FSPool: Learning Set Representations with Featurewise Sort Pooling

Yan Zhang
University of Southampton
Southampton, UK
yz5n12@ecs.soton.ac.uk

Jonathon Hare
University of Southampton
Southampton, UK
jsh2@ecs.soton.ac.uk

Adam Prügel-Bennett
University of Southampton
Southampton, UK
apb@ecs.soton.ac.uk

Abstract

Traditional set prediction models can struggle with simple datasets due to an issue we call the responsibility problem. We introduce a pooling method for sets of feature vectors based on sorting features across elements of the set to learn better set representations. This can be used to construct a permutation-equivariant auto-encoder, which avoids the responsibility problem. On a toy dataset of polygons and a set version of MNIST, we show that such an auto-encoder produces considerably better reconstructions. Used in set classification, FSPool significantly improves accuracy and convergence speed on the set versions of MNIST and CLEVR.

1 Introduction

The full version of this paper is available as [31]. In set auto-encoders, there are two main components: the *encoder* turns a set input into a feature vector, and the *decoder* turns a feature vector into a set output. In the encoder, a potential problem is the permutation-invariant pooling operation, which compresses a set of any size down to a single feature vector in one step. This can be a significant bottleneck in what these functions can represent efficiently, particularly when relations between elements of the set need to be modeled [17; 32]. In the decoder, the usual approach is to use an MLP or RNN to predict the set. However, there are very simple set datasets where this approach fails completely, which is due to a problem we call the *responsibility problem*.

1. We identify the *responsibility problem* when using standard neural networks to predict sets (section 2). This is a fundamental set prediction problem that has not been considered before in existing literature.
2. We propose FSPool: a differentiable, sorting-based pooling method for variable-size sets (section 3). By using our pooling in a set encoder and *inverting the sorting* in the decoder, we avoid the responsibility problem and can train this auto-encoder with an MSE loss instead of an assignment-based loss (e.g. Chamfer loss).
3. We show on two experiments that our model is more effective than traditional set auto-encoders that use MLP decoders. In several classification experiments, we replace the pooling function in an existing set encoder and consistently obtain improved results.

2 Responsibility problem

Suppose we have a dataset that contains sets of four 2d points (x-y coordinates) forming squares. The only variable is the rotation of these squares. Let us look at one square with some arbitrary initial rotation (see Figure 1). Each pair in the 8 outputs of an MLP decoder is *responsible* for producing one of the points in this square, which we mark with different colours in the figure.

Sets & Partitions Workshop at 33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

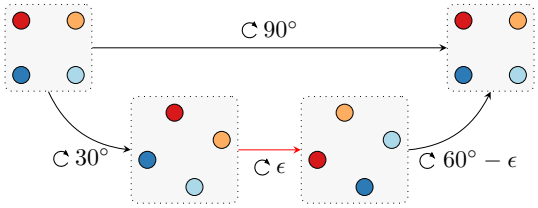

Figure 1: Discontinuity (red arrow) when rotating the set of points. The colours are used to denote which output of the network is responsible for which point. In this example, after $30°$ and a further small clockwise rotation by ϵ, the point that an output pair is responsible for has to suddenly change.

It turns out that standard neural networks struggle with modeling symmetries that arise because there are $n!$ different list representations of the same set. If we rotate the square (top left in figure) by 90 degrees (top right in figure), we simply permute the elements within the set. They are the same set, so they also encode to the same latent representation and produce the same list representation. This means that each output is still responsible for producing the point at the same position after the rotation, i.e. the dark red output is still responsible for the top left point, the light red output is responsible for the top right point, etc. However, this also means that at some point during that 90 degree rotation, *there must exist a discontinuous jump* (red arrow in figure) in how the outputs are assigned. We know that the 90 degree rotation must start and end with the top left point being produced by the dark red output. Thus, we know that there is a point where all the outputs must simultaneously change which point they are responsible for, so that completing the rotation results in the top left point being produced by the dark red output (bottom path in figure). *Even though we change the set continuously, the list representation (MLP outputs) must change discontinuously.*

This is a challenge for neural networks to learn, since they can typically only model functions without discontinuous jumps. As we move from squares to polygons with more vertices, it must learn an increasing frequency of situations where all the outputs must discontinuously change at once, which becomes very difficult to model (see experiment in subsection 4.1).

This example highlights a more general issue: whenever there are at least two set elements that can be smoothly interchanged, these discontinuities arise. For example, the bounding boxes in object detection can be interchanged in much the same way as the points of our square here. An MLP or RNN that tries to generate these must handle which of its outputs is responsible for what element in a discontinuous way.

3 Featurewise Sort Pooling

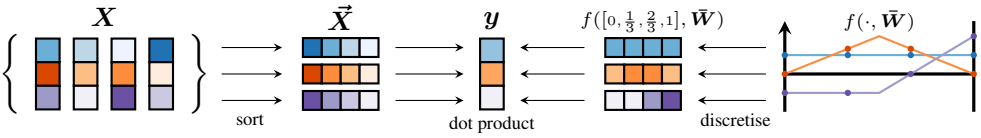

Figure 2: Overview of our FSPool model for variable-sized sets. In this example, the weights define piecewise linear functions with two pieces. The four dots on each line correspond to the positions where f is evaluated for a set size of four.

