MULTISET-EQUIVARIANT SET PREDICTION WITH 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 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 set of vectors in memory is to store it as a matrix, for which we need to choose some arbitrary order for the elements. To avoid being affected by this arbitrary order, recent set prediction models make use of a property called permutation-equivariance (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 this function that takes a multiset 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, 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. This is due to the limitation of being permutation-equivariant, which we call *set-equivariance*. 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.

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 (Subsection 2.3). This makes it the only deterministic model we are aware of that can satisfy exclusive multiset-equivariance.

2. **Implicit DSPN.** Motivated by our analysis on exclusive multiset-equivariance, we seek to further improve DSPN. We propose *implicit DSPN* (iDSPN) in Section 3: a version of DSPN that uses implicit differentiation, which enables better optimizers and more iterations to be used at a constant memory cost and less computation time. We then simplify this approach to avoid computing a Hessian entirely while still keeping the same benefits.

In our experiments in Section 4, we start with evaluating the first point by comparing exclusively multiset-equivariant models with both set-equivariant (Zaheer et al., 2017; Vaswani et al., 2017; Kosiorek et al., 2020) and non-equivariant models (Vaswani et al., 2017; Hochreiter & Schmidhuber, 1997) on synthetic data. The former completely fail on this simple toy task while the latter are much 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 model (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.

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 are 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

Set prediction models make use of set-to-set functions that are permutation-equivariant (Lee et al., 2019; Locatello et al., 2020; Carion et al., 2020; Kosiorek et al., 2020). This is desirable when

processing sets because it prevents a function from relying on the arbitrary order of the set in its matrix representation. Permutation-equivariant 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 Deep Sets (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 lemma:

Lemma 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 argued should be a valid permutation-equivariant function for multisets, but cannot be represented by a set-equivariant function. Therefore, we suggest 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 Lemma 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 over-smoothing 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 make a decision of which element is larger. This decision is fundamentally discontinuous. Since most models used in deep learning cannot represent discontinuous jumps, near the discontinuity (when the elements are similar) the modeling error will likely be high.

Summary We argue that 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 deterministic 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 $\boldsymbol{z}$. The input $\boldsymbol{z}$ could for example come from an image that is encoded into a vector. 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, Zhang et al. (2019) backpropagate through the gradient descent iterations in order to compute the gradients of the training objective with respect to the input vector $\boldsymbol{z}$ and the parameters $\boldsymbol{\theta}$ of the encoder.

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 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) perform by far the best among the ones they have tried. With this type of encoder, *DSPN becomes exclusively multiset-equivariant*. This is due to the use of numerical sorting in FSPool: the Jacobian of sorting is exclusively multiset-equivariant. We prove this in Appendix A.

Note that exclusive multiset-equivariance is not always obtained in DSPN, but depends on the choice of encoder. For instance, 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

The property of exclusive multiset-equivariance motivates us to study DSPN in greater detail. There are of course other factors that determine the quality of a set predictor, so we begin by pointing out some remaining problems of DSPN.

Problems in DSPN Increasing the number of optimization steps in DSPN generally results in a better solution, but requires significantly more memory and computation time. To be able to backpropagate through the inner optimization, the activations of every intermediate step have to be kept in memory. Furthermore, each additional optimization step requires backpropagating that step in the backward pass as well. These issues limit the number of iterations (10 in DSPN) that are computationally feasible, which can have a negative effect on the modeling capacity. 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. 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:

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

By applying the implicit function theorem on Equation 8, we 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 of 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 $\widetilde{\boldsymbol{H}} = \boldsymbol{H}/\gamma + \boldsymbol{I}$ 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 the case of backpropagating through a single gradient descent operation:

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

Similar to Fung et al. (2021), 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. Hence, we apply this simplification in all of our experiments.

Benefits First, this means that we no longer have to keep the entire computation graph of the optimization in memory during training. This saves a significant amount of memory: previously, all intermediate activations of every step in the optimization, both forward and backward passes through L, had to be kept in memory. In our proposed training procedure, 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 of the final set, we decrease the computational cost by a factor of $\frac{T-1}{2T}$, 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 individual steps of the optimizer do not even need to be differentiable anymore since the optimization procedure is treated as a black-box that does not need to be differentiated.

