict thresholds but trades off classification performance at a threshold of infinity. We did not experiment with different coordinate scales and simply kept it the same as DSPN.

E Example inputs and outputs

In the following, we show some example inputs of each dataset and corresponding model outputs. The focus is not on showing representative examples, but highlighting some qualitative differences.

Table 5 shows the worst failure case of each model trained with 100× samples when numbering elements within a class (Section 4.1). For each model, the sample is selected from the dataset by picking the prediction with the highest number of misclassifications. Since we make every model equivariant to the input, the classes shown in the prediction are guaranteed to be correct; the difficulty lies only in numbering them correctly. Note that the classes in the input are ordered arbitrarily and the order of the numberings within each class is arbitrary as well; we sort the outputs in the table so that it is easier to tell what elements are missing or duplicated. Interestingly, the worst input for transformer with PE, BiLSTM, DSPN, and iDSPN all contain a class that only occurs twice, which is atypically few.

Figure 4 shows the autoencoding of randomly-generated sets (not in training set nor validation set) for each n and d combination (Section 4.2). We leave out iDSPN without momentum because its performance is so similar to DSPN. Both models apply 20 iterations in the forward pass. The visual difference between model qualities is often somewhat subtle and best seen when zooming in on a digital display. It is most obvious on $(n = 4, d = 16)$ and $(n = 8, d = 16)$, but there are smaller differences visible with configurations like $(n = 32, d = 2)$ as well, where iDSPN with momentum tends to be slightly more centered on the ground-truth than DSPN. The examples where $n \times d \geq 512$ show that even if DSPN has a better loss than iDSPN on those configurations, the model quality is far from acceptable.

Table 6 and Table 7 show predictions on CLEVR for different number of evaluation iterations (Section 4.3). We checked the first 128 images of the CLEVR validation set manually and selected examples that looked difficult due to occluded objects, then evaluated the model on these examples. For easier examples, iDSPN is usually correct on all objects at 40 iterations. The mistakes of the model are reasonable, such as confusing a yellow cube that has its sides and bottom occluded for a cylinder (third image). The improvement seen when going from 128x128 to 256x256 images is also quite natural, even for a human observer.

Table 5: Worst failure cases for numbering elements within a class, selected by the highest amount of wrong numberings in one prediction. Predictions are sorted for better readability. Wrong numberings are marked in **red**.

DeepSets	Transformer without PE	Transformer with random PE	Transformer with PE	BiLSTM	DSPN	iDSPN
(a, 2)	(a, 4)	(a, 0)	(a, 0)	(a, 0)	(a, 0)	**(a, 2)**
(a, 2)	**(a, 4)**	(a, 1)	(a, 1)	(a, 1)	(a, 1)	**(a, 3)**
(a, 2)	**(a, 4)**	**(a, 1)**	(a, 2)	(a, 2)	(a, 2)	(b, 0)
(a, 2)	**(a, 4)**	(a, 2)	(a, 3)	(a, 3)	(a, 3)	(b, 1)
(a, 2)	**(a, 4)**	(a, 3)	(a, 4)	**(a, 3)**	(a, 4)	(b, 2)
(a, 2)	**(a, 4)**	(a, 4)	**(a, 4)**	(a, 4)	(a, 5)	(b, 3)
(a, 2)	**(a, 4)**	(a, 6)	(a, 5)	(a, 5)	(a, 6)	(b, 4)
(a, 2)	**(a, 4)**	(a, 7)	(a, 7)	(a, 6)	(a, 7)	(b, 5)
(a, 2)	**(a, 4)**	(a, 8)	(a, 8)	(a, 7)	(a, 8)	(b, 6)