The main idea behind our pooling method is simple: sorting each feature across the elements of the set and performing a weighted sum. The numerical sorting ensures the property of permutation-invariance. The difficulty lies in how to determine the weights for the weighted sum in a way that works for variable-sized sets.

A key insight for auto-encoding is that we can store the permutation that the sorting applies in the encoder and apply the inverse of that permutation in the decoder. This allows the model to restore the arbitrary order of the set element so that it no longer needs an assignment-based loss for training. This avoids the problem in Figure 1, because rotating the square by $90°$ also permutes the outputs of

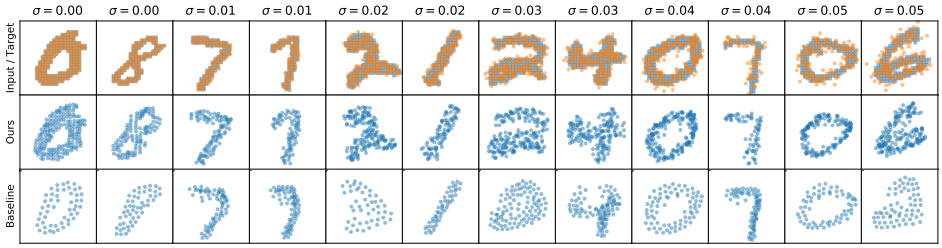

Figure 3: MNIST as point sets with different amounts of Gaussian noise (σ) and their reconstructions of a model trained with MSE loss or Chamfer loss. Orange points denote the noisy input that the model sees. The baseline uses sum pooling. Images are not cherry-picked.

the network accordingly. Thus, there is no longer a discontinuity in the outputs during this rotation. In other words, we make the auto-encoder permutation-*equivariant*: permuting the input set also permutes the neural network's output in the same way.

Due to space reasons, we describe the details of how our model works in Appendix A.

4 Experiments

Further experimental details can be found in Appendix E.

4.1 Rotating polygons

We start with our simple dataset of auto-encoding regular polygons, with each point in a set corresponding to the x-y coordinate of a vertex in that polygon. This dataset is designed to explicitly test whether the responsibility problem occurs in practice. We keep the set size the same within a training run and only vary the rotation. We try this with set sizes of increasing powers of 2.

Results First, we verified that if the latent space is always zeroed out, the model with FSPool is unable to train, suggesting that the latent space is being used and is necessary. For our training runs with set sizes up to 128, our auto-encoder is able to reconstruct the point set close to perfectly (see Appendix C). Meanwhile, the baseline converges significantly slower with high reconstruction error when the number of points is low (8 or fewer) and outputs the same set irrespective of input above that, regardless of loss function. We verified that for 4 points, the baseline shows the discontinuous jump behaviour as we predict in Figure 1. This experiment highlights the difficulty of learning this simple dataset with traditional approaches due to the responsibility problem.

4.2 Noisy MNIST sets

Next, we turn to the harder task of auto-encoding MNIST images – turned into sets of points – using a denoising auto-encoder. Each pixel that is above the mean pixel level is considered to be part of the set with its x-y coordinates as feature, scaled to be within the range of [0, 1]. The set size varies between examples and is 133 on average. We add Gaussian noise to the points in the set and use the set without noise as training target for the denoising auto-encoder.

Results We show example outputs of trained networks for various noise levels in Figure 3 where the baseline uses an MLP decoder. Our validation MSE loss for $\sigma = 0.00$ is **0.00022**, while the MLP baseline gets 0.00043 with sum pooling, 0.00130 with max pooling, and 0.00136 with mean pooling. Replacing the MLP decoder with an LSTM decoder (this matches the approach in [23]) gets a loss of 0.00056. In general, our model can reconstruct the digits much better than the baseline, which tends to predict too few points even though it always has 342 (the maximum set size) times 2 outputs available. Occasionally, the baseline makes big errors such as turning a 2 into a 3 ($\sigma = 0.02$) or a 6 into a 2 ($\sigma = 0.05$). These suggest that the baseline is perhaps memorising what a digit should look like and reconstructing only a template based on the (possibly wrongly) recognised digit.

Table 1: Classification accuracy (mean ± std) on MNIST $\sigma = 0.05$ over 6 runs (different pre-trained networks between runs). Frozen: training with frozen pre-trained auto-encoder weights. Unfrozen: unfrozen auto-encoder weights (fine-tuning). Random init: auto-encoder weights not used.

	1 epoch of training			10 epochs of training		
	Frozen	Unfrozen	Random init	Frozen	Unfrozen	Random init
FSPool	**83.4**%$_{\pm 2.0}$	**87.7**%$_{\pm 1.1}$	**84.8**%$_{\pm 1.3}$	**85.3**%$_{\pm 1.4}$	**91.6**%$_{\pm 0.8}$	**91.9**%$_{\pm 0.5}$
Sum	74.9%$_{\pm 2.0}$	70.7%$_{\pm 6.1}$	30.9%$_{\pm 4.9}$	79.3%$_{\pm 1.6}$	82.3%$_{\pm 2.2}$	70.5%$_{\pm 4.5}$
Mean	26.7%$_{\pm 4.5}$	32.8%$_{\pm 11.0}$	30.3%$_{\pm 1.4}$	37.0%$_{\pm 6.0}$	64.0%$_{\pm 25.9}$	71.9%$_{\pm 2.9}$
Max	73.4%$_{\pm 1.4}$	72.2%$_{\pm 1.3}$	53.0%$_{\pm 3.3}$	77.3%$_{\pm 1.2}$	80.2%$_{\pm 1.2}$	75.6%$_{\pm 1.4}$

Table 2: Mean and standard deviation of accuracy after 350 epochs, epochs to reach an accuracy milestone, and time required with a 1080 Ti GPU on CLEVR over 10 runs. The entry marked with * is an average over 8 instead of 10 runs because 2 runs did not reach 99% accuracy.