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 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. Unlike DSPN 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): $\lambda ||\boldsymbol{Y} - \boldsymbol{Y}_0||^2$. We will use this when comparing iDSPN to DSPN performance in Subsection 4.2.

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 enforce that every vector is in the probability simplex. As suggested by Blondel et al. (2021), we can

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 PE	✗	✗	$0.0_{\pm 0}$	$46.4_{\pm 26.2}$	$94.4_{\pm 1.8}$
Transformer with random PE	✗	✗	$28.5_{\pm 8.1}$	$22.6_{\pm 5.2}$	$22.7_{\pm 7.7}$
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	✗	✓	$\mathbf{96.8_{\pm 0.6}}$	$\mathbf{98.1_{\pm 0.3}}$	$\mathbf{97.9_{\pm 0.6}}$

incorporate such constraints through constrained optimization in implicit differentiation by changing Equation 8 to the following and using the standard derivation in Appendix B:

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

$\texttt{proj}$ 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, $\texttt{proj}$ could map each element onto the closest point in the probability simplex, like we do in Subsection 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.

In multiset-equivariant tasks, we want the output multiset to be ordered the same as the input multiset: the "first" element in the input corresponds to the "first" in the output, but the overall order is still irrelevant. We can make iDSPN multiset-equivariant to the input – not just to its initialization $\boldsymbol{Y}_0$ – by concatenating the input set to the set being optimized. In general, when we already know parts of the desired $\boldsymbol{Y}$ (e.g. certain elements or the values in certain dimensions), we can help the model by keeping those parts fixed and not optimizing them in the forward optimization. We use this in Subsection 4.1 to make iDSPN equivariant to the input.

4 Experiments

In this section, we evaluate three different aspects of our contributions: the usefulness of exclusive multiset-equivariance (Subsection 4.1), the differences between our implicit and automatic differentiation (Subsection 4.2), and the applicability of iDSPN to a larger-scale dataset (Subsection 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>`.

4.1 Difference between equivariances

First, we propose a task that clearly 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 class numbers, but varying these only affected the required number of training steps and model parameters. 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.

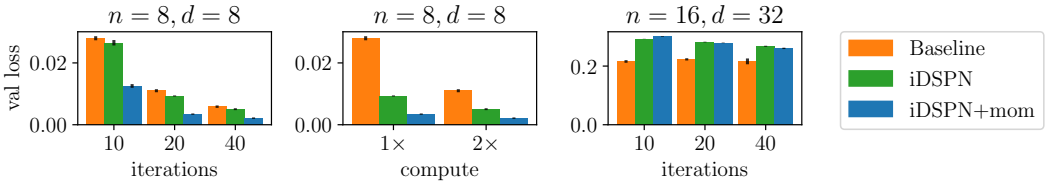


Figure 1: Implicit versus automatic differentiation for different dataset difficulties, reconstruction losses, lower is better. (left) and (right) compare models at equal iterations, (middle) compares models at roughly the same amount of computation. Note that (right) has a loss that is 10 times higher than the others.

Results Table 2 shows our results. iDSPN is close to solving the problem at any training set size by being exclusively multiset-equivariant like DSPN. 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, transformers with position encoding and BiLSTM 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 Implicit versus automatic differentiation

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 implicit differentiation 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 auto-encoding 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 i.i.d. 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.

Setup All models use exactly the same encoder, so DSPN and iDSPN without momentum have 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 taking the derivative at the final optimizer step, 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. These results apply to most dataset configurations (see Appendix C).

The only settings where iDSPN + momentum is 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 fundamental representation limits of set functions (Wagstaff et al., 2019). With performance being limited by the encoder, it is natural that better optimization brings little to no benefit.

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 T=30	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	$\mathit{99.4}_{\pm 0.3}$	$\mathit{99.2}_{\pm 0.4}$	$\mathit{99.0}_{\pm 0.5}$	$\mathit{97.8}_{\pm 0.7}$	$\mathit{86.9}_{\pm 1.0}$	$\mathit{47.2}_{\pm 3.8}$

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 several objects, the goal is to predict the set of their properties: 3d coordinate, size, color, and material (18 dimensions in total). An additional dimension is added as a binary indicator for whether an element is present or not to account for the different set sizes in the dataset. The main comparison we intend to make here is against Slot Attention, the state-of-the-art model on this dataset.

