University of Southampton

Faculty of Engineering and Physical Sciences

Learning to Represent and Predict Sets with Deep Neural Networks

Yan Zhang

Thesis for the degree of Doctor of Philosophy

December, 2019

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

Electronics and Computer Science

Doctor of Philosophy

LEARNING TO REPRESENT AND PREDICT SETS WITH DEEP NEURAL NETWORKS

by Yan Zhang

In this thesis, we develop various techniques for working with *sets* in machine learning. Each input or output is not an image or a sequence, but a set: an unordered collection of multiple objects, each object described by a feature vector. Their unordered nature makes them suitable for modeling a wide variety of data, ranging from objects in images to point clouds to graphs. Deep learning has recently shown great success on other types of structured data, so we aim to build the necessary structures for sets into deep neural networks.

The first focus of this thesis is the learning of better set representations (sets as input). Existing approaches have bottlenecks that prevent them from properly modeling relations between objects within the set. To address this issue, we develop a variety of techniques for different scenarios and show that alleviating the bottleneck leads to consistent improvements across many experiments.

The second focus of this thesis is the prediction of sets (sets as output). Current approaches do not take the unordered nature of sets into account properly. We determine that this results in a problem that causes discontinuity issues with many set prediction tasks and prevents them from learning some extremely simple datasets. To avoid this problem, we develop two models that properly take the structure of sets into account. Various experiments show that our set prediction techniques can significantly benefit over existing approaches.

Contents

List of figures

List of tables

Declaration of authorship

I, Yan Zhang, declare that the thesis entitled *Learning to Represent and Predict Sets with Deep Neural Networks* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: [115, 116, 117, 118].

Signature: ______________________________

Date: ______________________________

Acknowledgements

Adam Prügel-Bennett
Jonathon Hare
Freddie Nash
My family

List of abbreviations

Terminology

CNN Convolutional neural network

GPU Graphics processing unit

MLP Multi-layer perceptron

MSE Mean squared error

NMS Non-maximum suppression

RNN Recurrent neural network

SGD Stochastic gradient descent

VQA Visual question answering

Datasets

CIFAR-10 Tiny image dataset [60]

CLEVR Synthetic VQA dataset [53]

MNIST Handwritten digits dataset [63]

VQA v1/v2 VQA datasets [6, 38]

Models

AE-CD/AE-EMD Set auto-encoder with different set losses [1]

BAN Bilinear attention network [56]

Faster R-CNN Region-based CNN object detector [86]

GIN Graph isomorphism network [105]

LSTM Long short-term memory RNN [44]

RN Relation network [89]

Chapter 1

Sets in machine learning

Sets are collections of things without a natural ordering to them. For example, the types of fruit that someone has eaten in the past week can be considered a *set* of fruit. While it may be possible to order them in some way (such as by name, date last eaten, tastiness, etc.), there is no inherently "correct" ordering. Talking about the nutritional value of the collective $\{\text{apple}, \text{raspberry}, \text{blackberry}\}$ is exactly the same as talking about $\{\text{raspberry}, \text{blackberry}, \text{apple}\}$. The order that they are written down in is different, but the set of fruits itself is the same. This is in contrast to data that do have a natural order, like the words in a sentence – a sentence can lose much of its meaning when the individual words are shuffled around.

Such sets are a natural way of describing different kinds of data in the real world. There are many problems of interest to the machine learning community that can be described in terms of sets: predicting the set of objects in an image is known as object detection; Lidar scanners on self-driving cars produce a set of 3d points of their surroundings to find obstacles; properties of molecules can be predicted based on the set of atoms and the set of bonds connecting those atoms. In all of these examples, a machine learning model either takes a set as input or produces a set as output.

In this thesis, we will focus on a specific type of machine learning method for working with sets: deep neural networks. In recent years, these have stood out as one of the most successful approaches on structured data like images and text. Part of this success is due to the building blocks that these deep neural networks are made of. These building blocks (like convolutions for images) take advantage of the structure in the data and allow the neural networks to learn much more efficiently.

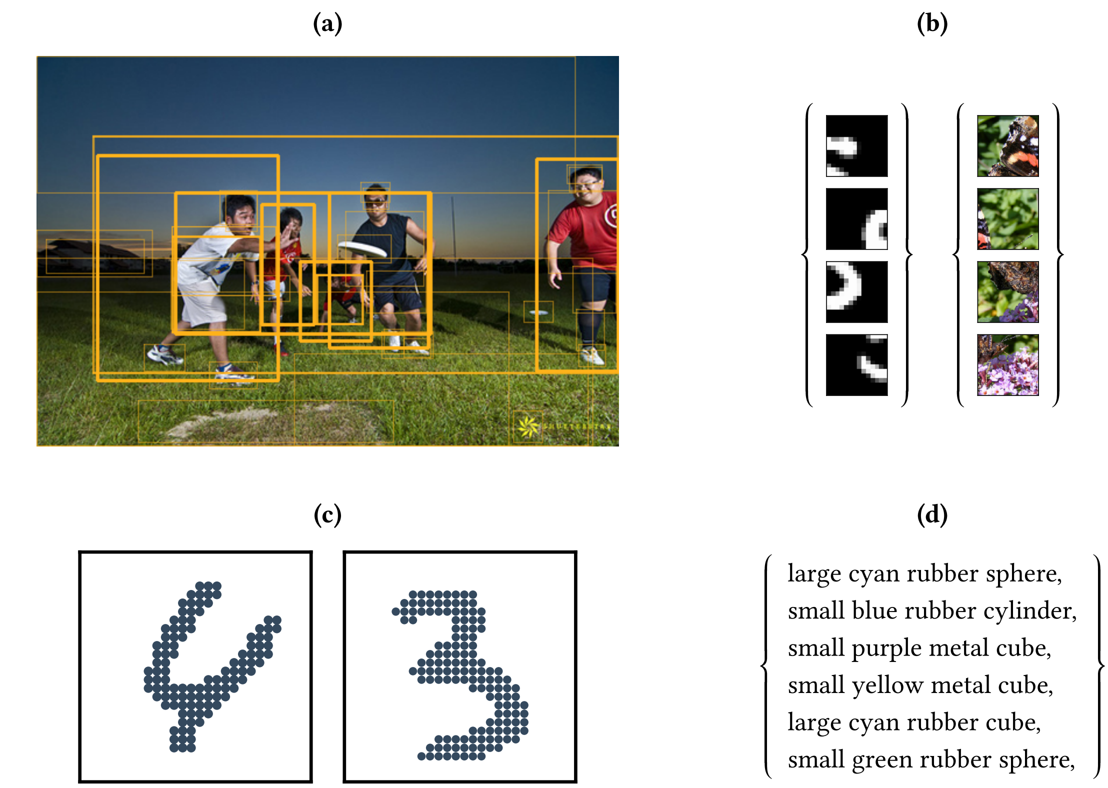


Figure 1.1: Example sets that we will work with throughout the thesis. **(a)** set of object proposals (Chapter 3). **(b)** set of image tiles (Chapter 4). **(c)** set of points in a point cloud (Chapter 5). **(d)** set of object attributes (Chapter 6).

With sets, this structure is given by their unordered nature. While there is existing work on neural network approaches to sets (which we briefly review in Chapter 2), the set domain has been studied to a much lesser extent than domains like images and text. What building blocks are needed to adequately handle sets with neural networks? This question is what this thesis explores by identifying shortcomings in existing techniques and developing new techniques for modeling sets with neural networks.

We begin by motivating our work with a benchmark task for Machine Intelligence: visual question answering. The task requires a machine to answer natural language questions about images, such as "how many people are there?" about the image in Figure 1.1 (a). These questions can be arbitrarily difficult, so the task probes the edge of what is possible with current machine learning techniques. While this task was initially defined with images as input, extracting and using the *set of objects* in the image has been found to lead to better results [4]. This makes visual question answering a task with set inputs. We identify a problem with existing visual question answering models in Chapter 3, which means that *counting questions* like the one above are virtually impossible to answer. We then propose a building block that fixes this issue. Our work on this task shows that sets can be quite interesting to work with and how existing methods for sets can fall short. This is what motivated us to continue researching ways of modeling sets with neural networks.

We move on from this specific application of sets to more general set problems. There are two main directions for this: a machine learning model can take a set as input or produce a set as output.

Sets as input In these tasks, the inputs are sets and the desired output is something else, like a classification or regression. In visual question answering, the input is the set of object proposals (along with a question) and the output is an answer. The objective is to extract as much relevant information out of the set as possible. We worked on two different methods for doing this:

- In Chapter 4, we propose a way of *learning to permute* the elements of the set into a list, which is easier to work with than the set. This is based on the assumption that there are some orderings that are easier to learn from [101], which our model tries to find.
- In Chapter 5, we propose a model that involves *sorting* the features in the set numerically. This allows the model to learn about the *distribution* of features, which can better capture relationships between set elements.

Sets as output In these tasks, we have some input and want to produce a set as output. In object detection, the input is an image and output is the set of objects in that image. In Chapter 5, we identify a problem that we call the *responsibility problem* with existing set prediction techniques, which can significantly hinder the learning of the neural network. We then propose a solution which works in the limited scenario of auto-encoders. From what we learned through this, we develop a general set prediction algorithm in Chapter 6 without the auto-encoder limitation. This is the first neural network for predicting sets that properly respects the unordered nature of sets, and – in our opinion – it is the most significant contribution of this thesis.