		Epochs to reach accuracy			Time for
Model	Accuracy	98.00%	98.50%	99.00%	350 epochs
FSPool	**99.27**%$_{\pm 0.18}$	**141**$_{\pm 5}$	**166**$_{\pm 16}$	**209**$_{\pm 33}$	8.8 h
RN	98.98%$_{\pm 0.25}$	144$_{\pm 6}$	189$_{\pm 29}$	*268$_{\pm 46}$	15.5 h
Sum	99.05%$_{\pm 0.17}$	146$_{\pm 13}$	191$_{\pm 40}$	281$_{\pm 56}$	8.0 h
Mean	98.96%$_{\pm 0.27}$	169$_{\pm 6}$	225$_{\pm 31}$	273$_{\pm 33}$	8.0 h
Max	96.99%$_{\pm 0.26}$	–	–	–	8.0 h

4.2.1 Classification

Instead of only auto-encoding MNIST sets, we can also classify them. We use the same dataset and replace the set decoder in our model and the baseline with a 2-layer MLP classifier. We consider three variants: using the trained auto-encoder weights for the encoder and freezing them, not freezing them (finetuning), and training all weights from random initialisation. This tests how informative the learned representations of the pre-trained auto-encoder and the encoder are.

Results We show our results for $\sigma = 0.05$ in Table 1 (relative results are largely unchanged for $\sigma = 0.00$). Even though our model could store information in the permutation that skips the latent space, our latent space contains more information to correctly classify a set. Our model with fixed encoder weights performs better after 1 epoch of training than the baseline model with unfrozen weights after 10 epochs of training. When allowing the encoder weights to change (Unfrozen and Random init), our results again improve significantly over the baseline.

4.3 CLEVR

CLEVR [13] is a visual question answering dataset where the task is to classify an answer to a question about an image. The images show scenes of 3D objects with different attributes, and the task is to answer reasoning questions such as "what size is the sphere that is left of the green thing". Since we are interested in sets, we use this dataset with the ground-truth state description – the set of objects (maximum size 10) and their attributes – as input instead of an image of the rendered scene.

Results Over 10 runs, Table 2 shows that our FSPool model is superior to the strong RN baseline in all the metrics considered. That is, it reaches a better final accuracy at 350 epochs and it reaches the listed accuracy milestones in fewer epochs. We show some of the learned functions $f(\cdot, \bar{W})$ in Appendix F. These confirm that FSPool uses more complex functions than just sums or maximums.

4.4 Graph classification

In Appendix D, we describe our experiments on graph classification where FSPool is used as graph readout. In summary, FSPool improves over a strong baseline on 6 out of 9 datasets.

References

[1] 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.

[2] Cem Anil, James Lucas, and Roger Grosse. Sorting out Lipschitz function approximation. arXiv:1811.05381, 2018.

[3] James Atwood and Don Towsley. Diffusion-convolutional neural networks. In *Advances in Neural Information Processing Systems 29 (NeurIPS)*, 2016.

[4] Cătălina Cangea, Petar Veličković, Nikola Jovanović, Thomas Kipf, and Pietro Liò. Towards sparse hierarchical graph classifiers. *NeurIPS Workshop*, Relational Representation Learning, 2018.

[5] Moustapha Cisse, Piotr Bojanowski, Edouard Grave, Yann Dauphin, and Nicolas Usunier. Parseval Networks: Improving robustness to adversarial examples. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 2017.

[6] Haoqiang Fan, Hao Su, and Leonidas J. Guibas. A point set generation network for 3D object reconstruction from a single image. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[7] Matthias Fey, Jan Eric Lenssen, Frank Weichert, and Heinrich Müller. SplineCNN: Fast geometric deep learning with continuous B-spline kernels. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[8] Hongyang Gao and Shuiwang Ji. Graph U-Net, 2019. URL `https://openreview.net/forum?id=HJePRoAct7`.

[9] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010.

[10] Aditya Grover, Eric Wang, Aaron Zweig, and Stefano Ermon. Stochastic optimization of sorting networks via continuous relaxations. In *International Conference on Learning Representations (ICLR)*, 2019.

[11] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Andrea Vedaldi. Gather-Excite: Exploiting feature context in convolutional neural networks. In *Advances in Neural Information Processing Systems 31 (NeurIPS)*, 2018.

[12] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. Spatial transformer networks. In *Advances in Neural Information Processing Systems 28 (NeurIPS)*, 2015.

[13] Daniel D. Johnson. Learning graphical state transitions. In *International Conference on Learning Representations (ICLR)*, 2017.

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

[15] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[16] Gonzalo Mena, David Belanger, Scott Linderman, and Jasper Snoek. Learning Latent Permutations with Gumbel-Sinkhorn Networks. In *International Conference on Learning Representations (ICLR)*, 2018.

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

[18] Marion Neumann, Roman Garnett, Christian Bauckhage, and Kristian Kersting. Propagation kernels: Efficient graph kernels from propagated information. *Machine Learning*, 102(2): 209–245, 2016. ISSN 0885-6125.