(a, 2)	**(a, 4)**	(a, 9)	(a, 9)	(a, 8)	(a, 9)	(b, 7)
(a, 2)	**(a, 4)**	(a, 10)	(a, 10)	(a, 9)	(a, 10)	(b, 8)
(a, 2)	**(a, 4)**	(a, 11)	(a, 11)	(a, 11)	(a, 11)	(b, 9)
(a, 2)	**(a, 4)**	(a, 12)	(a, 12)	(a, 12)	(a, 12)	(b, 10)
(b, 1)	(b, 0)	(b, 0)	(a, 13)	(a, 13)	(a, 13)	(b, 11)
(b, 1)	**(b, 0)**	**(b, 0)**	(a, 14)	**(b, 3)**	**(b, 4)**	(b, 12)
(b, 1)	**(b, 0)**	(b, 1)	(a, 15)	**(b, 3)**	**(b, 6)**	(b, 13)
(b, 1)	**(b, 0)**	(b, 2)	(a, 16)	(c, 0)	(c, 0)	(c, 0)
(b, 1)	**(b, 0)**	(b, 4)	(a, 17)	(c, 1)	(c, 1)	(c, 1)
(b, 1)	**(b, 0)**	(b, 5)	(a, 18)	(c, 2)	(c, 2)	(c, 2)
(b, 1)	**(b, 0)**	(c, 0)	(a, 19)	(c, 3)	(c, 3)	(c, 3)
(b, 1)	**(b, 0)**	(c, 1)	(a, 21)	(c, 4)	(c, 4)	(c, 4)
(b, 1)	**(b, 0)**	(c, 2)	(a, 22)	**(c, 4)**	(c, 5)	(c, 5)
(b, 1)	**(b, 0)**	(c, 3)	(a, 26)	(c, 5)	(c, 6)	(c, 6)
(b, 1)	**(b, 0)**	(c, 4)	(a, 27)	(c, 7)	**(c, 6)**	**(c, 6)**
(b, 1)	**(b, 0)**	(c, 5)	**(a, 27)**	(c, 8)	(c, 7)	(c, 7)
(b, 1)	**(b, 0)**	(c, 6)	(a, 28)	**(c, 8)**	(c, 8)	(c, 8)
(c, 5)	(c, 4)	(c, 7)	**(a, 28)**	(c, 9)	**(c, 8)**	(c, 9)
(c, 5)	**(c, 4)**	(c, 8)	**(a, 28)**	(c, 11)	(c, 9)	(c, 10)
(c, 5)	**(c, 4)**	(c, 9)	**(a, 28)**	(c, 12)	(c, 10)	(c, 11)
(c, 5)	**(c, 4)**	(c, 10)	(a, 29)	(c, 13)	**(c, 10)**	(c, 12)
(c, 5)	**(c, 4)**	(c, 11)	**(a, 29)**	(d, 0)	(c, 11)	(c, 13)
(c, 5)	**(c, 4)**	(c, 12)	**(a, 29)**	(d, 1)	(c, 12)	(c, 14)
(c, 5)	**(c, 4)**	(c, 13)	**(a, 29)**	(d, 2)	(c, 13)	(c, 15)
(c, 5)	**(c, 4)**	(c, 14)	**(a, 29)**	(d, 4)	(c, 14)	(c, 16)
(c, 5)	**(c, 4)**	(d, 0)	(b, 0)	(d, 5)	(c, 15)	(c, 17)
(c, 5)	**(c, 4)**	(d, 2)	(b, 1)	(d, 6)	(c, 16)	(c, 18)
(c, 5)	**(c, 4)**	(d, 3)	(b, 2)	(d, 7)	(c, 17)	(c, 19)
(c, 5)	**(c, 4)**	(d, 4)	(b, 3)	(d, 10)	**(c, 17)**	(c, 20)
(c, 5)	**(c, 4)**	(d, 5)	(b, 4)	(d, 11)	(c, 18)	(c, 22)
(c, 5)	**(c, 4)**	**(d, 5)**	(b, 5)	(d, 12)	(c, 19)	(c, 23)
(c, 5)	**(c, 4)**	**(d, 5)**	**(b, 5)**	(d, 13)	(c, 20)	(c, 24)
(c, 5)	**(c, 4)**	(d, 6)	(b, 6)	(d, 15)	**(c, 20)**	(c, 25)
(d, 14)	(d, 0)	(d, 7)	(b, 7)	(d, 16)	(c, 21)	(c, 26)
(d, 14)	**(d, 0)**	(d, 8)	(b, 8)	(d, 17)	(c, 22)	**(c, 26)**
(d, 14)	**(d, 0)**	(d, 9)	(b, 9)	(d, 19)	(c, 23)	**(c, 26)**
(d, 14)	**(d, 0)**	(d, 10)	(b, 10)	(d, 20)	(c, 24)	(c, 27)
(d, 14)	**(d, 0)**	(d, 11)	(b, 11)	(d, 22)	(c, 26)	(c, 28)
(d, 14)	**(d, 0)**	**(d, 11)**	(b, 12)	(d, 23)	(c, 27)	(c, 29)
(d, 14)	**(d, 0)**	(d, 14)	(c, 0)	(d, 24)	**(c, 27)**	**(c, 29)**
(d, 14)	**(d, 0)**	(d, 15)	(c, 1)	(d, 25)	(c, 28)	**(c, 29)**
(d, 14)	**(d, 0)**	(d, 17)	(c, 2)	(d, 26)	(d, 0)	(d, 0)
(d, 14)	**(d, 0)**	(d, 18)	(c, 3)	**(d, 26)**	(d, 1)	(d, 1)
(d, 14)	**(d, 0)**	**(d, 18)**	(c, 4)	(d, 27)	(d, 2)	(d, 2)
(d, 14)	**(d, 0)**	(d, 19)	(c, 5)	(d, 28)	(d, 3)	(d, 3)
(d, 14)	**(d, 0)**	(d, 20)	(c, 6)	**(d, 28)**	(d, 4)	(d, 4)
(d, 14)	**(d, 0)**	**(d, 20)**	(c, 7)	**(d, 28)**	(d, 5)	(d, 5)
(d, 14)	**(d, 0)**	(d, 24)	(c, 8)	**(d, 28)**	(d, 6)	(d, 6)
(d, 14)	**(d, 0)**	(d, 25)	(c, 9)	(d, 29)	(d, 7)	(d, 7)
(d, 14)	**(d, 0)**	**(d, 25)**	(c, 10)	**(d, 29)**	(d, 8)	(d, 8)
(d, 14)	**(d, 0)**	**(d, 25)**	(c, 11)	**(d, 29)**	(d, 9)	(d, 9)
(d, 14)	**(d, 0)**	(d, 27)	(c, 12)	**(d, 29)**	(d, 10)	(d, 10)
(d, 14)	**(d, 0)**	**(d, 27)**	**(c, 12)**	**(d, 29)**	(d, 11)	(d, 11)
(d, 14)	**(d, 0)**	**(d, 27)**	**(d, 4)**	**(d, 29)**	(d, 12)	(d, 12)
(d, 14)	**(d, 0)**	**(d, 27)**	**(d, 5)**	**(d, 29)**	(d, 13)	(d, 13)

Figure 4: Random set examples for each combination of n and d. For $d > 2$, only the first two dimensions are shown. All models use 20 iterations.