Setup We 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 the set by iDSPN. We include a few modifications to improve our results and also simplify the model slightly, which we elaborate on in Appendix D. Most notably, we use Nesterov's Accelerated Gradient as iDSPN optimizer. 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. A threshold of infinity ignores the 3d coordinates.

Results Table 3 shows our results. iDSPN significantly improves over any configuration of Slot Attention on all AP metrics. It is better at the attribute classification ignoring the 3d coordinates (96.4 $\rightarrow$ 98.8) and improves especially for the stricter AP thresholds like $AP_{0.125}$ (7.9 $\rightarrow$ 76.9). Note that a stricter AP threshold is always upper bounded by the looser AP threshold, so iDSPN is guaranteed to be better than Slot Attention† on $AP_{0.0625}$. We observed some overfitting when using 128x128 image inputs, which did not show up in preliminary experiments when training an autoencoder with the ground-truth set as input. We reduced this overfitting significantly by increasing the image size to 256x256 while keeping the latent vector size the same, which results in further performance improvements. 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). As sets are also 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., 2015; Stewart & Andriluka, 2016; Rezatofighi et al., 2018; Vaswani et al., 2017; Achlioptas et al., 2018; Li et al., 2018) 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). DESP (Zhang et al., 2021) is exclusively multiset-equivariant but not a standard set predictor. It also uses the Jacobian of sorting, but has the slightly different goal of diverse sampling by learning set energies without a traditional set loss, so we do not compare against them in the experiments.

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 Subsection 2.2 still apply, such as the fact that elements can become similar again in later parts of the network and that there can be modeling problems with similar (not necessarily equal) elements. Indeed, in Subsection 4.1 the transformer with random position embeddings (similar application of noise to 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 can easily arise in graphs (e.g. in molecules, the multiset of neighbors often contains multiple atoms of the same chemical element), so the difference in invariance and equivariance between sets and multisets is especially important in such contexts.

6 Conclusion and Future Work

In this paper, 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 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 such as 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 theory and practice.

REPRODUCIBILITY STATEMENT

All the code with scripts to reproduce the exact hyperparameters used and scripts to calculate the evaluation results. Pretrained models and training runs (public wandb tables)? Detailed experimental descriptions and how to generate the datasets. Proofs in appendix

ETHICS STATEMENT

Set prediction is versatile so it encompasses many tasks like object detection and face tracking. Can be used for surveillance.

REFERENCES

Panos Achlioptas, Olga Diamanti, Ioannis Mitliagkas, and Leonidas J Guibas. Learning representations and generative models for 3D point clouds. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.

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*, volume 33, pp. 16351–16361, 2020.

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. Fixed point networks: Implicit depth models with jacobian-free backprop. *arXiv preprint arXiv:2103.12803*, 2021.

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 *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

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

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*, 2020.

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.

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.

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 *Proceedings of the 36th International Conference on Machine Learning*, pp. 3744–3753, 2019.

Q. Li, Z. Han, and X.-M. Wu. Deeper Insights into Graph Convolutional Networks for Semi-Supervised Learning. In *The Thirty-Second AAAI Conference on Artificial Intelligence*. AAAI, 2018.

Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. In *Proceedings of the 35th 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*, 2020.

James Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, volume 27, pp. 735–742, 2010.

Y. 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.

Aravind Rajeswaran, Chelsea Finn, Sham M Kakade, and Sergey Levine. Meta-learning with implicit gradients. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

S. Hamid Rezatofighi, Roman Kaskman, Farbod T. Motlagh, Qinfeng Shi, Daniel Cremers, Laura Leal-Taixé, and Ian Reid. Deep perm-set net: Learn to predict sets with unknown permutation and cardinality using deep neural networks. 2018.

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 *Proceedings of the 30th International Conference on Machine Learning*, 2013.

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*, 2017.

Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*, 2015.

Edward Wagstaff, Fabian B. Fuchs, Martin Engelcke, Ingmar Posner, and Michael A. Osborne. On the limitations of representing functions on sets. In *Proceedings of the 36th 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*, 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*, 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*, 2021.