[19] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, 2016.

[20] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[21] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research*, 12: 2539–2561, 2011. ISSN 1532-4435.

[22] Zenglin Shi, Yangdong Ye, and Yunpeng Wu. Rank-based pooling for deep convolutional neural networks. *Neural Networks*, 83:21 – 31, 2016. ISSN 0893-6080.

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

[24] Sean Welleck, Zixin Yao, Yu Gai, Jialin Mao, Zheng Zhang, and Kyunghyun Cho. Loss functions for multiset prediction. In *Advances in Neural Information Processing Systems 31 (NeurIPS)*, 2018.

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

[26] Yaoqing Yang, Chen Feng, Yiru Shen, and Dong Tian. FoldingNet: Point cloud auto-encoder via deep grid deformation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[27] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. In *Advances in Neural Information Processing Systems 31 (NeurIPS)*, 2018.

[28] Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. GraphRNN: Generating realistic graphs with deep auto-regressive models. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018.

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

[30] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[31] Yan Zhang, Jonathon Hare, and Adam Prügel-Bennett. FSPool: Learning set representations with featurewise sort pooling. *arXiv:1906.02795*, 2019. URL `https://arxiv.org/abs/1906.02795`.

[32] Yan Zhang, Jonathon Hare, and Adam Prügel-Bennett. Learning representations of sets through optimized permutations. In *International Conference on Learning Representations (ICLR)*, 2019.

A Full Model

A.1 Fixed-size sets

We are given a set of n feature vectors $\boldsymbol{X} = [\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)}]$ where each $\boldsymbol{x}^{(i)}$ is a column vector of dimension d placed in some arbitrary order in the columns of $\boldsymbol{X} \in \mathbb{R}^{d \times n}$. From this, the goal is to produce a single feature vector in a way that is invariant to permutation of the columns in the matrix.

We first sort each of the d features across the elements of the set by numerically sorting within the rows of $\boldsymbol{X}$ to obtain the matrix of sorted features $\vec{\boldsymbol{X}}$:

$$\vec{X}_{i,j} = \textsc{Sort}(\boldsymbol{X}_{i,:})_j \tag{1}$$

where $\boldsymbol{X}_{i,:}$ is the ith row of $\boldsymbol{X}$ and $\textsc{Sort}(\cdot)$ sorts a vector in descending order. While this may appear strange since the columns of $\vec{\boldsymbol{X}}$ no longer correspond to individual elements of the set, there are good reasons for doing this. A transformation (such as with an MLP) prior to the pooling can ensure that the features being sorted are mostly independent so that little information is lost by treating the features independently. There are efficient parallelised SORTs in Deep Learning frameworks such as PyTorch.

Then, we apply a learnable weight matrix $\boldsymbol{W} \in \mathbb{R}^{d \times n}$ to $\vec{\boldsymbol{X}}$ by elementwise multiplying and summing over the columns (row-wise dot products).

$$y_i = \sum_j^n W_{i,j} \vec{X}_{i,j} \tag{2}$$

$\boldsymbol{y} \in \mathbb{R}^d$ is the final pooled representation of $\vec{\boldsymbol{X}}$. The weight vector allows different weightings of different ranks and is similar in spirit to the parametric version of the gather step in Gather-Excite [11]. This is a generalisation of both max and sum pooling, since max pooling can be obtained with the weight vector $[1, 0, \ldots, 0]$ and sum pooling can be obtained with the $\mathbf{1}$ vector. Thus, it is also a powerful pooling method for multi-sets [25] while being more flexible [17] in what it can represent.

A.2 Variable-size sets

When the size n of sets can vary, our previous weight matrix can no longer have a fixed number of columns. To deal with this, we define a *continuous* version of the weight vector in each row: we use a fixed number of weights to parametrise a piecewise linear function $f : [0, 1] \to \mathbb{R}$, also known as calibrator function [12]. For a set of size three, this function would be evaluated at 0, 0.5, and 1 to determine the three weights for the weighted sum. For a set of size four, it would be evaluated at 0, 1/3, 2/3, and 1. This decouples the number of columns in the weight matrix from the set size that it processes, which allows it to be used for variable-sized sets.

To parametrise a piecewise linear function f, we have a weight vector $\bar{\boldsymbol{w}} \in \mathbb{R}^k$ where $k - 1$ is the number of pieces defined by the k points. With the ratio $r \in [0, 1]$,

$$f(r, \bar{\boldsymbol{w}}) = \sum_{i=1}^{k} \max(0, 1 - |r(k-1) - (i-1)|)\bar{w}_i \tag{3}$$

The $\max(\cdot)$ term selects the two nearest points to r and linearly interpolates them. For example, if $k = 3$, choosing $r \in [0, 0.5]$ interpolates between the first two points in the weight vector with $(1 - 2r)w_0 + 2rw_1$.

We have a different $\bar{\boldsymbol{w}}$ for each of the d features and place them in the rows of a weight matrix $\bar{\boldsymbol{W}} \in \mathbb{R}^{d \times k}$, which no longer depends on n. Using these rows with f to determine the weights:

$$y_i = \sum_{j=1}^{n} f(\frac{j-1}{n-1}, \boldsymbol{W}_{i,:}) \vec{X}_{i,j} \tag{4}$$

$\boldsymbol{y}$ is now the pooled representation with a potentially varying set size n as input. When $n = k$, this reduces back to Equation 2. In this paper, we fixed $k = 20$ for all experiments without tuning it.