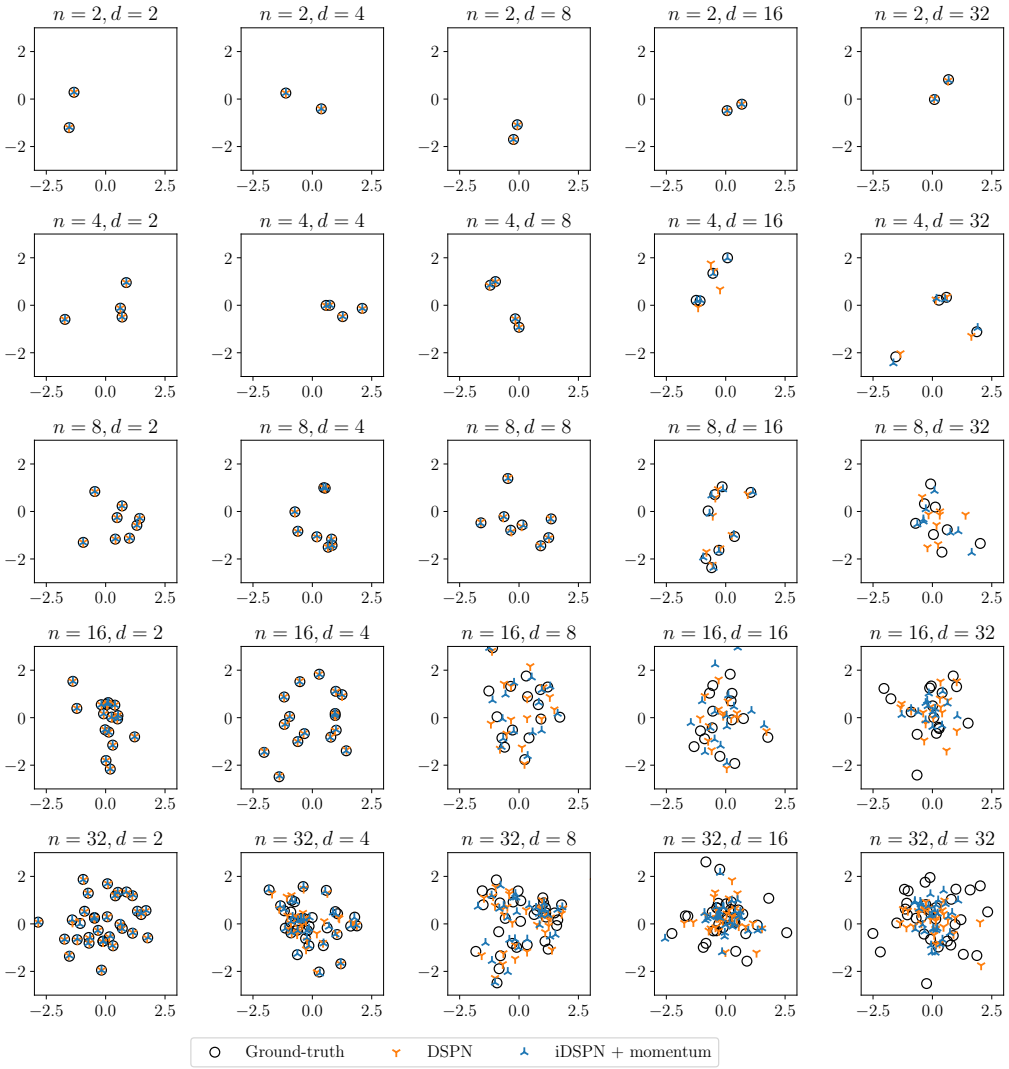


Table 6: CLEVR examples with 128x128 images, cherry-picked by difficulty (highly-occluded objects). Incorrect attributes and Euclidean distances (d) greater than 0.25 marked in **red**. Ground-truth objects are ordered going from top left to bottom right diagonally, predicted objects are matched to that ordering based on Equation 21. We choose an iDSPN run with median performance and evaluate at different iterations.