Yan Zhang, Jonathon Hare, and Adam Prügel-Bennett. Deep Set Prediction Networks. In *Advances in Neural Information Processing Systems 32*, 2019.

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

A THE JACOBIAN OF SORTING IS EXCLUSIVELY MULTISET-EQUIVARIANT

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 when sorting $\left[\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right][0,0]$ versus $\left[\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right][0,0]$. The input does not change regardless of how we permute $[0,0]$, 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 use $\texttt{sort} : \mathbb{R}^n \to \mathbb{R}^n$ to denote sorting of a multiset with scalar elements in ascending order. This is extended to FSPool with higher-dimensional elements by independently sorting each dimension across the multiset.

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 backwards (similar to how max pooling is differentiated), 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. To do this, we define $\texttt{sort}(\boldsymbol{X}) = \boldsymbol{S_X X}$ where $\boldsymbol{S_X}$ is the permutation matrix that sorting $\boldsymbol{X}$ would apply.

Lemma 2. *The Jacobian of sorting $f(\boldsymbol{X}) = \partial \texttt{sort}(\boldsymbol{X}) / \partial \boldsymbol{X} = \boldsymbol{S_X}^\top$ 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 $\boldsymbol{P}_2 = \boldsymbol{S}_{\boldsymbol{P}_1\boldsymbol{X}}^\top \boldsymbol{S_X}$. We use the fact that $\boldsymbol{S}$ is a permutation matrix so $\boldsymbol{S}^T\boldsymbol{S} = \boldsymbol{S}\boldsymbol{S}^T = \boldsymbol{I}$. The first condition is:

$$f(\boldsymbol{P}_1\boldsymbol{X}) = \boldsymbol{P}_2 f(\boldsymbol{X}) \tag{11}$$
$$\boldsymbol{S}_{\boldsymbol{P}_1\boldsymbol{X}}^\top = \boldsymbol{P}_2 \boldsymbol{S}_{\boldsymbol{X}}^\top \tag{12}$$
$$= \boldsymbol{S}_{\boldsymbol{P}_1\boldsymbol{X}}^\top \boldsymbol{S_X} \boldsymbol{S}_{\boldsymbol{X}}^\top \tag{13}$$
$$= \boldsymbol{S}_{\boldsymbol{P}_1\boldsymbol{X}}^\top \tag{14}$$

We now use the fact that sorting is permutation-invariant, so for any $\boldsymbol{P}, \boldsymbol{S_X X} = \boldsymbol{S_{PX} P X}$. The second condition is:

$$\boldsymbol{P}_1\boldsymbol{X} = \boldsymbol{P}_2\boldsymbol{X} \tag{15}$$
$$= \boldsymbol{S}_{\boldsymbol{P}_1\boldsymbol{X}}^\top \boldsymbol{S_X X} \tag{16}$$
$$= \boldsymbol{S}_{\boldsymbol{P}_1\boldsymbol{X}}^\top \boldsymbol{S}_{\boldsymbol{P}_1\boldsymbol{X}} \boldsymbol{P}_1\boldsymbol{X} \tag{17}$$
$$= \boldsymbol{P}_1\boldsymbol{X} \tag{18}$$

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 IMPLICIT DIFFERENTATION DERIVATION

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 full implicit differentiation and present the simplifications for the Hessian afterwards.

To simplify notation, we combine the target vector with the encoder parameters into $\Theta = (\boldsymbol{z}, \boldsymbol{\theta})$. We repeat the optimality condition for convenience:

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

This implicitly defines a minimizer function $\boldsymbol{Y}^*(\Theta)$. The implicit function theorem [47] states its Jacobian $\partial \boldsymbol{Y}^*(\Theta)/\partial\Theta$ exists if F is continuously differentiable and the Jacobian $\partial F(\boldsymbol{Y}^*, \Theta)/\partial \boldsymbol{Y}^*$ is a square invertible matrix. Using F 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 F 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 $\Theta \in \{\boldsymbol{z}, \boldsymbol{\theta}\}$ and reorder to obtain:

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

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

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 $\boldsymbol{v}$. We can efficiently compute this by solving the much smaller linear system $\boldsymbol{A}\boldsymbol{u} = \nabla_{\boldsymbol{Y}^*}\mathcal{L}$ for $\boldsymbol{u}$. This allows us to compute the desired gradient with $\nabla_\Theta \mathcal{L} = (\nabla_{\boldsymbol{Y}^*}\mathcal{L})^\top \boldsymbol{X} = \boldsymbol{u}^\top \boldsymbol{A}\boldsymbol{X} = \boldsymbol{u}^\top \boldsymbol{B}$ (Blondel et al., 2021).