We conclude the thesis by discussing potential future research directions in Chapter 7. Sets are a natural way of modeling object-based representations, which can be beneficial in a much wider variety of tasks and models than sets are currently used in.

1.1 List of contributions

Concretely, this thesis contributes the following:

Chapter 2

- We provide an accessible introduction to modeling sets with neural networks. We cover how sets are represented (Section 2.2), set encoders (Section 2.3), set decoders (Section 2.4), and how sets are used in other areas of machine learning (Section 2.5).

Chapter 3 These contributions have been published as [118] in the International Conference on Learning Representations (ICLR) 2018 and have been presented at the Visual Question Answering Challenge workshop, hosted at the Conference on Computer Vision and Pattern Recognition (CVPR) 2018.

- We review related work on counting objects in images with neural networks (Section 3.2).
- We identify a problem in existing VQA models and explain why they struggle with counting questions as a result (Section 3.3).
- We propose a model for counting sets of objects that avoids this problem (Section 3.4).
- We evaluate our model on a toy dataset that we developed for testing counting ability in an isolated setting (Subsection 3.5.1), and on the full VQA v2 dataset (Subsection 3.5.2).
- We open-source the code to reproduce all our experiments at: `https://github.com/Cyanogenoid/vqa-counting` and the first publically-available code to reproduce a strong baseline by Kazemi et al. [54] at: `https://github.com/Cyanogenoid/pytorch-vqa`.

Chapter 4 These contributions have been published as [117] in the International Conference on Learning Representations (ICLR) 2019.

- We propose a model for learning permutations based on learning pairwise comparisons (Section 4.2).
- We review related work on learning permutations (Section 4.3).
- We evaluate our model on sorting numbers (Subsection 4.4.1), assembling image mosaics explicitly (Subsection 4.4.2) or implicitly (Subsection 4.4.3), and VQA v2 (Subsection 4.4.4).
- We open-source the code to reproduce all experiments at: `https://github.com/Cyanogenoid/perm-optim`,

which also includes the first reproduction of the baseline results by Mena et al. [71].

Chapter 5 These contributions have been presented as [116] at the Sets & Partitions workshop, hosted at the Neural Information Processing Systems (NeurIPS) 2019 conference, and have been submitted to the International Conference on Learning Representations (ICLR) 2020.

- We identify a responsibility problem with existing set prediction methods, which results in discontinuities (Section 5.3).
- We propose a set encoder based on sorting features numerically and an associated set decoder for auto-encoding (Section 5.4).
- We review related work on using sorting in neural networks (Section 5.5).
- We evaluate our auto-encoder on a rotating polygon toy dataset (Subsection 5.6.1) and a set version of MNIST (Subsection 5.6.2).
- We evaluate our encoder on MNIST set classification (Subsection 5.6.3), CLEVR (Subsection 5.6.4), and graph classification (Subsection 5.6.5).
- We open-source the code to reproduce all experiments at: `https://github.com/Cyanogenoid/fspool`, which also includes the first reproduction of the baseline graph classification results by Xu et al. [105].

Chapter 6 These contributions have been published as [115] in Advances in Neural Information Processing Systems 32 (NeurIPS) 2019 and have been presented at the Sets & Partitions workshop, hosted at the Neural Information Processing Systems (NeurIPS) 2019 conference.

- We propose a model for general set prediction (Section 6.3) that avoids the responsibility problem.
- We review related work on predicting sets with neural networks (Section 6.4).
- We evaluate our model on auto-encoding MNIST sets (Subsection 6.5.1), predicting the set of bounding boxes in an image (Subsection 6.5.2), and predicting the set of object attributes in an image (Subsection 6.5.3).
- We open-source the code to reproduce all experiments at: `https://github.com/Cyanogenoid/dspn`, which includes an implementation of the models by Achlioptas et al. [1].

Chapter 7

- We discuss future research directions and open problems. We discuss ideas in the area of set encoders (Section 7.1), set decoders (Section 7.2), and using latent sets in neural networks (Section 7.3).

Chapter 2

Basics of set neural networks

In this chapter, we will introduce the basics of working with sets using neural networks. This provides the necessary background from the set literature to understand this thesis, especially Chapter 4, Chapter 5, and Chapter 6. This chapter is not an exhaustive review of the literature, but an accessible introduction into the foundational work on modeling sets with neural networks. We leave the more detailed discussions of related work in the appropriate chapters, in particular Section 3.2, Section 4.3, Section 5.5, Section 6.4.

2.1 Overview

First, let us be clear with what we mean when we talk about sets. In essence, sets are *unordered* collections of *entities* or *elements*. These entities can be objects, people, atoms, symbols, and so on. Since we are working in machine learning, it is useful to describe each entity in the set with a feature vector. For example, we can have the set of points in a point cloud and store their 3d position (xyz coordinates) as the set $\{[x_1, y_1, z_1]^\top, [x_2, y_2, z_2]^\top, \ldots\}$ with each $[x_i, y_i, z_i]^\top$ as the feature vector for a point in the set. Because the order of the points does not matter – swapping the order of any two points does not change the shape captured by the point cloud – this is indeed a set. Since the entities in the set typically correspond to real-world things, it does not make much practical sense to talk about infinitely many entities. So, the sets we work with will always have a finite number of elements in them. However, there is one major difference compared to sets in mathematics: duplicates are typically allowed, so when we talk about sets, we usually mean multisets (also known as bags).

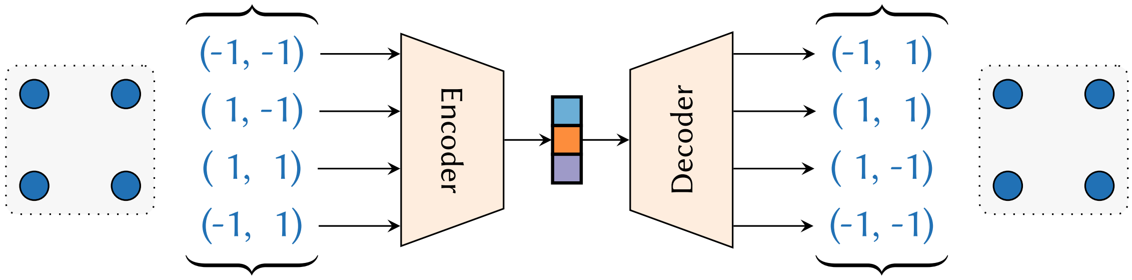


Figure 2.1: A neural network for auto-encoding a set of four 2d points. The set encoder encodes the input set into a feature vector, which the set decoder decodes to predict a set. Notice how the output points are not necessarily in the same order as the input points.

Let us see how this fits into existing machine learning setups. In traditional supervised learning, we have a dataset that consists of many input-output pairs. The subject of interest in this thesis is when the input or the output of the model is a set of feature vectors. These sets usually vary in size across the dataset, so any model for sets has to be able to work with varying set sizes.

There are two main types of operations that we care about, which we visualise in Figure 2.1.

1. When the input is a set, we want to *encode* the input set into a feature vector (Section 2.3).

2. When the output is a set, we want to *decode* a feature vector into the output set (Section 2.4).

By relating sets of feature vectors with a single feature vector in this way, we can combine these operations on sets with other, well-established methods that operate on feature vectors. If we can also make encoding and decoding differentiable, they can be included in deep neural networks. This gives us a fully differentiable model for sets that can be trained with conventional methods for neural networks like stochastic gradient descent (SGD) and its variants.

This differentiability is one of the important foundations that much work in Deep Learning is built upon. We will take care to build our models from differentiable primitives so that they easily fit into the usual gradient-based neural network training framework. For more background on deep neural networks, we refer you to part two of the Deep Learning book [37].

2.2 Representation in memory

To understand how set encoding and decoding works, we first have to understand how the set itself is represented.

Even though sets are orderless, we still have to store them in memory, which forces them to be ordered in some way. Conveniently, the order of the set elements does not matter, so we can store the elements in the set in *any* arbitrary order. In essence, we treat the set of feature vectors as if it were just a list of feature vectors, which means that we can simply store it as a matrix. A set of size n wherein each feature vector has dimensionality d can therefore be stored as an $\mathbb{R}^{d \times n}$ or $\mathbb{R}^{n \times d}$ matrix. We will use one or the other in the upcoming chapters depending on what makes the narrative the clearest, but we will always clarify which one we are using in a chapter. In this chapter, we store them as $\mathbb{R}^{d \times n}$ matrices: each column corresponds to one set element.

Example 2.1

Let us look at an example. The set:

$$\left\{ \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} \right\} \tag{2.1}$$

has two elements of dimensionality three. It can be stored as one of these two (equivalent) matrices:

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \text{ or } \begin{bmatrix} 4 & 1 \\ 5 & 2 \\ 6 & 3 \end{bmatrix}. \tag{2.2}$$

Storing the set as a matrix is memory efficient and easy to work with through the usual matrix operations like matrix multiplication. However, this comes with the trade-off that there are now multiple representations of the same set, one for each possible permutation of the set elements. Since these different representations are an artefact of the way the set is stored in memory, a key aspect when working with sets is to not rely on this arbitrary ordering of the elements.

There is one additional consideration for minibatch-wise training with sets, which is needed with stochastic gradient descent. Since the set size typically varies across the dataset, we end up with sets of different sizes in a minibatch. To make computation on a minibatch efficient, the typical approach is to pad all sets within a batch to a fixed size with elements of all zeros. This fixed size is usually the size of the largest set in the minibatch or the size of the largest set in the dataset. We then need to keep track of which elements are padding elements to not affect the result. For example, if we want to compute a mean over the set elements, we cannot divide the sum by the number of columns in the matrix (which includes padding elements not part of the set), but we have to divide by the actual set size.