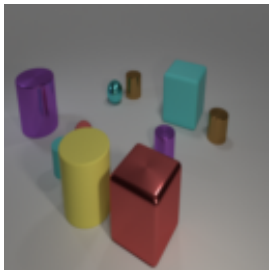
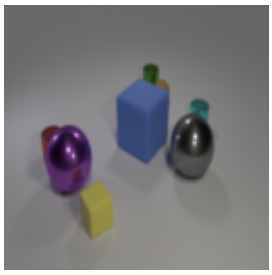
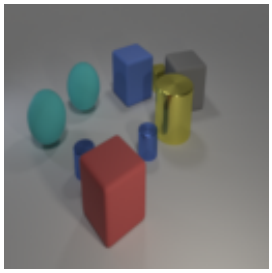
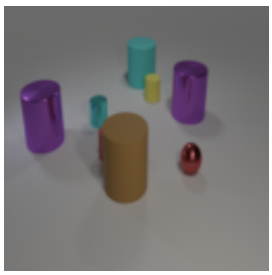
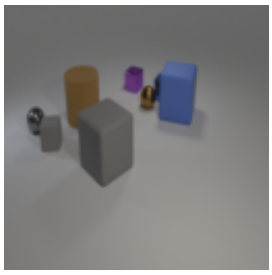
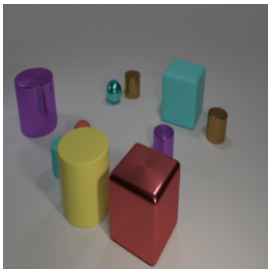
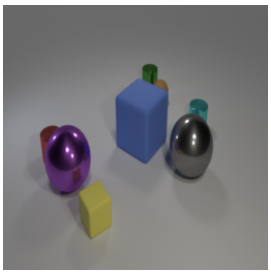
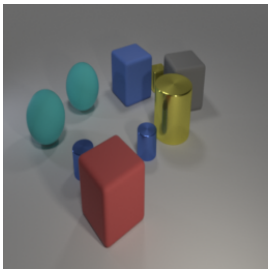
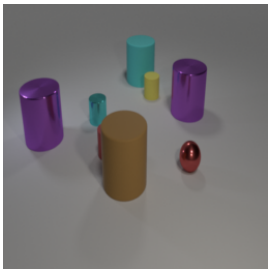
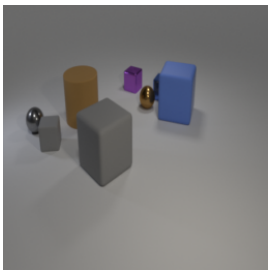
Image	Ground-truth	iDSPN 10 iterations	iDSPN 20 iterations	iDSPN 40 iterations
	(-2.95 -2.56 0.70) large purple metal cylinder (-2.26 0.66 0.35) small cyan metal sphere (-2.13 1.62 0.35) small brown metal cylinder (-0.83 -1.80 0.35) small red rubber sphere (-0.34 -2.86 0.35) small cyan rubber cylinder (0.37 2.25 0.70) large cyan rubber cube (1.25 -2.94 0.70) large yellow rubber cylinder (1.41 0.07 0.35) small purple metal cylinder (2.02 2.29 0.35) small brown metal cylinder (2.93 -2.16 0.70) large red metal cube	(-2.79 -2.19 0.64) **d=0.41** large purple metal cylinder (-2.48 0.60 0.26) d=0.24 small cyan metal sphere (-1.46 2.06 -0.06) **d=0.90** small brown metal cylinder (-0.69 -2.33 0.35) **d=0.55** small **yellow** rubber sphere (0.45 -2.73 0.62) **d=0.84** small **yellow** rubber cylinder (0.73 0.96 0.44) **d=1.36** **small red** rubber cube (0.55 -2.16 -0.49) **d=1.59** large **brown metal** cylinder (0.59 0.13 -0.12) **d=0.95** small purple metal cylinder (2.04 2.23 -0.04) **d=0.40** small brown metal cylinder (3.15 -2.34 0.68) **d=0.29** large red metal cube	(-2.85 -2.58 0.79) d=0.13 large purple metal cylinder (-2.32 0.41 0.41) **d=0.26** small cyan metal sphere (-2.04 1.70 0.41) d=0.14 small brown metal cylinder (-0.81 -1.84 0.32) d=0.05 small red rubber sphere (-0.08 -2.73 0.43) **d=0.30** small **yellow** rubber cylinder (0.44 2.24 0.70) d=0.06 large cyan rubber cube (1.04 -2.36 0.84) **d=0.63** large yellow rubber cylinder (1.04 0.05 0.46) **d=0.39** small purple metal cylinder (1.97 2.21 0.46) d=0.14 small brown metal cylinder (2.96 -2.34 0.83) d=0.23 large red metal cube	(-2.89 -2.53 0.67) d=0.07 large purple metal cylinder (-2.31 0.45 0.33) d=0.21 small cyan metal sphere (-2.03 1.74 0.37) d=0.16 small brown metal cylinder (-0.81 -1.83 0.33) d=0.04 small red rubber sphere (-0.24 -2.86 0.33) d=0.11 small cyan rubber cylinder (0.42 2.28 0.68) d=0.06 large cyan rubber cube (1.30 -2.90 0.72) d=0.07 large yellow rubber cylinder (1.09 0.10 0.31) **d=0.33** small purple metal cylinder (1.96 2.26 0.38) d=0.07 small brown metal cylinder (2.90 -2.20 0.70) d=0.05 large red metal cube
	(-1.73 2.38 0.35) small green metal cylinder (-1.21 -2.81 0.35) small red metal cylinder (-0.77 2.09 0.35) small brown rubber sphere (0.10 -2.88 0.70) large purple metal sphere (0.38 -0.02 0.70) large blue rubber cube (1.25 2.14 0.35) small cyan metal cylinder (1.62 -2.93 0.35) small yellow rubber cube (2.30 0.40 0.70) large gray metal sphere	(-1.52 2.75 0.31) **d=0.42** small green metal cylinder (-1.22 -2.60 0.21) **d=0.25** small **purple** metal cylinder (-0.14 2.12 0.36) **d=0.63** small brown rubber **cylinder** (0.48 -2.71 0.32) **d=0.57** large purple metal sphere (-0.08 -0.79 0.74) **d=0.90** large **gray metal sphere** (0.62 0.55 0.31) **d=1.70** small **blue** metal