Instead of solving the linear system exactly, we use a much faster approximation by applying a fixed number of conjugate gradient steps. In our experiments, we get a solution close to machine precision with only three iterations. Another advantage of conjugate gradient is that we only need to compute the Hessian-vector product $\boldsymbol{A}\boldsymbol{u}$ instead of the full Hessian $\boldsymbol{A}$. Not only does this avoid storing $\boldsymbol{A}$ explicitly – which decreases memory requirements –, with reverse-mode autodiff like in PyTorch [] and Tensorflow [] it also improves parallelization: to compute the full Hessian explicitly we would have to differentiate each entry in F separately. 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 differentation in meta-learning [iMAML] and neural architecture search [iDARTS] 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 sec [].

Of course, in the main text these efficiency improvements are irrelevant since we simplify to not solve a linear system in the first place.

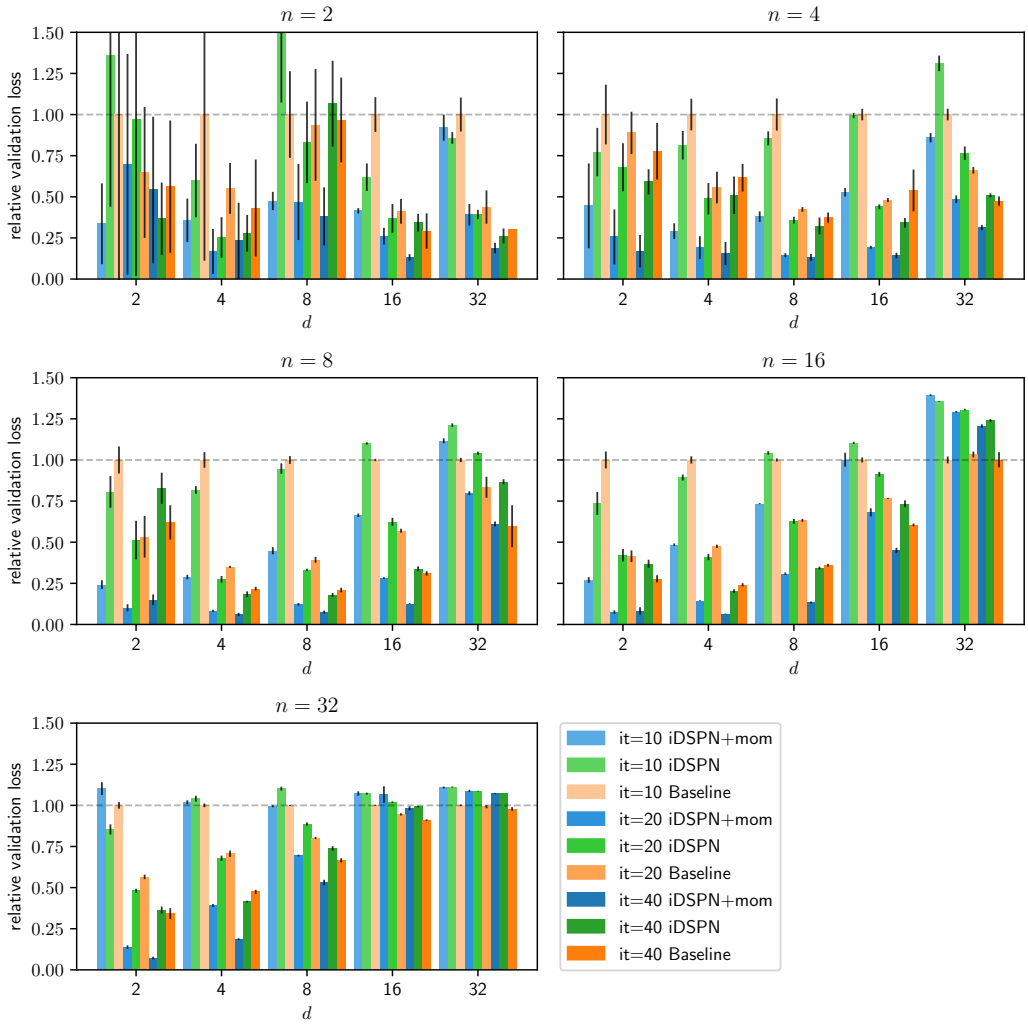
C Random sets full results

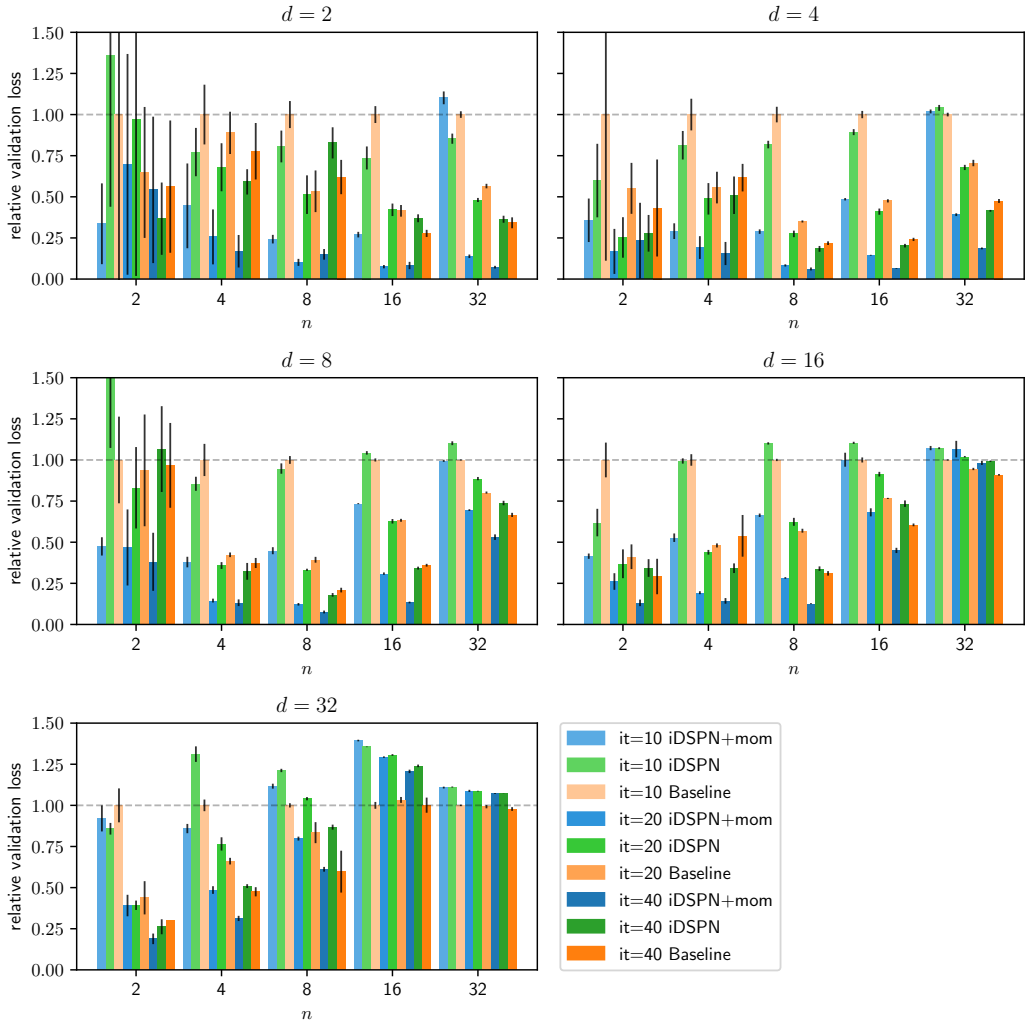
We had to exclude 10 runs of DSPN due to significantly worse results making their average uninformative. The following excluded DSPN runs were all using 40 iterations:

- 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 $n = 4, d = 32$.