2.3 Set encoders

Now that we know how sets are stored, let us take a look at how such sets can be encoded into a feature vector. We want to build a neural network that can take a set of feature vectors as input and produce a feature vector representation of that set as output. Set encoders can for example be used for classifying what type of object the shape of a 3d point cloud forms, or answering natural language questions about a set of objects (visual question answering).

At this point, we have a matrix representation of the set and could just use a traditional neural network approach – such as a multilayer perceptron (MLP) – on it. The $d \times n$ matrix can be flattened into a dn-dimensional vector and we could feed this into a normal fully-connected neural network. This is essentially what Murphy et al. [76] do, which we discuss in more detail in Section 4.3. However, this has a major problem: different representations of the same set can give a different output. There is no guarantee that $f([1, 2, 3]) = f([2, 3, 1]) = f([3, 2, 1]) = \ldots$, even though they all represent the same set $\{1, 2, 3\}$.

While a neural net as a universal approximator could learn to map all of these to the same output, this quickly becomes infeasible for non-trivial set sizes. With n elements in a set and thus $n!$ different representations of the same set, an MLP quickly runs into its modeling capabilities. This is why Murphy et al. [76] recommend sampling many permutations at test time and averaging the results to counteract this reliance on the arbitrary permutation.

We already know that order should not matter for sets. So, let us be smarter about this: rather than letting the MLP learn freely, we can structure the neural network so that it is *impossible* for it to rely on the arbitrary order. This lets the neural network focus on extracting the pertinent information in the set, without having to learn to ignore the order at the same time. Building the necessary structures into a neural network to properly work with sets is the key approach in this thesis.

2.3.1 Properties

To enforce a neural network to not rely on the ordering of the set elements, there are two properties that are important to know about. In the context of deep neural networks, these were first defined by Zaheer et al. [112].

Permutation-invariance The property of permutation-invariance says that the output of a function *does not change* when its input is permuted. If a permutation-invariant function is applied to a set, then we can guarantee that regardless of which arbitrary order we stored the elements in, we get the same output. This ensures that the arbitrarily-chosen order of the set can never affect the result. A permutation-invariant function can be thought of as serving the same purpose as pooling in convolutional neural networks (CNNs): it reduces multiple feature vectors to a single feature vector.

Definition 1 A function $f : \mathbb{R}^{d \times n} \to \mathbb{R}^c$ is permutation-invariant iff it satisfies:

$$f(\boldsymbol{X}) = f(\boldsymbol{XP}) \tag{2.3}$$

for all permutation matrices $\boldsymbol{P}$.

Example 2.2

Take the set from the previous example:

$$\left\{ \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} \right\} \tag{2.4}$$

Summing each feature over the set is permutation-invariant. Regardless of which order the set is stored in, we have:

$$\begin{bmatrix} 1+4 \\ 2+5 \\ 3+6 \end{bmatrix} = \begin{bmatrix} 4+1 \\ 5+2 \\ 6+3 \end{bmatrix} = \begin{bmatrix} 5 \\ 7 \\ 9 \end{bmatrix}. \tag{2.5}$$

We can do this because summing (as well as other operations like calculating the mean, maximum, or the product) is commutative and associative.

Another permutation-invariant function is numerical sorting. We will elaborate on how to make use of this in Chapter 5.

When building set encoders, this property of permutation-invariance is what is ultimately needed. However, in the example, there are currently no weights to be learned and the possible operations are quite simplistic. Using a single sum as set encoder can only express rather basic information about the set. This is where the second property comes in, which adds in learnable weights.

Permutation-equivariance Permutation-equivariance says that the output of a function *changes in a predictable way* when its input is permuted. The difference to the permutation-invariance property is that the output is still a set, not a single feature vector. Changing the order of the input should only change the order of the output, not any of its values. Again, this property is to ensure that the arbitrary order of the set elements does not affect the result.

Definition 2 A function $g : \mathbb{R}^{d \times n} \to \mathbb{R}^{c \times n}$ is permutation-equivariant iff it satisfies:

$$g(\boldsymbol{XP}) = g(\boldsymbol{X})\boldsymbol{P} \tag{2.6}$$

for all permutation matrices $\boldsymbol{P}$.

Example 2.3

The most important permutation-equivariant function is applying a function on each set element, i.e. on each feature vector. Since this function is applied on each set element individually, it does not take the order into account, so it is permutation-equivariant. Let us see what happens on our running example with the function $g(\boldsymbol{x}) = \sum_i x_i$.

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \xrightarrow{g} \begin{bmatrix} g\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} & g\begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} \end{bmatrix} = \begin{bmatrix} 6 & 15 \end{bmatrix} \tag{2.7}$$

$$\begin{bmatrix} 4 & 1 \\ 5 & 2 \\ 6 & 3 \end{bmatrix} \xrightarrow{g} \begin{bmatrix} g\begin{pmatrix} 4 \\ 5 \\ 6 \end{pmatrix} & g\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \end{bmatrix} = \begin{bmatrix} 15 & 6 \end{bmatrix} \tag{2.8}$$

Swapping the first and second element of the input also swaps the first and second element of the output. A widely-used choice for g is a neural network, which lets the model *learn* a transformation of the set elements. Another popular choice is concatenation of the same feature vector to every element, which is common in soft attention models [9, 74].

With the two main properties in place, we can start combining functions with these properties to give us more complex functions.

Corollary 1 *The composition of two permutation-equivariant functions g and h is permutation-equivariant.*

Proof.

$$h\big(g(\boldsymbol{X}\boldsymbol{P})\big) = h\big(g(\boldsymbol{X})\boldsymbol{P}\big) = h\big(g(\boldsymbol{X})\big)\boldsymbol{P} \tag{2.9}$$

□

Corollary 2 *The composition of a permutation-equivariant function g with a permutation-invariant function f is permutation-invariant.*

Proof.

$$f\big(g(\boldsymbol{X}\boldsymbol{P})\big) = f\big(g(\boldsymbol{X})\boldsymbol{P}\big) = f\big(g(\boldsymbol{X})\big) \tag{2.10}$$

□

This means that we can compose equivariant and invariant functions to build learnable and more powerful set encoders while still being permutation-invariant on the whole.

2.3.2 Specific set encoders

Let us look at some specific examples of set encoder architectures used in the literature.

Deep Sets One of the simplest set encoders is the Deep Sets model by Zaheer et al. [112]. At its core, it is defined as:

$$h(\boldsymbol{X}) = p\left(\sum_{\boldsymbol{x}\in\boldsymbol{X}} g(\boldsymbol{x})\right) \tag{2.11}$$

We use $\boldsymbol{x} \in \boldsymbol{X}$ to refer to the columns in the matrix $\boldsymbol{X}$, which correspond to the different set elements. The set encoder h applies a neural network g on every set element $\boldsymbol{x}$ (permutation-equivariant), then sums over the transformed set (permutation-invariant). The feature vector result of this is fed into another neural network p to perform the task at hand, such as classification. While this seems quite basic, it turns out that it is a universal approximator for permutation-invariant functions [112, 103].

Relation Networks Of course, universal approximation does not mean that the set encoder problem is solved, since it says nothing about learnability and generalisation. Just like how there are improvements to neural networks with a single hidden layer (which are already capable of universal approximation [27]), there are improvements to the

Deep Sets model as well. One such improvement are Relation Networks [89]:

$$h(\boldsymbol{X}) = p\left(\sum_{\boldsymbol{x},\boldsymbol{y}\in\boldsymbol{X}} g(\boldsymbol{x},\boldsymbol{y})\right) \tag{2.12}$$

The difference here is that the sum is not over the transformed set elements, but over the transformed *pairs* of set elements. Expanding the set into the set of all pairs first is permutation-equivariant. To model relations between different elements, the Deep Sets model can only let information from the set elements interact through the sum. With the Relation Network, g can model relations between pairs of elements much more easily since it receives the pairs as inputs. This has been successfully used in tasks where relations between set elements are important [113, 80, 46]. This comes at the cost of $\Theta(n^2)$ time complexity, compared to the $\Theta(n)$ complexity of Deep Sets.

PointNet PointNet [84] is a model designed for 3d point clouds. There are two main differences to what we have seen earlier: the permutation-invariant function is not a sum, but a maximum, and there are multiple stages of encoding.

$$h_1(\boldsymbol{X}) = p_1\big(\max_{\boldsymbol{x}\in\boldsymbol{X}} g_1(\boldsymbol{x})\big) \tag{2.13}$$

$$h_2(\boldsymbol{X}) = p_2\big(\max_{\boldsymbol{x}\in\boldsymbol{X}} g_2(\text{Concat}[\boldsymbol{x}, h_1(\boldsymbol{X})])\big) \tag{2.14}$$

Here, the set is first encoded to a feature vector using an elementwise maximum in h_1. This feature vector is concatenated to each element of the set again, and this new set is once again encoded. This two-stage process lets global information about the set (obtained with h_1) be shared with the individual elements, which can then affect the representation in h_2. Concatenation of a fixed feature vector to every element of the set is permutation-equivariant, since it is simply a function applied to every element. p_1, p_2, g_1, and g_2 are once again MLPs with trainable weights.