cylinder (1.46 -2.83 0.31) d=0.18 small yellow rubber cube (2.03 0.63 0.84) **d=0.37** large gray metal sphere	(-1.58 2.64 0.47) **d=0.32** small green metal cylinder (-1.34 -2.82 0.35) d=0.13 small red metal cylinder (-0.15 2.05 0.34) **d=0.62** small brown rubber **cylinder** (0.23 -2.83 0.79) d=0.16 large purple metal sphere (0.12 0.02 0.60) **d=0.28** large blue rubber cube (0.83 1.08 0.40) **d=1.14** small **blue** metal **cube** (1.57 -2.90 0.39) d=0.06 small yellow rubber cube (2.34 0.39 0.78) d=0.09 large gray metal sphere	(-1.73 2.48 0.37) d=0.10 small green metal cylinder (-1.32 -2.82 0.30) d=0.12 small red metal cylinder (-0.49 1.76 0.35) **d=0.44** small brown rubber **cylinder** (0.16 -2.83 0.66) d=0.09 large purple metal sphere (0.29 -0.05 0.71) d=0.09 large blue rubber cube (1.20 2.15 0.32) d=0.06 small cyan metal cylinder (1.58 -2.87 0.33) d=0.07 small yellow rubber cube (2.26 0.39 0.69) d=0.04 large gray metal sphere
	(-2.73 -0.68 0.70) large cyan rubber sphere (-2.09 1.53 0.70) large blue rubber cube (-2.00 -2.71 0.70) large cyan rubber sphere (-1.69 2.84 0.35) small yellow metal cube (-0.18 2.91 0.70) large gray rubber cube (0.12 -2.40 0.35) small blue metal cylinder (0.75 1.26 0.70) large yellow metal cylinder (0.91 -0.23 0.35) small blue metal cylinder (1.97 -2.56 0.70) large red rubber cube	(-2.60 0.45 0.75) **d=1.13** large cyan rubber sphere (-1.03 -0.97 -0.61) **d=3.02** **small cyan metal cylinder** (-2.27 -2.48 0.30) **d=0.54** large cyan rubber sphere (-0.58 2.04 0.25) **d=1.36** **large** yellow metal **cylinder** (-0.45 3.01 0.64) **d=0.29** large gray rubber cube (0.03 -1.84 0.31) **d=0.57** small blue metal cylinder (0.99 -0.14 0.58) **d=1.42** large **blue** metal cylinder (0.43 0.90 -0.23) **d=1.35** small blue metal cylinder (2.00 -2.79 0.76) d=0.24 large red rubber cube	(-2.59 -0.04 0.70) **d=0.66** large cyan rubber sphere (-1.92 1.28 0.51) **d=0.36** large blue rubber cube (-2.23 -2.80 0.83) **d=0.28** large cyan rubber sphere (-1.33 2.81 0.55) **d=0.41** small yellow metal **cylinder** (-0.34 3.02 0.73) d=0.20 large gray rubber cube (-0.01 -2.30 0.38) d=0.17 small blue metal cylinder (0.71 0.91 0.72) **d=0.35** large **blue** metal cylinder (0.77 -0.29 0.43) d=0.17 small blue metal cylinder (1.98 -2.63 0.67) d=0.08 large red rubber cube	(-2.66 -0.38 0.71) **d=0.30** large cyan rubber sphere (-2.15 1.55 0.70) d=0.07 large blue rubber cube (-2.14 -2.80 0.69) d=0.16 large cyan rubber sphere (-1.47 2.75 0.36) d=0.24 small yellow metal **cylinder** (-0.29 2.98 0.67) d=0.13 large gray rubber cube (0.06 -2.17 0.37) d=0.24 small blue metal cylinder (0.87 1.16 0.68) d=0.16 large yellow metal cylinder (0.78 -0.49 0.36) **d=0.29** small blue metal cylinder (1.95 -2.65 0.68) d=0.10 large red rubber cube