A.3 Auto-encoder

To create an auto-encoder, we need a decoder that turns the latent space back into a set. Analogously to image auto-encoders, we want this decoder to roughly perform the operations of the encoder in reverse. The FSPool in the encoder has two parts: sorting the features, and pooling the features. Thus, the FSUnpool version should "unpool" the features, and "unsort" the features. For the former, we define an unpooling version of Equation 4 that distributes information from one feature vector to a variable-size list of feature vectors. For the latter, the idea is to store the permutation of the sorting from the encoder and use the inverse of it in the decoder to unsort it. This allows the auto-encoder to restore the original ordering of set elements, which makes it permutation-equivariant.

With $\boldsymbol{y'} \in \mathbb{R}^d$ as the vector to be unpooled, we define the unpooling similarly to Equation 4 as

$$\vec{X}'_{i,j} = f(\frac{j-1}{n-1}, \boldsymbol{W}'_{i,:})y'_i \tag{5}$$

In the non-autoencoder setting, the lack of differentiability of the permutation is not a problem due to the pathwise differentiability. However, in the auto-encoder setting we make use of the permutation in the decoder. While gradients can still be propagated through it, it introduces discontinuities whenever the sorting order in the encoder for a set changes, which we empirically observed to be a problem for successful learning. To avoid this issue, we use the recently proposed sorting networks [10], which provide a continuous relaxation of numerical sorting. This gives us a differentiable approximation of a permutation matrix $\boldsymbol{P}_i \in [0, 1]^{n \times n}, i \in \{1, \dots, d\}$ for each of the d features, which we can use in the decoder while still keeping the model fully differentiable. It comes with the trade-off of increased computation costs with $O(n^2)$ time and space complexity, so we only use the relaxed sorting in the auto-encoder setting. It is possible to decay the temperature of the relaxed sort throughout training to 0, which allows the more efficient traditional sorting algorithm to be used at inference time.

Lastly, we can use the inverse of the permutation from the encoder to restore the original order.

$$X'_{i,j} = (\vec{\boldsymbol{X}}'_{i,:}\boldsymbol{P}_i^T)_j \tag{6}$$

where $\boldsymbol{P}_i^T$ permutes the elements of the ith row in $\vec{X}'$.

Because the permutation is stored and used in the decoder, this makes our auto-encoder similar to a U-net architecture [15] since it is possible for the network to skip the small latent space. Typically we find that this only starts to become a problem when d is too big, in which case it is possible to only use a subset of the P_i in the decoder to counteract this.

B Related Work

We are proposing a differentiable function that maps a *set* of feature vectors to a single feature vector. This has been studied in many works such as Deep Sets [29] and PointNet [20], with universal approximation theorems being proven. In our notation, the Deep Sets model is $g(\sum_j h(\boldsymbol{X}_{:,j}))$ where $h : \mathbb{R}^d \to \mathbb{R}^p$ and $g : \mathbb{R}^p \to \mathbb{R}^q$. Since this is $O(n)$ in the set size n, it is clear that while it may be able to approximate any set function, problems that depend on higher-order interactions between different elements of the set will be difficult to model aside from pure memorisation. This explains the success of relation networks (RN), which simply perform this sum over all *pairs* of elements, and has been extended to higher orders in Murphy et al. [17]. Our work proposes an alternative operator to the sum that is intended to allow some relations between elements to be modeled through the sorting, while not incurring as large of a computational cost as the $O(n^2)$ complexity of RNs.

Sorting-based set functions The use of sorting has often been considered in the set learning literature due to its natural way of ensuring permutation-invariance. The typical approach is to sort elements of the set as units rather than our approach of sorting each feature individually.

For example, the similarly-named SortPooling [30] sorts the elements based on one feature of each element. However, this introduces discontinuities into the optimisation whenever two elements swap positions after the sort. For variable-sized sets, they simply truncate (which again adds discontinuities) or pad the sorted list to a fixed length and process this with a CNN, treating the sorted vectors as a sequence. Similarly, Cangea et al. [4] and Gao & Ji [8] truncate to a fixed-size set by computing a score for each element and keeping elements with the top-k scores. In contrast, our pooling handles variable set sizes without discontinuities through the featurewise sort and continuous weight space. Gao & Ji [8] propose a graph auto-encoder where, in the decoder, they use the "inverse" of what the top-k operator does in the encoder, which is comparable to our approach. Instead of numerically sorting, Mena et al. [16] and Zhang et al. [32] aim to *learn* an ordering of set elements.

Outside of the set learning literature, rank-based pooling in a convolutional neural network has been used in Shi et al. [22], where the rank is turned into a weight. Sorting within a single feature vector has been used for modeling more powerful functions under a Lipschitz constraint for Wasserstein GANs [2] and improved robustness to adversarial examples [5].

Set prediction Assignment-based losses combined with an MLP or similar are a popular choice for various auto-encoding and generative tasks on point clouds [6; 26; 1]. An interesting alternative approach is to perform the set generation sequentially [24; 13; 28]. The difficulty lies in how to turn the set into one or multiple sequences, which these papers to solve in different ways.

C Polygons

Table 3: Direct mean squared error (in hundredths) on Polygon dataset with different number of points in the set.