Attention Soft attention attempts to model the human ability to focus on one thing out of many things. While not a full set encoder by itself, this is used as a building block in not only set encoders, but also in image and text encoders. The typical soft attention model [74, 9] takes in a set and additional contextual information as input, and "attends" to part of the set that is relevant to that context. This will come in use in Chapter 3, where the context is the question about the set.

$$h(\boldsymbol{X}, \boldsymbol{z}) = \sum_{\boldsymbol{x} \in \boldsymbol{X}} g(\boldsymbol{x}, \boldsymbol{z}) \boldsymbol{x} \tag{2.15}$$

$g(\boldsymbol{x}, \boldsymbol{z})$ is a neural network that outputs a scalar – which makes this a weighted sum – and is also often normalised with a softmax function so that it is a weighted average of the elements $\boldsymbol{x}$. The specific weightings depend on the context vector $\boldsymbol{z}$.

Elements where g has a high value have a greater influence on the result than where g is low. By giving certain elements higher weightings than others, the model can "focus" on the relevant elements of the set.

A modification of this idea called self-attention [100] has recently found great success in language modeling. In essence, *every* $\boldsymbol{x}$ is used as $\boldsymbol{z}$ to attend on the $\boldsymbol{X}$ set (this is where the "self" comes from), rather than using a single $\boldsymbol{z}$ from a separate model. This model is related to Relation Networks, since all pairs of $\boldsymbol{x}, \boldsymbol{y} \in \boldsymbol{X}$ are modeled by g again.

The example set encoders we have covered here demonstrate how compositions of permutation-invariant and permutation-equivariant functions are used in practice.

2.3.3 Pooling bottleneck

We mentioned earlier that the typical permutation-invariant primitives have no learnable parameters. Operations like sum and max are very simple, but have to compress a (potentially very large) set into a feature vector *in a single step*. This heavy compression can discard a lot of information that could be useful for the downstream task [76, 84].

This is what motivates us to develop *learned*, permutation-invariant approaches in Chapter 4 and Chapter 5. We show that by having a more sophisticated, learned model for this step, results are frequently improved. Due to the nature of composability of these set function properties, replacing the sum or max pooling in an existing set encoder with our approaches is straightforward.

2.4 Set decoders

Set decoders turn a feature vector into a set and are used when the desired output is set-structured. There are a variety of such tasks, ranging from object detection (predicting the set of objects in an image) to molecule generation (predicting the set of atoms and the set of bonds connecting them). This problem turns out to be much less studied than the set encoder case.

There are two new aspects to consider: how can we make this set prediction with a neural network, and how can we compute a loss between sets to use as the training objective?

2.4.1 Set losses

Regardless of how the set is predicted, we have to compute the loss between the predicted set and a ground-truth set in order to train the model. The problem here is that both predicted set as well as ground-truth set are in an arbitrary order. Naïvely computing a pairwise loss – such as a mean squared error – does not work, since there is no guarantee that the elements are in the same order in the matrix representation.

Example 2.4

Let us say that the prediction that a model has made is:

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \tag{2.16}$$

but the set label was stored as:

$$\begin{bmatrix} 4 & 1 \\ 5 & 2 \\ 6 & 3 \end{bmatrix}. \tag{2.17}$$

The model made the correct prediction of $\{[1,2,3]^\top, [4,5,6]^\top\}$, but a mean squared error would incorrectly think that the prediction was wrong. The order of the elements in the target set is arbitrary, so it is impossible for the model to guess the order. By pairing up the first column in the prediction with the second column in the label and vice versa, this issue is resolved.

Since the loss should not be affected by the order of either set, we need something that is permutation-invariant in both its arguments. One such loss is the *Chamfer loss*, which matches up every element of a predicted set $\hat{Y} = [\hat{\boldsymbol{y}}_1, \hat{\boldsymbol{y}}_2, \ldots]$ to the closest element in the target set $Y = [\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots]$ and vice versa. A normal pairwise loss – like a mean squared error – is used to measure this closeness.

$$L_{\text{cha}}(\hat{Y}, Y) = \sum_i \min_j \left\| \hat{\boldsymbol{y}}_i - \boldsymbol{y}_j \right\|^2 + \sum_j \min_i \left\| \hat{\boldsymbol{y}}_i - \boldsymbol{y}_j \right\|^2 \tag{2.18}$$

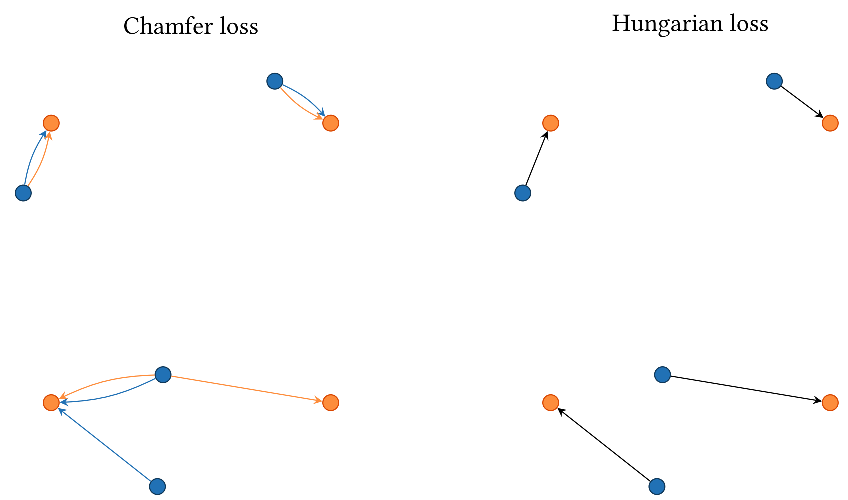


Figure 2.2: Example of the assignments of the Chamfer loss (left) and Hungarian loss (right). Orange points mark the target set, blue points mark the predicted set. The arrows show which direction the loss pulls a predicted point. Each point of one colour (e.g. orange) is matched up with the closest point of the other colour (e.g. blue). Blue arrows denote assignments from the first term in Equation 2.18, orange arrows denote assignments from the second term.

The nested sums and minimisations ensure the desired permutation invariance in both arguments. One issue is that one element in one set can be matched up with multiple elements in the other set. This means that there can be issues with different set sizes and duplicates: the loss between $[1, 1, 2]$ and $[1, 2, 2, 2]$ is 0, even though the multisets that are represented are clearly different. Conceptually, the Chamfer loss treats the inputs as proper sets, not the multisets that we usually work with.

A more sophisticated loss that does not have this problem involves the linear assignment problem with the pairwise losses as assignment costs:

$$L_{\text{hun}}(\hat{Y}, Y) = \min_{\pi \in \Pi} \sum_i \left\| \hat{\boldsymbol{y}}_i - \boldsymbol{y}_{\pi(i)} \right\|^2 \tag{2.19}$$

where Π is the space of all permutations. This minimisation can be solved exactly with the Hungarian algorithm and has the benefit of assigning each element in one set to exactly one element from the other set. This comes with the trade-off of $\Theta(n^3)$ time complexity and that it is not easily parallelisable. We visualise the difference between these two losses in Figure 2.2, which shows the Hungarian loss matching up elements one-to-one.

2.4.2 Predicting sets

Now that we know how to define the training objective for set prediction models, we need a model that is able to predict sets. The main approach to predicting sets in the literature is to use an MLP or RNN that simply predicts the matrix representation of the set. To predict a set of size n with vectors of size d, an MLP with nd number of outputs is used. Alternatively, the initial state of an RNN (such as an LSTM [44]) with d hidden units is initialised with the input feature vector and run for n steps to produce the $n \times d$ outputs. To handle sets with variable sizes, one option is to pad all sets in a dataset with zeros to a fixed size and concatenate an additional feature to each set element, indicating whether that element is a padding element. These two approaches are essentially the only general ways to predict sets in the literature without making domain-specific assumptions about the sets.

Aside: Object detection

For some specialised tasks like object detection, there are existing approaches like Faster R-CNN [86] that are not based on sets, but a combination of heuristics. These are specifically tuned to the image domain input and have certain drawbacks that a set-based approach avoids. A set-based approach has the advantage of being fully end-to-end differentiable, which potentially improves the quality of the predictions compared to a multi-stage object detector like Faster R-CNN. Traditional object detectors also require post-processing of the outputs outside of the training algorithm with techniques like non-maximum suppression, which is somewhat inelegant.

The goal of this thesis is to explore approaches which work generically for sets, without requiring input domain-specific changes. Therefore, we limit ourselves to feature vectors as input when predicting sets. Structured data like image feature maps can always be turned into a feature vector, but not necessarily vice versa.

These methods are somewhat unsatisfying, since they treat the set as if it were just a list, with the only difference to predicting a list being the use of the set loss. The MLP and RNN outputs are *ordered*, but they are used to predict sets, which are *unordered*. We indeed find in Section 5.3 that this can cause a *responsibility problem*. The mismatch between ordered MLP or RNN outputs and unordered sets results in discontinuity issues which hinders learning. The study in Chapter 5 is in part about this problem and our solution to it in the auto-encoding setting. Then, we build on this work to develop a set prediction method in Chapter 6 without the responsibility problem that is no longer limited to the auto-encoder setting. We show that our method can perform tasks like end-to-end object detection in a purely set-based way.

2.5 Other sets in machine learning

In this section, we discuss some related topics on sets in machine learning. This puts our work on sets into a broader context.