	(-2.82 2.97 0.70) large cyan rubber cylinder (-2.28 -2.62 0.70) large purple metal cylinder (-1.71 -0.51 0.35) small cyan metal cylinder (-1.45 2.38 0.35) small yellow rubber cylinder (0.00 -1.35 0.35) small red rubber cylinder (0.43 2.64 0.70) large purple metal cylinder (1.32 -1.70 0.70) large brown rubber cylinder (2.41 0.23 0.35) small red metal sphere	(-2.86 2.82 0.78) d=0.18 large cyan rubber cylinder (-2.34 -2.70 0.70) d=0.10 large purple metal cylinder (-1.02 -0.54 0.63) **d=0.74** small cyan metal cylinder (-1.72 2.36 0.41) **d=0.27** small yellow rubber cylinder (0.22 -0.88 0.59) **d=0.57** small **brown** rubber cylinder (0.42 2.57 0.69) d=0.07 large purple metal cylinder (1.23 -1.69 0.67) d=0.10 large brown rubber cylinder (2.36 0.33 0.10) **d=0.27** small red metal sphere	(-2.84 3.01 0.71) d=0.04 large cyan rubber cylinder (-2.26 -2.65 0.81) d=0.12 large purple metal cylinder (-1.74 -0.63 0.34) d=0.12 small cyan metal cylinder (-1.52 2.36 0.39) d=0.08 small yellow rubber cylinder (0.24 -1.00 0.34) **d=0.42** small **brown** rubber cylinder (0.45 2.62 0.70) d=0.03 large purple metal cylinder (1.22 -1.83 0.72) d=0.16 large brown rubber cylinder (2.44 0.22 0.39) d=0.04 small red metal sphere	(-2.85 2.98 0.70) d=0.03 large cyan rubber cylinder (-2.28 -2.63 0.71) d=0.01 large purple metal cylinder (-1.67 -0.69 0.34) d=0.19 small cyan metal cylinder (-1.58 2.43 0.34) d=0.14 small yellow rubber cylinder (0.22 -1.00 0.35) **d=0.41** small red rubber cylinder (0.46 2.62 0.67) d=0.05 large purple metal cylinder (1.25 -1.80 0.72) d=0.12 large brown rubber cylinder (2.44 0.25 0.36) d=0.04 small red metal sphere
	(-2.87 -2.91 0.35) small gray metal sphere (-2.78 2.14 0.35) small purple metal cube (-2.33 -1.03 0.70) large brown rubber cylinder (-1.73 -2.79 0.35) small gray rubber cube (-1.31 2.74 0.35) small blue metal cube (-1.18 1.67 0.35) small brown metal sphere (-0.01 2.36 0.70) large blue rubber cube (0.45 -1.73 0.70) large gray rubber cube	(-2.81 -2.86 0.40) d=0.09 small gray metal sphere (-2.61 2.22 0.49) d=0.23 small purple metal cube (-2.00 -0.62 0.57) **d=0.54** large brown rubber cylinder (-1.61 -2.20 0.72) **d=0.71** **large** gray rubber cube (-0.48 0.65 0.41) **d=2.26** small **green rubber** cube (-1.43 2.43 0.47) **d=0.81** small brown metal sphere (-0.06 2.36 0.60) d=0.11 large blue rubber cube (0.06 -2.07 0.51) **d=0.55** **small** gray rubber cube	(-2.80 -2.82 0.46) d=0.15 small gray metal sphere (-2.70 2.33 0.29) d=0.21 small purple metal cube (-2.36 -0.88 0.76) d=0.17 large brown rubber cylinder (-1.73 -2.71 0.40) d=0.10 small gray rubber cube (-1.15 1.80 0.25) **d=0.96** small **gray** metal **sphere** (-1.13 1.89 0.38) d=0.23 small brown metal sphere (0.01 2.47 0.66) d=0.12 large blue rubber cube (0.50 -1.76 0.82) d=0.13 large gray rubber cube	(-2.83 -2.88 0.33) d=0.05 small gray metal sphere (-2.72 2.35 0.42) d=0.22 small purple metal cube (-2.42 -0.95 0.73) d=0.13 large brown rubber cylinder (-1.64 -2.59 0.34) d=0.22 small gray rubber cube (-1.20 2.24 0.34) **d=0.52** small **green** metal **sphere** (-1.17 1.60 0.36) d=0.06 small brown metal sphere (0.01 2.36 0.69) d=0.02 large blue rubber cube (0.49 -1.76 0.68) d=0.05 large gray rubber cube

Table 7: CLEVR examples with 256x256 images, cherry-picked by difficulty (highly-occluded objects), same examples as in Table 6. Incorrect attributes and Euclidean distances (d) greater than 0.25 marked in **red**. Ground-truth objects are ordered going from top left to bottom right diagonally, predicted objects are matched to that ordering based on Equation 21. We choose an iDSPN run with median performance and evaluate at different iterations.