Set size	2	4	8	16	32	64
FSPool	0.000	0.001	0.000	0.000	0.000	0.0001
Random	100.323	100.134	99.367	99.951	99.438	99.523

Table 4: Chamfer loss (in hundredths) on Polygon dataset with different number of points in the set.

Set size	2	4	8	16	32	64
FSPool	0.001	0.001	0.001	0.000	0.001	0.002
MLP + Chamfer	1.189	1.771	0.274	1.272	0.316	0.085
MLP + Hungarian	1.517	0.400	0.251	1.266	0.326	0.081
Random	72.848	19.866	5.112	1.271	0.322	0.081

Table 5: Linear assignment loss (in hundredths) on Polygon dataset with different number of points in the set.

Set size	2	4	8	16	32	64
FSPool	0.000	0.001	0.000	0.000	0.000	0.001
MLP + Chamfer	0.595	0.885	0.137	0.641	0.160	0.285
MLP + Hungarian	0.758	0.200	0.126	0.634	0.163	0.040
Random	36.424	9.933	2.556	0.635	0.161	0.041

In Table 3, Table 4, and Table 5, we show the results of various model and training loss combinations. We include a random baseline that outputs a polygon with the correct size and centre, but random rotation.

These show that FSPool with the direct MSE training loss is clearly better than the baseline with either linear assignment or Chamfer loss on all the evaluation metrics. When the set size is 16 or greater, the other combinations only perform as well as the random baseline because they output the same constant set regardless of input.

D Graph classification

Experimental setup The datasets and node features used are the same as in GIN; we did not cherry-pick them. Because the social network datasets are purely structural without node features, a constant 1 feature is used on the RDT datasets and the one-hot-encoded node degree is used on the other social network datasets.

We follow the standard methodology on these graph classification datasets of performing 10-fold cross validation, and repeat this with 10 different random seeds (100 runs of every hyperparameter combination on every dataset). The hyperparameter sweep is done based on best validation accuracy and over the same combinations as in GIN.

Note that in GIN, hyperparameters are selected based on best *test* accuracy. This is a problem, because they consider the number of epochs a hyperparameter when accuracies tend to significantly vary between individual epochs. For example, our average result on the PROTEINS dataset would change from 73.4% to 77.1% if we were to select based on best test accuracy, which would be better than their 76.2%.

Results We show our results of GIN-FSPool and the GIN baseline averaged over 10 repeats in Table 6. On the majority of datasets, FSPool has slightly better accuracies than the strong baseline and consistently takes fewer epochs to reach its highest validation accuracy. On the two RDT datasets, this improvement is large. Interestingly, these are the two datasets where the number of nodes to be pooled is by far the largest with an average of 400+ nodes per graph, compared to the next largest COLLAB with an average of 75 nodes. This is evidence that FSPool is helping to avoid the bottleneck problem of pooling a large set of feature vectors to a single feature vector. On the two datasets where FSPool was statistically significantly worse, using $k = 5$ instead of $k = 20$ eliminates the gap in accuracy between the two models.

We emphasise that the main comparison to be made is between the GIN-Base and the GIN-FSPool model, since that is the only comparison where the only factor of difference is the pooling method. When comparing against other models, the network architecture, training hyperparameters, and evaluation methodology can differ significantly.

Keep in mind that while GIN-Base looks much worse than the original GIN-Base*, the difference is that our implementation has hyperparameters properly selected by validation accuracy, while GIN-Base* selected them by test accuracy. If we were to select based on test accuracy, our implementation frequently outperforms their results. Also, they only performed a single run of 10-fold cross-validation.

E Experimental details

For all experiments, we used FSPool and the unpooling version of it with $k = 20$. We guessed this value without tuning, and we did not observe any major differences when we tried to change this on CLEVR once. $\bar{\boldsymbol{W}}$ can be initialised in different ways, such as by sampling from a standard Gaussian. However, for the purposes of starting the model as similarly as possible to the sum pooling baseline on CLEVR and on the graph classification datasets, we initialise $\bar{\boldsymbol{W}}$ to a matrix of all 1s on them.

E.1 Polygons

The polygons are centred on 0 with a radius of 1. The points in the set are randomly permuted to remove any ordering in the set from the generation process that a model that is not permutation-invariant or permutation-equivariant could exploit. We use a batch size of 16 for all three models and train it for 10240 steps. We use the Adam optimiser [14] with 0.001 learning rate and their suggested values for the other optimiser parameters (PyTorch defaults). Weights of linear and convolutional layers are initialised as suggested in Glorot & Bengio [9]. The size of every hidden layer is set to 16 and the latent space is set to 1 (it should only need to store the rotation as latent variable).

Table 6: Cross-validation classification results (%) on various commonly-used graph classification datasets, with the mean cross-validation accuracy averaged over 10 repeats and sample standard deviations (±). Hyperparameters of entries marked with * are known to be selected based on test accuracy instead of validation accuracy, so results are likely not comparable to other existing approaches that were (hopefully) selected based on validation accuracy. Our results were selected based on validation accuracy.