2.5.1 Pooling in CNNs

Global average pooling, which is commonly applied at the end of CNNs such as ResNets [42], average the feature vectors across all spatial positions in the CNN feature map. Essentially, global average pooling treats the different spatial positions as if they were a set. The same applies to global max pooling in order to summarise a feature map into a feature vector, which has been used by Perez et al. [83]. This can lead to surprising consequences where small patches of the input image can be shuffled around and processed independently (just like a set) without much loss of performance [17].

On a smaller scale, this applies to the non-global average and max pooling as well. For max pooling with a kernel size of 2×2, the four input feature vectors are treated as if they were a set.

Since these pooling methods are simply permutation-invariant functions, we can substitute them with any other permutation-invariant function. Our methods in Chapter 4 and Chapter 5 could therefore be used in CNNs. However, it is unclear whether they would make any significant difference with the much greater importance of convolutions over poolings in CNNs.

2.5.2 Per-element prediction

In some tasks, we have a set as input and want to predict something about each element of the set, rather than the set as a whole. In other words, these are set-to-set problems. While each element could be independently predicted, there is some prior belief that there is additional information to be gained from considering the other elements in the set at the same time. An example of this is detecting an outlier in a set of inputs [64]: determining an outlier requires finding commonalities between the non-outliers, so it is sensible to make a prediction about all elements in the set at once rather than one-by-one.

Unlike in set encoders, we want the output in these models to be equivariant rather than invariant. If $h([\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}]) = [\boldsymbol{a}', \boldsymbol{b}', \boldsymbol{c}']$, then changing the input to $h([\boldsymbol{b}, \boldsymbol{a}, \boldsymbol{c}])$ should give us $[\boldsymbol{b}', \boldsymbol{a}', \boldsymbol{c}']$. Because the order of the output is always the same as the input, there is no responsibility problem and no need for the assignment-based set losses. This makes it an easier problem than something like image-to-set. We will make use of this benefit in Chapter 5.

Taking a step back, supervised learning with a *dataset* is itself a set problem. The inputs in the dataset are fed into a model – the machine learning algorithm – which predicts an output for each element in the set (set-to-set). There is usually a loss over the dataset, which is permutation-invariant by averaging or summing across the individual losses. From this, computing things like ∂ loss$/\partial$ parameters is permutation-invariant too. Training algorithms that do not use the whole dataset at once (like SGD) are made invariant in expectation by randomly sampling from the dataset. Using a permutation of the dataset that is easier to learn with is known as curriculum learning [15].

2.5.3 Multi-labeling

Multi-labeling tasks can be seen as set prediction problems, where the goal is predict a set of labels. These labels come from a fixed, finite set of all possible labels. The main difference to our set prediction setup is that because there are only a finite number of possible labels, we can pick a fixed arbitrary ordering for these labels just like in classification or word embeddings. An n-hot vector encoding (a vector with a 0 in a dimension if the label is not present and a 1 when the label is present) therefore suffices for multi-label classification, though more set-oriented methods have also been studied [41, 12].

In contrast, the sets we are working with have vectors in $\mathbb{R}^d$ as elements. With infinitely many possible elements, there is no order we can enforce on the set elements a priori, so the strategy for multi-labeling does not work. Choosing one anyway results discontinuities in the labels, similar to the responsibility problem that we discuss in Section 5.3. This is why we need something like our model in Chapter 6 to predict these more complex sets with elements in $\mathbb{R}^d$.

2.5.4 Clustering

Lastly, clustering can be seen as a set prediction task. The goal in clustering is to find a partitioning of the input set where “similar” points are grouped into the same partition. So, the output is a *set of sets*, where the inner sets contain the elements of the input. This is usually done in an unsupervised setting, while our method in Chapter 6 works in the supervised learning setting. Our approach to set prediction could be used to train a *supervised* clustering algorithm like Finley et al. [32].

Chapter 3

Motivation: Counting in visual question answering

In this chapter, we will motivate our work on sets with the visual question answering task. We will develop a method for counting sets of object proposals that have associated bounding box information. These sets can be obtained from traditional object detectors.

When we worked on this problem, our focus was not on sets in general, but specifically the set of object proposals. We later realised the fundamental importance of properly using the given set representation, which motivated us to study sets more generally in later chapters.

These contributions have been published as [118] in the International Conference on Learning Representations (ICLR) 2018 and have been presented at the Visual Question Answering Challenge workshop, hosted at the Conference on Computer Vision and Pattern Recognition (CVPR) 2018.

3.1 Introduction

Consider the problem of counting how many cats there are in Figure 3.1. Solving this involves several rough steps: understanding what instances of that type can look like, finding them in the image, and adding them up. This is a common task in visual question answering (VQA) – answering questions about images – and is rated as among the tasks requiring the lowest human age to be able to answer [6]. However, current models for VQA on natural images struggle to answer *any* counting questions successfully outside of dataset biases [50].

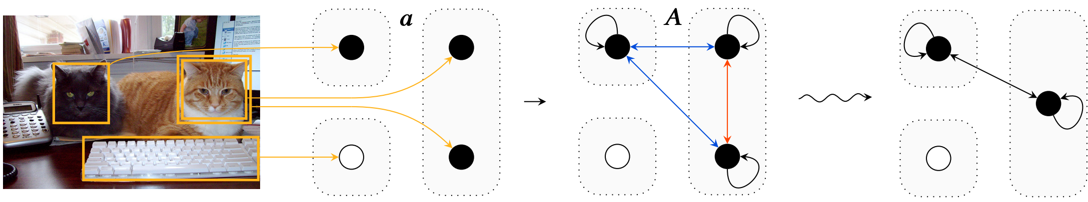


Figure 3.1: Simplified example about counting the number of cats. The light-coloured cat is detected twice and results in a duplicate proposal. This shows the conversion from the attention weights $\boldsymbol{a}$ to a graph representation $\boldsymbol{A}$ and the eventual goal of this module with exactly one proposal per true object. There are 4 proposals (vertices) capturing 3 underlying objects (groups in dotted lines). There are 3 relevant proposals (black with weight 1) and 1 irrelevant proposal (white with weight 0). Red edges mark *intra*-object edges between duplicate proposals and blue edges mark the main *inter*-object duplicate edges. In graph form, the object groups, colouring of edges, and shading of vertices serve illustration purposes only; the model does not have these access to these directly.

One reason for this is the presence of a fundamental problem with counting in the widely-used soft attention mechanisms (Section 3.3) that we identify. Another reason is that unlike standard counting tasks, there is no ground truth labelling of where the objects to count are. Coupled with the fact that models need to be able to count a large variety of objects and that, ideally, performance on non-counting questions should not be compromised, the task of counting in VQA seems very challenging.

To make this task easier, we can use object proposals – a set of pairs of a bounding box and object features – from object detection networks as input instead of learning from pixels directly. In any moderately complex scene, this runs into the issue of double-counting overlapping object proposals. This is a problem present in many natural images, which leads to inaccurate counting in real-world scenarios.

Our main contribution is a differentiable neural network module that tackles this problem and consequently can learn to count (Section 3.4). Used alongside an attention mechanism, this module avoids a fundamental limitation of soft attention while producing strong counting features. As the input is a set, we build our model to be permutation-invariant to the order of the object proposals. We then provide experimental evidence of the effectiveness of this module (Section 3.5). On a toy dataset, we demonstrate that this module enables robust counting in a variety of scenarios. On the number category of the VQA v2 Open-Ended dataset [38], a relatively simple baseline model using the counting module outperforms all previous models – including large ensembles of state-of-the-art methods – without degrading performance on other categories.

3.2 Related work

Usually, greedy non-maximum suppression (NMS) is used to eliminate duplicate bounding boxes. The main problem with using it as part of a model is that its gradient is piecewise constant. Various differentiable variants such as by Azadi et al. [8], Hosang et al. [45], and Henderson et al. [43] exist. The main difference is that, since we are interested in counting, our module does not need to make discrete decisions about which bounding boxes to keep; it outputs counting features, not a smaller set of bounding boxes. Our module is also easily integrated into standard VQA models that utilise soft attention without any need for other network architecture changes and can be used without using true bounding boxes for supervision.

On the VQA v2 dataset [38] that we apply our method on, only few advances on counting questions have been made. The main improvement in accuracy is due to the use of object proposals in the visual processing pipeline, proposed by Anderson et al. [4]. Their object proposal network is trained with classes in singular and plural forms, for example "tree" versus "trees", which only allows primitive counting information to be present in the object features after region-of-interest pooling. Our approach differs in the way that instead of relying on counting features being present in the input, we create counting features using information present in the attention map over object proposals. This has the benefit of being able to count anything that the attention mechanism can discriminate instead of only objects that belong to the predetermined set of classes that had plural forms.

Using these object proposals, Trott et al. [99] train a sequential counting mechanism with a reinforcement learning loss on the counting question subsets of VQA v2 and Visual Genome. They achieve a small increase in accuracy and can obtain an interpretable set of objects that their model counted, but it is unclear whether their method can be integrated into traditional VQA models due to their loss not applying to non-counting questions. Since they evaluate on their own dataset, their results can not be easily compared to existing results in VQA.

Methods such as by Santoro et al. [89] and Perez et al. [83] can count on the synthetic CLEVR VQA dataset [52] successfully without bounding boxes and supervision of where the objects to count are. They also use more training data (~250,000 counting questions in the CLEVR training set versus ~50,000 counting questions in the VQA v2 training set), much simpler objects, and synthetic question structures.