Image	Ground-truth	iDSPN 10 iterations	iDSPN 20 iterations	iDSPN 40 iterations
	(-2.95 -2.56 0.70) large purple metal cylinder (-2.26 0.66 0.35) small cyan metal sphere (-2.13 1.62 0.35) small brown metal cylinder (-0.83 -1.80 0.35) small red rubber sphere (-0.34 -2.86 0.35) small cyan rubber cylinder (0.37 2.25 0.70) large cyan rubber cube (1.25 -2.94 0.70) large yellow rubber cylinder (1.41 0.07 0.35) small purple metal cylinder (2.02 2.29 0.35) small brown metal cylinder (2.93 -2.16 0.70) large red metal cube	(-2.70 -1.53 0.53) **d=1.07** **small** purple metal **sphere** (-2.43 0.59 0.38) d=0.18 small **red** metal **cylinder** (-1.11 1.51 0.35) **d=1.02** small brown metal cylinder (0.01 -2.17 0.14) **d=0.94** small **cyan** rubber **cylinder** (0.84 -2.20 0.59) **d=1.37** small **yellow** rubber cylinder (0.26 1.52 0.69) **d=0.74** large **brown** rubber **cylinder** (-1.12 -2.51 0.75) **d=2.41** large **red** rubber cylinder (0.32 0.48 0.46) **d=1.16** small **cyan** metal **cube** (2.34 2.00 0.35) **d=0.43** small brown metal cylinder (2.89 -2.53 0.77) **d=0.38** large red metal cube	(-2.11 -2.49 0.69) **d=0.84** large **red rubber** cylinder (-2.51 -0.48 0.46) **d=1.18** small **purple** metal sphere (-2.28 1.80 0.31) d=0.23 small brown metal cylinder (-0.68 -2.02 0.38) **d=0.26** small **cyan** rubber sphere (0.02 -0.97 0.36) **d=1.93** small cyan rubber **cube** (0.36 2.32 0.70) d=0.07 large cyan rubber cube (0.94 -2.98 0.48) **d=0.38** large yellow rubber cylinder (1.44 -0.32 0.41) **d=0.40** small purple metal cylinder (1.97 2.27 0.36) d=0.06 small brown metal cylinder (2.86 -2.41 0.66) **d=0.26** large red metal cube	(-3.01 -2.50 0.69) d=0.09 large purple metal cylinder (-2.22 0.56 0.35) d=0.11 small cyan metal sphere (-2.20 1.60 0.35) d=0.08 small brown metal cylinder (-0.78 -1.76 0.36) d=0.06 small red rubber sphere (-0.37 -2.61 0.34) d=0.25 small cyan rubber **cube** (0.40 2.30 0.69) d=0.06 large cyan rubber cube (1.24 -3.02 0.73) d=0.09 large yellow rubber cylinder (1.45 0.03 0.36) d=0.06 small purple metal cylinder (2.02 2.24 0.34) d=0.05 small brown metal cylinder (2.85 -2.21 0.68) d=0.10 large red metal cube
	(-1.73 2.38 0.35) small green metal cylinder (-1.21 -2.81 0.35) small red metal cylinder (-0.77 2.09 0.35) small brown rubber sphere (0.10 -2.88 0.70) large purple metal sphere (0.38 -0.02 0.70) large blue rubber cube (1.25 2.14 0.35) small cyan metal cylinder (1.62 -2.93 0.35) small yellow rubber cube (2.30 0.40 0.70) large gray metal sphere	(-1.53 2.32 0.40) d=0.21 small green metal cylinder (-1.18 -2.85 0.37) d=0.05 small red metal cylinder (0.19 0.38 0.54) **d=1.97** small brown **metal** sphere (0.01 -2.44 0.53) **d=0.48** large purple metal sphere (0.28 0.16 0.60) d=0.22 large blue rubber cube (0.81 2.32 0.51) **d=0.50** small cyan metal cylinder (1.76 -3.24 0.37) **d=0.34** small yellow rubber cube (2.23 0.19 0.80) d=0.24 large gray metal sphere	(-1.03 1.59 0.14) **d=1.07** small green metal cylinder (-1.23 -2.78 0.33) d=0.04 small red metal cylinder (-1.10 2.05 0.32) **d=0.33** small brown rubber sphere (0.13 -2.85 0.71) d=0.04 large purple metal sphere (0.36 0.25 0.64) **d=0.28** large blue rubber cube (1.06 2.23 0.39) d=0.22 small cyan metal cylinder (1.68 -2.91 0.36) d=0.07 small yellow rubber cube (2.28 0.41 0.74) d=0.05 large gray metal sphere	(-1.79 2.43 0.37) d=0.08 small green metal cylinder (-1.18 -2.83 0.34) d=0.04 small red metal cylinder (-0.70 2.00 0.35) d=0.12 small brown rubber sphere (0.14 -2.90 0.70) d=0.04 large purple metal sphere (0.39 0.09 0.69) d=0.11 large blue rubber cube (1.23 2.12 0.35) d=0.03 small cyan metal cylinder (1.68 -2.96 0.35) d=0.07 small yellow rubber cube (2.30 0.35 0.70) d=0.05 large gray metal sphere