We did not observe any stability issues with iDSPN, so we did not have to exclude any iDSPN runs.

TODO: how to read the plots, what the relative loss means


$n = 2$
$n = 4$
$n = 8$
$n = 16$
$n = 32$
relative validation loss
1.50
1.25
1.00
0.75
0.50
0.25
0.00
2
4
8
16
32
d
it=10 iDSPN+mom
it=10 iDSPN
it=10 Baseline
it=20 iDSPN+mom
it=20 iDSPN
it=20 Baseline
it=40 iDSPN+mom
it=40 iDSPN
it=40 Baseline


$d = 2$
$d = 4$
$d = 8$
$d = 16$
$d = 32$
relative validation loss
1.50
1.25
1.00
0.75
0.50
0.25
0.00
2
4
8
16
32
n
it=10 iDSPN+mom
it=10 iDSPN
it=10 Baseline
it=20 iDSPN+mom
it=20 iDSPN
it=20 Baseline
it=40 iDSPN+mom
it=40 iDSPN
it=40 Baseline

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

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 []:

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

It finds the optimal 1-to-1 mapping from each element in the prediction $\boldsymbol{Y}^{(1)} = [\boldsymbol{y}_1^{(1)}, \dots, \boldsymbol{y}_n^{(1)}]^\top$ to each element in the ground-truth $\boldsymbol{Y}^{(2)} = [\boldsymbol{y}_1^{(2)}, \dots, \boldsymbol{y}_n^{(2)}]^\top$, minimizing the total pairwise losses. This loss is permutation-invariant in both its arguments and is able to map duplicate predicted elements to different ground-truth elements, 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 the additional supervised loss term that DSPN proposes is not necessary for successful training.

D.1 NUMBERING WITHIN A CLASS

Dataset The input of every example is represented by 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 matrix of size 64×64 by counting the occurrences for each unique input class sequentially from top to bottom and represent the count as a 64 dimensional one-hot vector. 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.

Models In this experiment, we consider an element-wise classification problem, thus requiring that the model can assign an output element to every input element. Both the equivariant DeepSets version (instead of invariant) and BiLSTM naturally have this property. For the Transformer baselines we require an additional modification: The initial output set is based on an affine transformation $(\boldsymbol{Y}_0)_i = \boldsymbol{W}\boldsymbol{x}_i + \boldsymbol{b}$ of the input elements. In the Transformer with (random) PE baseline, we include the position encoding only for the initial output set and not for the input. Adding it to the input significantly degraded the sample efficiency. We adapt the iDSPN model for this setting, by concatenating the input to the initial output set. Since the output set is randomly sampled, we do not need to apply any element-wise matching for this. During the inner optimization of iDSPN, we only update the parts corresponding to the output set and keep the input fixed. Additionally, we apply projected gradient descent, by projecting each element back onto a valid point in the probability simplex after each gradient descent 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 if and only if every element is correct. The baselines are very volatile during training, further resulting in very large variances at the end of training. To reduce this variance, we pick the best model according to the validation accuracy which we evaluate 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

Dataset Every set has a fixed size with n elements of dimensionality d. As previously mentioned, every element is sampled i.i.d. from $\mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$. 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 Huber loss 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($d \to 512$)–ReLU–Linear($512 \to 512$)–FSPool (Zhang et al., 2020), where the linear layers are applied on each element in a multiset individually with shared weights across the elements.

Training All models are trained for 40 epochs at batch size 128 with Adam () 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 have any benefit from more training.

D.3 CLEVR Object Property Prediction

Where possible, we use exactly the same setup as DSPN. This includes the same dataset setup, the same evaluation procedure, a learning rate of 1e-3 with the Adam optimizer [], the same number of epochs, the same hidden and latent dimensionalities of 512.

We make a few changes to the training procedure of DSPN:

- Instead of using a relation network [] – 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 does 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 is Linear($19 \to 512$)–ReLU–Linear($512 \to 512$)–FSPool (Zhang et al., 2020). 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.
- 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.
- 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.

E EXAMPLE INPUTS AND OUTPUTS

TODO