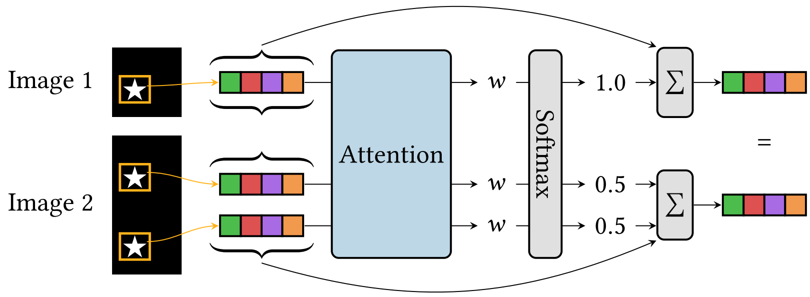


Figure 3.2: Example demonstrating the problem with using soft attention for counting. The resulting two feature vectors are the same for the two images, even though the images show a different number of stars.

More traditional approaches based on Lempitsky et al. [66] learn to produce a target density map, from which a count is computed by integrating over it. In this setting, Cohen et al. [25] make use of overlaps of convolutional receptive fields to improve counting performance. Chattopadhyay et al. [22] use an approach that divides the image into smaller non-overlapping chunks, each of which is counted individually and combined together at the end. In both of these contexts, the convolutional receptive fields or chunks can be seen as sets of bounding boxes with a fixed structure in their positioning. Note that while Chattopadhyay et al. [22] evaluate their models on a small subset of counting questions in VQA, major differences in training setup make their results not comparable to our work.

3.3 Problems with soft attention

The main message in this section is that using the feature vectors obtained after the attention mechanism is not enough to be able to count; the attention maps themselves should be used, which is what we do in our counting module.

Models in VQA have consistently benefited from the use of soft attention [74, 10] on the image, commonly implemented with a shallow convolutional network. It learns to output a weight for the feature vector at each spatial position in the feature map, which is first normalised and then used for performing a weighted sum over the spatial positions to produce a single feature vector. However, soft spatial attention severely limits the ability for a model to count.

Consider the task of counting the number of stars for two images (Figure 3.2): an image showing a single star on a clean background and an image that consists of two side-by-side copies of the first image. What we will describe applies to both spatial feature maps and sets of object proposals as input, but we focus on the latter case for simplicity. With an object detection network, we detect one star in the first image and two stars in the second image, producing the same feature vector for all three detections. The attention mechanism then assigns all three instances of the same star the same weight.

The usual normalisation used for the attention weights is the softmax function, which normalises the weights to sum to 1. Herein lies the problem: the star in the first image receives a normalised weight of 1, but the two stars in the second image now each receive a weight of 0.5. After the weighted sum, we are effectively averaging the two stars in the second image back to a single star. As a consequence, the feature vector obtained after the weighted sum is exactly the same between the two images and we have lost all information about a possible count from the attention map. Any method that normalises the weights to sum to 1 suffers from this issue.

Multiple glimpses [61] – sets of attention weights that the attention mechanism outputs – or several steps of attention [108, 70] do not circumvent this problem. Each glimpse or step can not separate out an object each, since the attention weight given to one feature vector does not depend on the other feature vectors to be attended over. Hard attention [9, 74] and structured attention [57] may be possible solutions to this, though no significant improvement in counting ability has been found for the latter so far [120]. Ren et al. [85] circumvent the problem by limiting attention to only work within one bounding box at a time, remotely similar to our approach of using object proposal features.

Without normalisation of weights to sum to one, the scale of the output features depends on the number of objects detected. In an image with 10 stars, the output feature vector is scaled up by 10. Since deep neural networks are typically very scale-sensitive – the scale of weight initialisations and activations is generally considered quite important [73] – and the classifier would have to learn that joint scaling of all features is somehow related to count, this approach is not reasonable for counting objects. This is shown in Teney et al. [98] where they provide evidence that sigmoid normalisation not only degrades accuracy on non-number questions slightly, but also does not help with counting.

3.4 Counting module

In this section, we describe a differentiable mechanism for counting from attention weights, while also dealing with the problem of overlapping object proposals to reduce double-counting of objects. This involves some nontrivial details to produce counts that are as accurate as possible. The main idea is illustrated in Figure 3.1 with the two main steps shown in Figure 3.3 and Figure 3.4. The use of this module allows a model to count while still being able to exploit the benefits of soft attention.

Our key idea for dealing with overlapping object proposals is to turn these object proposals into a graph that is based on how they overlap. We then remove and scale edges in a specific way such that an estimate of the number of underlying objects is recovered.

Our general strategy is to primarily design the module for the unrealistic extreme cases of perfect attention maps and bounding boxes that are either fully overlapping or fully distinct. By introducing some parameters and only using differentiable operations, we give the ability for the module to interpolate between the correct behaviours for these extreme cases to handle the more realistic cases. These parameters are responsible for handling variations in attention weights and partial bounding box overlaps in a manner suitable for a given dataset.

3.4.1 Piecewise linear activation

To introduce these parameters, we use several piecewise linear functions $f_1, \dots, f_8$ as activation functions, approximating arbitrary functions with domain and range [0, 1]. We show how these functions look after training in Figure 3.7 (page 35). The shapes of these functions are learned to handle the specific nonlinear interactions necessary for dealing with overlapping proposals. Through their parametrisation we enforce that $f_k(0) = 0$, $f_k(1) = 1$, and that they are monotonically increasing. The first two properties are required so that the extreme cases that we explicitly handle are left unchanged. In those cases, f_k is only applied to values of 0 or 1, so the activation functions can be safely ignored for understanding how the module handles them. By enforcing monotonicity, we can make sure that, for example, an increased value in an attention map should never result in the prediction of the count to decrease.

Intuitively, the interval $[0, 1]$ is split into d equal size intervals. Each contains a line segment that is connected to the neighbouring line segments at the boundaries of the intervals. These line segments form the shape of the activation function.

For each function f_k, there are d weights $w_{k1}, \dots, w_{kd}$, where the weight w_{ki} is the gradient for the interval $[\frac{i-1}{d}, \frac{i}{d})$. We arbitrarily fix d to be 16 in this chapter, observing no significant difference when changing it to 8 and 32 in preliminary experiments. All w_{ki} are enforced to be non-negative by always using the absolute value of them, which yields the monotonicity property. Dividing the weights by $\sum_m^d |w_{km}|$ yields the property that $f(1) = 1$. The function can be written as

$$f_k(x) = \sum_{i=1}^{d} \max(0, 1 - |dx - i|) \frac{\sum_{j=1}^{i} |w_{kj}|}{\sum_{m=1}^{d} |w_{km}|} \tag{3.1}$$

In essence, the max term selects the two nearest boundary values of an interval, which are normalised cumulative sums over the w_k weights, and linearly interpolates between the two. This approach is similar to the subgradient approach by Jaderberg et al. [51] to make sampling from indices differentiable. All w_{ki} are initialised to 1, which makes the functions linear on initialisation. When applying $f_k(\boldsymbol{x})$ to a vector-valued input $\boldsymbol{x}$, it is assumed to be applied elementwise. By caching the normalised cumulative sum $\sum_j^i |w_{kj}|/\sum_m^d |w_{km}|$, this function has linear time complexity with respect to d and is efficiently implementable on GPUs.

Extensions to this are possible through Deep Lattice Networks [111], which preserve monotonicity across several nonlinear neural network layers. They would allow $\boldsymbol{A}$ and $\boldsymbol{D}$ to be combined in more sophisticated ways beyond an elementwise product, possibly improving counting performance as long as the property of the range lying within $[0, 1]$ is still enforced in some way.

3.4.2 Input

Given a set of features from object proposals, an attention mechanism produces a weight for each proposal based on the question. The counting module takes as input the n largest attention weights $\boldsymbol{a} = [a_1, \ldots, a_n]^\mathsf{T}$ and their corresponding bounding boxes $\boldsymbol{b} = [b_1, \ldots, b_n]^\mathsf{T}$. We assume that the weights lie in the interval $[0, 1]$, which can easily be achieved by applying a logistic function on the attention map.

In the extreme cases that we explicitly handle, we assume that the attention mechanism assigns a value of 1 to a_i whenever the ith proposal contains a relevant object and a value of 0 whenever it does not. This is in line with what usual soft attention mechanisms learn, as they produce higher weights for relevant inputs. We also assume that either two object proposals fully overlap (in which case they must be showing the same object and thus receive the same attention weight) or that they are fully distinct (in which case they show different objects). Keep in mind that while we make these assumptions to make reasoning about the behaviour easier, the learned parameters in the activation functions are intended to handle the more realistic scenarios when the assumptions do not apply.

Instead of partially overlapping proposals, the problem now becomes the handling of *exact duplicate* proposals of underlying objects in a differentiable manner.

3.4.3 Deduplication

We start by changing the vector of attention weights $\boldsymbol{a}$ into a graph representation in which bounding boxes can be utilised more easily. Hence, we compute the outer product of the attention weights to obtain an attention matrix.

$$\boldsymbol{A} = \boldsymbol{a}\boldsymbol{a}^{\top} \tag{3.2}$$

$\boldsymbol{A} \in \mathbb{R}^{n \times n}$ can be interpreted as an adjacency matrix for a weighted directed graph. In this graph, the ith vertex represents the object proposal associated with a_i and the edge between any pair of vertices (i, j) has weight $a_i a_j$. In the extreme case where a_i is virtually 0 or 1, products are equivalent to logical AND operators. It follows that the subgraph containing only the vertices satisfying $a_i = 1$ is a complete digraph with self-loops.