	(-2.73 -0.68 0.70) large cyan rubber sphere (-2.09 1.53 0.70) large blue rubber cube (-2.00 -2.71 0.70) large cyan rubber sphere (-1.69 2.84 0.35) small yellow metal cube (-0.18 2.91 0.70) large gray rubber cube (0.12 -2.40 0.35) small blue metal cylinder (0.75 1.26 0.70) large yellow metal cylinder (0.91 -0.23 0.35) small blue metal cylinder (1.97 -2.56 0.70) large red rubber cube	(-2.71 -1.83 0.69) **d=1.15** large cyan rubber sphere (-2.55 1.25 0.66) **d=0.54** large blue rubber cube (-0.67 -1.94 0.57) **d=1.55** large cyan **metal** sphere (-1.35 2.36 0.30) **d=0.59** small yellow metal **cylinder** (-0.03 2.21 0.69) **d=0.72** large gray rubber cube (0.62 -1.77 0.12) **d=0.84** small blue metal cylinder (-0.29 2.44 0.57) **d=1.58** large yellow metal **cube** (0.74 0.47 0.45) **d=0.72** small blue metal cylinder (1.85 -2.34 0.69) **d=0.25** large red rubber cube	(-2.75 -0.59 0.70) d=0.08 large cyan rubber sphere (-2.06 1.56 0.69) d=0.04 large blue rubber cube (-2.09 -2.71 0.68) d=0.09 large cyan rubber sphere (-1.56 2.78 0.37) d=0.15 small yellow metal **cylinder** (-0.17 2.87 0.71) d=0.04 large gray rubber cube (0.26 -2.31 0.31) d=0.17 small blue metal cylinder (0.76 1.36 0.73) d=0.11 large yellow metal cylinder (0.88 -0.36 0.38) d=0.14 small blue metal cylinder (1.99 -2.52 0.71) d=0.04 large red rubber cube	(-2.76 -0.69 0.70) d=0.03 large cyan rubber sphere (-2.04 1.54 0.71) d=0.05 large blue rubber cube (-2.05 -2.69 0.69) d=0.06 large cyan rubber sphere (-1.58 2.80 0.38) d=0.11 small yellow metal **cylinder** (-0.18 2.89 0.69) d=0.03 large gray rubber cube (0.14 -2.34 0.36) d=0.07 small blue metal cylinder (0.74 1.31 0.70) d=0.05 large yellow metal cylinder (1.01 -0.37 0.34) d=0.18 small blue metal cylinder (1.98 -2.55 0.70) d=0.01 large red rubber cube
	(-2.82 2.97 0.70) large cyan rubber cylinder (-2.28 -2.62 0.70) large purple metal cylinder (-1.71 -0.51 0.35) small cyan metal cylinder (-1.45 2.38 0.35) small yellow rubber cylinder (0.00 -1.35 0.35) small red rubber cylinder (0.43 2.64 0.70) large purple metal cylinder (1.32 -1.70 0.70) large brown rubber cylinder (2.41 0.23 0.35) small red metal sphere	(-2.92 3.07 0.55) d=0.21 large cyan rubber cylinder (-1.86 -1.51 0.53) **d=1.20** large purple metal cylinder (-0.65 -0.98 0.17) **d=1.17** small **brown** metal cylinder (-1.31 2.00 0.47) **d=0.41** small yellow rubber cylinder (1.11 -1.28 0.49) **d=1.11** small red rubber cylinder (0.30 2.58 0.60) d=0.17 large purple metal cylinder (-1.04 -2.39 0.59) **d=2.46** large **purple metal** cylinder (2.04 0.36 0.48) **d=0.42** small red metal sphere	(-2.78 3.02 0.76) d=0.08 large cyan rubber cylinder (-1.69 -2.70 0.69) **d=0.60** large purple metal cylinder (-1.77 -0.41 0.52) d=0.20 small cyan metal cylinder (-1.26 2.29 0.34) d=0.21 small yellow rubber cylinder (-0.78 -0.85 0.31) **d=0.93** small **cyan** rubber cylinder (0.50 2.71 0.68) d=0.10 large purple metal cylinder (1.04 -1.48 0.51) **d=0.41** **small red** rubber cylinder (2.37 0.17 0.36) d=0.08 small red metal sphere	(-2.82 2.92 0.69) d=0.05 large cyan rubber cylinder (-2.24 -2.56 0.70) d=0.08 large purple metal cylinder (-1.78 -0.47 0.37) d=0.09 small cyan metal cylinder (-1.39 2.42 0.35) d=0.07 small yellow rubber cylinder (0.07 -1.32 0.32) d=0.07 small red rubber cylinder (0.42 2.73 0.69) d=0.09 large purple metal cylinder (1.30 -1.76 0.68) d=0.06 large brown rubber cylinder (2.41 0.22 0.35) d=0.01 small red metal sphere
	(-2.87 -2.91 0.35) small gray metal sphere (-2.78 2.14 0.35) small purple metal cube (-2.33 -1.03 0.70) large brown rubber cylinder (-1.73 -2.79 0.35) small gray rubber cube (-1.31 2.74 0.35) small blue metal cube (-1.18 1.67 0.35) small brown metal sphere (-0.01 2.36 0.70) large blue rubber cube (0.45 -1.73 0.70) large gray rubber cube	(-2.44 -2.11 0.39) **d=0.91** small gray metal sphere (-2.46 2.17 0.19) **d=0.36** small purple metal cube (-1.72 -1.06 0.56) **d=0.63** large brown rubber cylinder (-2.30 -2.98 0.52) **d=0.62** small gray **metal** cube (-0.96 2.02 0.21) **d=0.81** small blue metal cube (-1.67 1.90 0.48) **d=0.56** small brown metal sphere (-0.46 2.39 0.67) **d=0.46** large blue rubber cube (0.15 -1.75 0.68) **d=0.30** large gray rubber cube	(-2.86 -2.92 0.36) d=0.01 small gray metal sphere (-2.76 2.06 0.35) d=0.09 small purple metal cube (-2.26 -0.98 0.64) d=0.11 large brown rubber cylinder (-1.78 -2.73 0.39) d=0.08 small gray rubber cube (-1.43 2.26 0.50) **d=0.52** small blue metal cube (-1.12 1.72 0.35) d=0.08 small brown metal sphere (-0.17 2.39 0.71) d=0.16 large blue rubber cube (0.40 -1.72 0.72) d=0.06 large gray rubber cube	(-2.87 -2.93 0.35) d=0.02 small gray metal sphere (-2.80 2.15 0.37) d=0.03 small purple metal cube (-2.26 -0.98 0.68) d=0.10 large brown rubber cylinder (-1.81 -2.78 0.34) d=0.08 small gray rubber cube (-1.38 2.16 0.34) **d=0.59** small blue metal cube (-1.14 1.72 0.34) d=0.06 small brown metal sphere (-0.22 2.45 0.71) d=0.23 large blue rubber cube (0.44 -1.68 0.69) d=0.06 large gray rubber cube