Social Network	IMDB-B	IMDB-M	RDT-B	RDT-M5K	COLLAB
Num. graphs	1000	1500	2000	5000	5000
Num. classes	2	3	2	5	3
Avg. nodes	19.8	13.0	429.6	508.5	74.5
Max. nodes	136	89	3063	2012	492
WL* [25]	73.8	50.9	81.0	52.5	78.9
DCNN [3]	49.1	33.5	–	–	52.1
PATCHY-SAN [19]	71.0 ±2.3	45.2 ±2.8	86.3 ±1.6	49.1 ±0.7	72.6 ±2.2
SORTPOOL [30]	70.0 ±0.9	47.8 ±0.9	–	–	73.8 ±0.5
DIFFPOOL [27]	–	–	–	–	75.5
GIN-BASE* [25]	75.1	52.3	92.4	57.5	80.2
GIN-FSPOOL	**72.0** ±1.3	**49.6** ±1.6	**89.3**±1.4	**52.4**±1.1	79.4 ±0.6
- *epochs*	**52** ±46	**37** ±20	**155** ±58	**67**±23	**76**±47
GIN-BASE	71.7 ±0.9	48.8 ±2.0	84.7 ±2.3	48.1 ±1.3	**80.1**±0.5
- *epochs*	73 ±78	78 ±92	158 ±55	175 ±53	202 ±33

Bioinformatics	MUTAG	PROTEINS	PTC	NCI1
Num. graphs	188	1113	344	4110
Num. classes	2	2	2	2
Avg. nodes	17.9	39.1	25.5	29.8
Max. nodes	28	620	109	111
PK [30; 18]	76.0 ±2.7	73.7 ±0.7	59.5 ±2.4	82.5 ±0.5
WL [30; 21]	84.1 ±1.9	74.7 ±0.5	58.0 ±2.5	85.5 ±0.5
WL* [25]	90.4	75.0	59.9	86.0
DCNN [3]	67.0	61.3	56.6	62.6
PATCHY-SAN [19]	92.6 ±4.2	75.9 ±2.8	60.0 ±4.8	78.6 ±1.9
SORTPOOL [30]	85.8 ±1.7	75.5 ±0.9	58.6 ±2.5	74.4 ±0.5
DIFFPOOL [27]	–	76.3	–	–
GIN-BASE* [25]	89.4	76.2	64.6	82.7
GIN-FSPOOL	87.3 ±3.8	**73.4** ±2.9	**60.7** ±4.9	77.8 ±1.0
- *epochs*	**170** ±114	**62** ±48	206 ±102	**366** ±82
GIN-BASE	87.3 ±4.4	72.3 ±1.5	58.6 ±3.8	**79.3**±0.7
- *epochs*	217 ±119	92 ±47	**185** ±127	384 ±49

Table 7: Average of best hyperparameters over 10 repeats.

	IMDB-B	IMDB-M	RDT-B	RDT-M5K	COLLAB	MUTAG	PROTEINS	PTC	NCI1
GIN-FSPOOL									
- *dimensionality*	64.0	64.0	64.0	64.0	64.0	22.4	20.8	20.8	28.8
- *batch size*	72.8	100	59.2	59.2	72.8	32.0	60.8	60.8	128
- *dropout*	0.40	0.30	0.25	0.25	0.25	0.40	0.45	0.35	0.45
GIN-BASE									
- *dimensionality*	64.0	64.0	64.0	64.0	64.0	24.0	19.2	22.4	25.6
- *batch size*	86.4	86.4	38.8	86.4	100	60.8	60.8	41.6	128
- *dropout*	0.25	0.30	0.40	0.30	0.50	0.30	0.40	0.30	0.30

E.2 MNIST

We train on the training set of MNIST for 10 epochs and the shown results come from the test set of MNIST. For an image, the coordinate of a pixel is included if the pixel is above the mean pixel level of 0.1307 (with pixel levels ranging 0–1). Again, the order of the points are randomised. We did not include results of the linear assignment loss because we did not get the model to converge to results of similar quality to the direct MSE loss or Chamfer loss, and training time took too long (> 1 day) in order to find better parameters.

The latent space is increased from 1 to 16 and the size of the hidden layers is increased from 16 to 32. All other hyperparameters are the the same as for the Polygons dataset. For classification, we also train for 10 epochs. Therefore, the Fixed and the Unfrozen weights configurations receive a total of 20 epochs of training, since the weights that they start with have already been trained for 10 epochs.

E.3 CLEVR

The architecture and hyperparameters come from the third-party open-source implementation available at `https://github.com/mesnico/RelationNetworks-CLEVR`. For the RN baseline, the set is first expanded into the set of all pairs by concatenating the 2 feature vectors of the pair for all pairs of elements in the set. The question representation coming from the 256-unit LSTM, processing the question tokens in reverse with each token embedded into 32 dimensions, is concatenated to all elements in the set. Each element of this new set is first processed by a 4-layer MLP with 512 neurons in each layer and ReLU activations. The set of feature vectors is pooled with a sum and the output of this is processed with a 3-layer MLP (hidden sizes 512, 1024, and number of answer classes) with ReLU activations. A dropout rate of 0.05 is applied before the last layer of this MLP. Adam is used with a starting learning rate of 0.000005, which doubles every 20 epochs until the maximum learning rate of 0.0005 is reached. Weight decay of 0.0001 is applied. The model is trained for 350 epochs.

E.4 Graph classification

The GIN architecture starts with 5 sequential blocks of graph convolutions. Each block starts with summing the feature vector of each node's neighbours into the node's own feature vector. Then, an MLP is applied to the feature vectors of all the nodes individually. The details of this MLP were somewhat unclear in [25] and we chose Linear-ReLU-BN-Linear-ReLU-BN in the end. We tried Linear-BN-ReLU-Linear-BN-ReLU as well, which gave us slightly worse validation results for both the baseline and the FSPool version. The outputs of each of the 5 blocks are concatenated and pooled, either with a sum for the social network datasets, mean for the social network datasets (this is as specified in GIN), or with FSPool for both types of datasets. This is followed by BN-Linear-ReLU-Dropout-Linear as classifier with a softmax output and cross-entropy loss. We used the torch-geometric library [7] to implement this model.