In this representation, our objective is to eliminate edges in such a way that, conceptually, the underlying true objects – instead of proposals thereof – are the vertices of that complete subgraph. In order to then turn that graph into a count, recall that the number of edges $|E|$ in a complete digraph with self-loops relates to the number of vertices $|V|$ through $|E| = |V|^2$. $|E|$ can be computed by summing over the entries in an adjacency matrix and $|V|$ is then the count. Notice that when $|E|$ is set to the sum over $\boldsymbol{A}$, $|E| = \sum_{ij} \boldsymbol{a}_i \boldsymbol{a}_j = (\sum_i \boldsymbol{a}_i)^2$, which implies $|V| = \sum_i \boldsymbol{a}_i$. This convenient property implies that when all proposals are fully distinct, the module can output the same as simply summing over the original attention weights by default.

There are two types of duplicate edges to eliminate to achieve our objective: *intra*-object edges and *inter*-object edges.

Intra-object edges

First, we eliminate intra-object edges between duplicate proposals of a single underlying object.

To compare two bounding boxes, we use the usual intersection-over-union (IoU) metric. We define the distance matrix $\boldsymbol{D} \in \mathbb{R}^{n \times n}$ to be

$$D_{ij} = 1 - \mathrm{IoU}(b_i, b_j) \tag{3.3}$$

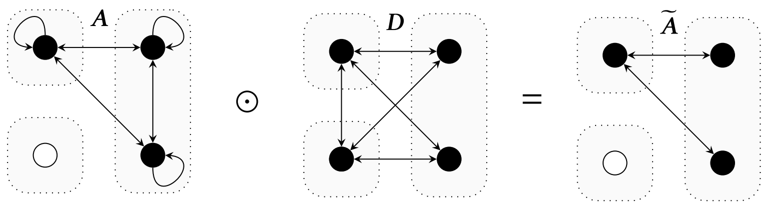


Figure 3.3: Removal of intra-object edges by masking the edges of the attention matrix $\boldsymbol{A}$ with the distance matrix $\boldsymbol{D}$. The black vertices now form a graph without self-loops. The self-loops need to be added back in later.

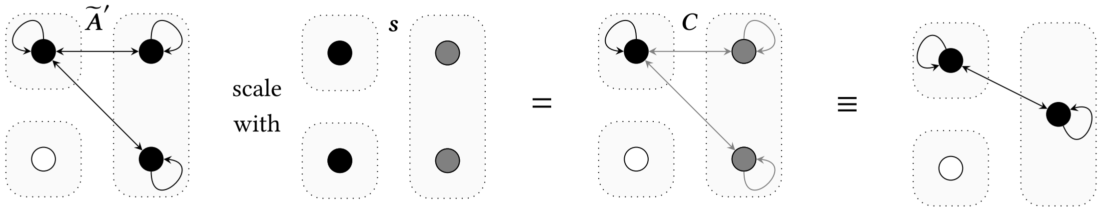


Figure 3.4: Removal of duplicate inter-object edges by computing a scaling factor for each vertex and scaling $\widetilde{\boldsymbol{A}}'$ accordingly. $\widetilde{\boldsymbol{A}}'$ is $\widetilde{\boldsymbol{A}}$ with self-loops already added back in. The scaling factor for one vertex is computed by counting how many vertices have outgoing edges to the same set of vertices; all edges of the two proposals on the right are scaled by 0.5. This can be seen as averaging proposals within each object and is equivalent to removing duplicate proposals altogether under a sum.

$\boldsymbol{D}$ can also be interpreted as an adjacency matrix. It represents a graph that has edges everywhere except when the two bounding boxes that an edge connects would overlap.

Intra-object edges are removed by elementwise multiplying ($\odot$) the distance matrix with the attention matrix (Figure 3.3).

$$\widetilde{\boldsymbol{A}} = f_1(\boldsymbol{A}) \odot f_2(\boldsymbol{D}) \tag{3.4}$$

$\widetilde{\boldsymbol{A}}$ no longer has self-loops, so we need to add them back in at a later point to still satisfy $|E| = |V|^2$. Notice that we start making use of the activation functions mentioned earlier to handle intermediate values in the interval $(0, 1)$ for both $\boldsymbol{A}$ and $\boldsymbol{D}$. They regulate the influence of attention weights that are not close to 0 or 1 and the influence of partial overlaps.

Inter-object edges

Second, we eliminate inter-object edges between duplicate proposals of different underlying objects.

The main idea (depicted in Figure 3.4) is to count the number of proposals associated to each individual object, then scale down the weight of their associated edges by that number. If there are two proposals of a single object, the edges involving those proposals should be scaled by 0.5.

In essence, this averages over the proposals within each underlying object because we only use the sum over the edge weights to compute the count at the end. Conceptually, this reduces multiple proposals of an object down to one as desired. Since we do not know how many proposals belong to an object, we have to estimate this. We do this by using the fact that proposals of the same object are similar.

Keep in mind that $\widetilde{\boldsymbol{A}}$ has no self-loops nor edges between proposals of the same object. As a consequence, two nonzero rows in $\widetilde{\boldsymbol{A}}$ are the same if and only if the proposals are the same. If the two rows differ in at least one entry, then one proposal overlaps a proposal that the other proposal does not overlap, so they must be different proposals. This means for comparing rows, we need a similarity function that satisfies the criteria of taking the value 1 when they differ in no places and 0 if they differ in at least one place. We define a differentiable similarity between proposals i and j as

$$\mathrm{Sim}_{ij} = f_3(1 - |a_i - a_j|) \prod_k f_3(1 - |X_{ik} - X_{jk}|) \tag{3.5}$$

where $\boldsymbol{X} = f_4(\boldsymbol{A}) \odot f_5(\boldsymbol{D})$ is the same as $\widetilde{\boldsymbol{A}}$ except with different activation functions. The $\prod$ term compares the rows of proposals i and j. Using this term instead of $f_4(1 - D_{ij})$ was more robust to inaccurate bounding boxes in initial experiments.

Note that the $f_3(1 - |a_i - a_j|)$ term handles the edge case when there is only one proposal to count. Since $\boldsymbol{X}$ does not have self-loops, $\boldsymbol{X}$ contains only zeros in that case, which causes the row corresponding to $a_i = 1$ to be incorrectly similar to the rows where $a_{j \neq i} = 0$. By comparing the attention weights through that term as well, this issue is avoided.

Now that we can check how similar two proposals are, we count the number of times any row is the same as any other row and compute a scaling factor s_i for each vertex i.

$$s_i = 1 / \sum_j \mathrm{Sim}_{ij} \tag{3.6}$$

The time complexity of computing $\boldsymbol{s} = [s_1, \ldots, s_n]^\top$ is $\Theta(n^3)$ as there are n^2 pairs of rows and $\Theta(n)$ operations to compute the similarity of any pair of rows.

Since these scaling factors apply to each vertex, we have to expand $\boldsymbol{s}$ into a matrix using the outer product in order to scale both incoming and outgoing edges of each vertex. We can also add self-loops back in, which need to be scaled by $\boldsymbol{s}$ as well. Then, the count matrix $\boldsymbol{C}$ is

$$\boldsymbol{C} = \widetilde{\boldsymbol{A}} \odot \boldsymbol{s}\boldsymbol{s}^{\mathsf{T}} + \operatorname{diag}(\boldsymbol{s} \odot f_1(\boldsymbol{a} \odot \boldsymbol{a})) \tag{3.7}$$

where $\operatorname{diag}(\cdot)$ expands a vector into a diagonal matrix with the vector on the diagonal.

The scaling of self-loops involves a non-obvious detail. Recall that the diagonal that was removed when going from $\boldsymbol{A}$ to $\widetilde{\boldsymbol{A}}$ contains the entries $f_1(\boldsymbol{a} \odot \boldsymbol{a})$. Notice however that we are scaling this diagonal by $\boldsymbol{s}$ and not $\boldsymbol{s} \odot \boldsymbol{s}$. This is because the number of inter-object edges scales quadratically with respect to the number of proposals per object, but the number of self-loops only scales linearly.

3.4.4 Output

Under a sum, $\boldsymbol{C}$ is now equivalent to a complete graph with self-loops that involves all relevant objects instead of relevant proposals as originally desired.

To turn $\boldsymbol{C}$ into a count c, we set $|E| = \sum_{i,j} C_{ij}$ as mentioned and

$$c = |V| = \sqrt{|E|} \tag{3.8}$$

We verified experimentally that when our extreme case assumptions hold, c is always an integer and equal to the correct count, regardless of the number of duplicate object proposals.

To avoid issues with scale when the number of objects is large, we turn this single feature into several classes, one for each possible number. Since we only used the object proposals with the largest n weights, the predicted count c can be at most n. We define the output $\boldsymbol{o} = [o_0, o_1, \ldots, o_n]^{\mathsf{T}}$ to be

$$o_i = \max(0, 1 - |c - i|) \tag{3.9}$$

This results in a vector that is 1 at the index of the count and 0 everywhere else when c is exactly an integer, and a linear interpolation between the two corresponding one-hot vectors when the count falls in-between two integers.

3.4.5 Output confidence

Finally, we might consider a prediction made from values of $\boldsymbol{a}$ and $\boldsymbol{D}$ that are either close to 0 or close to 1 to be more reliable – we explicitly handle these after all – than when many values are close to 0.5. To incorporate this idea, we scale $\boldsymbol{o}$ by a confidence value in the interval $[0, 1]$.