The starting learning rate for Adam is 0.01 and is reduced every 50 epochs. Weights are initialised as suggested in [9]. The hyperparameters to choose from are: dropout ratio $\in \{0, 0.5\}$, batch size $\in \{32, 128\}$, if bioinformatics dataset hidden sizes of all layers $\in \{16, 32\}$ and 500 epochs, if social network dataset the hidden size is 64 and 250 epochs. Due to GPU memory limitations we used a batch size of 100 instead of 128 for social network datasets. The best hyperparameters are selected based on best average validation accuracy across the 10-fold cross-validation, where one of the 9 training folds is used as validation set each time. In other words, within one 10-fold cross-validation run the hyperparameters used for the test set are the same, while across the 10 repeats of this with different seeds the best hyperparameters may differ.

F Learned pooling functions

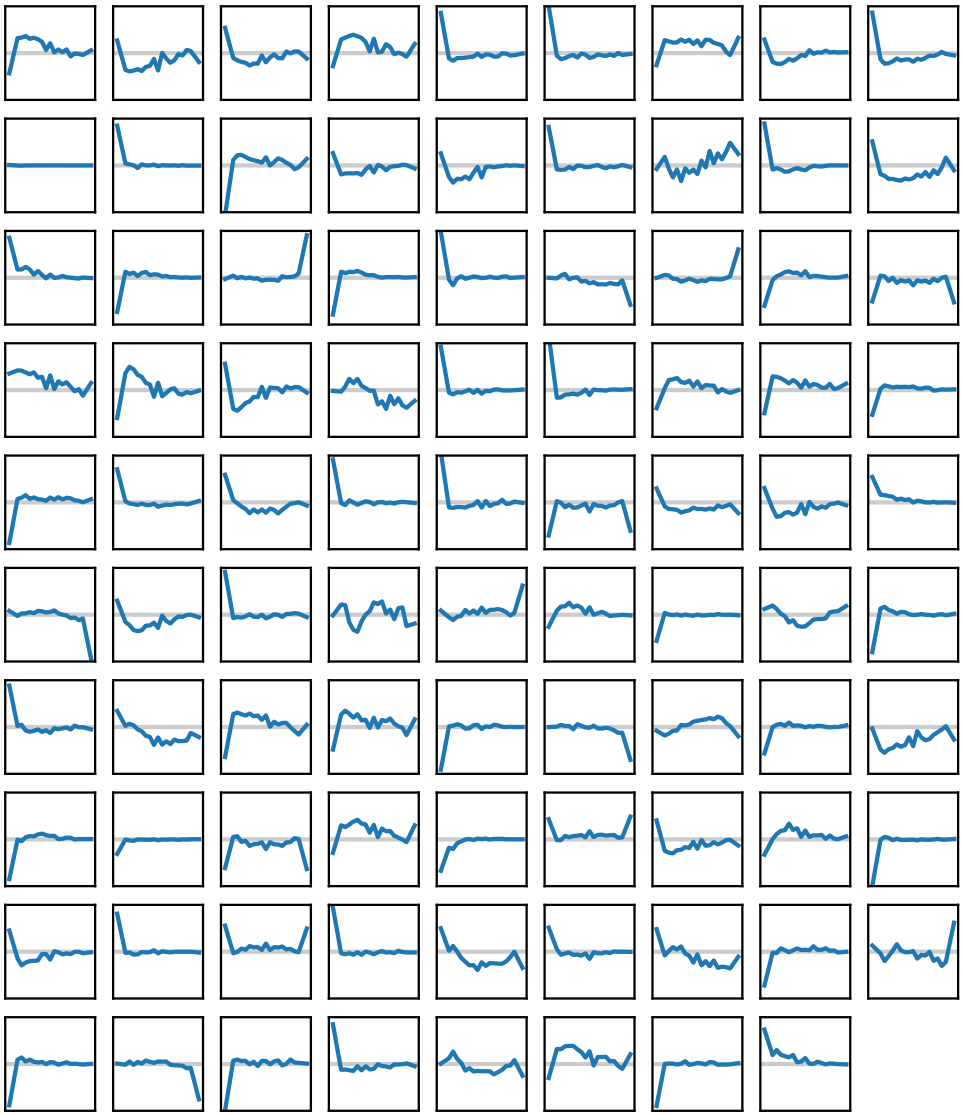

Figure 4: Shapes of piecewise linear functions learned by the FSPool model on CLEVR. These show $r \in [0, 1]$ on the x-axis and $f(r, \bar{w})$ on the y-axis for a particular $\bar{w}$ of a fully-trained model. A common shape among these functions are variants of max pooling: close to 0 weight for most ranks and a large non-zero weight on either the maximum or the minimum value, for example in row 2 column 2. There are many functions that simple maximums or sums can *not* easily represent, such as a variant of max pooling with the values slightly below the max receiving a weight of the opposite sign (see row 1 column 1) or the shape in the penultimate row column 5. The functions shown here may have a stronger tendency towards 0 values than normal due to the use of weight decay on CLEVR.