We define $p_{\boldsymbol{a}}$ and p_D to be the average distances to 0.5. The choice of 0.5 is not important, because the module can learn to change it by changing where $f_6(x) = 0.5$ and $f_7(x) = 0.5$.

$$p_{\boldsymbol{a}} = \frac{1}{n} \sum_i |f_6(a_i) - 0.5| \tag{3.10}$$

$$p_D = \frac{1}{n^2} \sum_{i,j} |f_7(D_{ij}) - 0.5| \tag{3.11}$$

Then, the output of the module with confidence scaling is

$$\widetilde{\boldsymbol{o}} = f_8(p_{\boldsymbol{a}} + p_D) \cdot \boldsymbol{o} \tag{3.12}$$

In summary, we only used differentiable operations to deduplicate object proposals and obtain a feature vector that represents the predicted count. This allows easy integration into any model with soft attention, enabling a model to count from an attention map. Each step that we applied is either permutation-equivariant or permutation-invariant, which makes our whole model permutation-invariant.

3.5 Experiments

We provide the source code to reproduce our experiments at `https://github.com/Cyanogenoid/vqa-counting`.

3.5.1 Toy task

First, we design a simple toy task to evaluate counting ability. This dataset is intended to only evaluate the performance of counting; thus, we skip any processing steps that are not directly related such as the processing of an input image. Samples from this dataset are given in Figure 3.5.

The classification task is to predict an integer count $\hat{c}$ of true objects, uniformly drawn from 0 to 10 inclusive, from a set of bounding boxes and the associated attention weights. 10 square bounding boxes with side length $l \in (0, 1]$ are placed in a square image with unit side length. The x and y coordinates of their top left corners are uniformly drawn from $U(0, 1-l)$ so that the boxes do not extend beyond the image border. l is used to control the overlapping of bounding boxes: a larger l leads to the fixed number of objects to be more tightly packed, increasing the chance of overlaps. $\hat{c}$ number of these boxes are randomly chosen to be

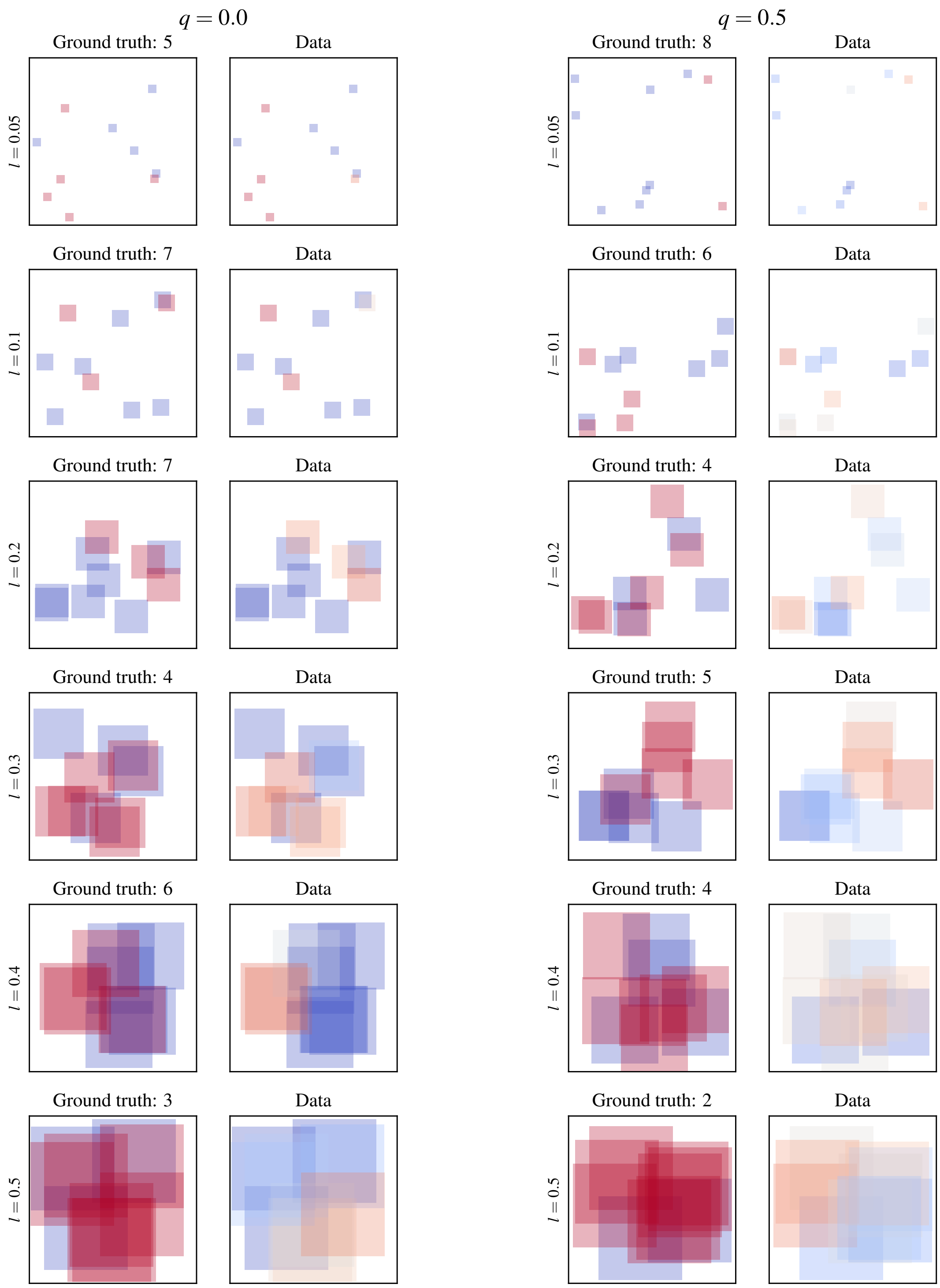


Figure 3.5: Example toy dataset data for varying bounding box side lengths l and noise q. The ground truth column shows bounding boxes of randomly placed true objects (blue) and of irrelevant objects (red). The data column visualises the samples that are actually used as input (dark blues represent weights close to 1, dark reds represent weights close to 0, lighter colours represent weights closer to 0.5). The weight of the ith bounding box b_i is defined as $a_i = (1 - q)$ score $+ qz$ where the score is the maximum overlap of b_i with any true bounding box or 0 if there are no true bounding boxes and z is drawn from $U(0, 1)$. Note how this turns red bounding boxes that overlap a lot with a blue bounding box in the ground truth column into a blue bounding box in the data column, which simulates the duplicate proposal that we have to deal with. Best viewed in colour.

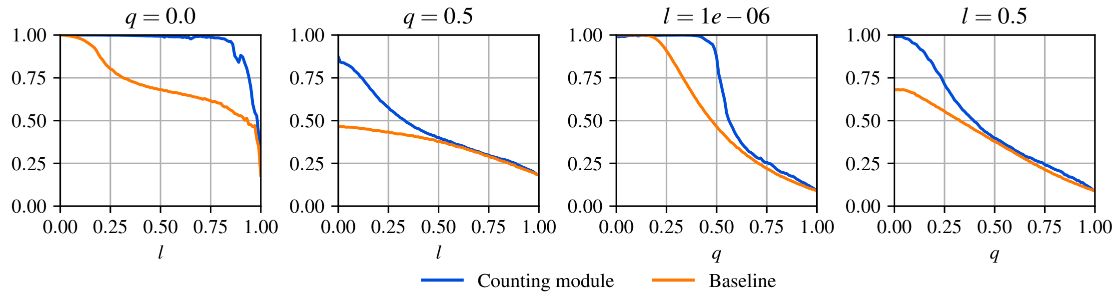


Figure 3.6: Accuracies on the toy task as side length l and noise q are varied in 0.01 step sizes. Our counting module is in blue, the baseline is in orange.

true bounding boxes. The *score* of a bounding box is the maximum IoU overlap of it with any true bounding box. Then, the attention weight is a linear interpolation between the score and a noise value drawn from $U(0, 1)$, with $q \in [0, 1]$ controlling this trade-off. q is the attention noise parameter: when q is 0, there is no noise and when q is 1, there is no signal. Increasing q also indirectly simulates imprecise placements of bounding boxes in real datasets.

We compare the counting module against a simple baseline that simply sums the attention weights and turns the sum into a feature vector with Equation 3.9. Both models are followed by a linear projection to the classes 0 to 10 inclusive and a softmax activation. They are trained with cross-entropy loss for 1000 iterations using Adam [58] with a learning rate of 0.01 and a batch size of 1024.

Results

The results of varying l while keeping q fixed at various values and vice versa are shown in Figure 3.6. Regardless of l and q, the counting module performs better than the baseline in most cases, often significantly so. Particularly when the noise is low, the module can deal with high values for l very successfully, showing that it accomplishes the goal of increased robustness to overlapping proposals. The module also handles moderate noise levels decently as long as the overlaps are limited. The performance when both l and q are high is closely matched by the baseline, likely due to the high difficulty of those parametrisations leaving little information to extract in the first place.

We can also look at the shape of the activation functions themselves, shown in Figure 3.7 (full version: Figure 3.8 and Figure 3.9), to understand how the behaviour changes with varying dataset parameters. For simplicity, we limit our description to the two easiest-to-interpret functions in Figure 3.7: f_1 for the attention weights and f_2 for the bounding box distances.

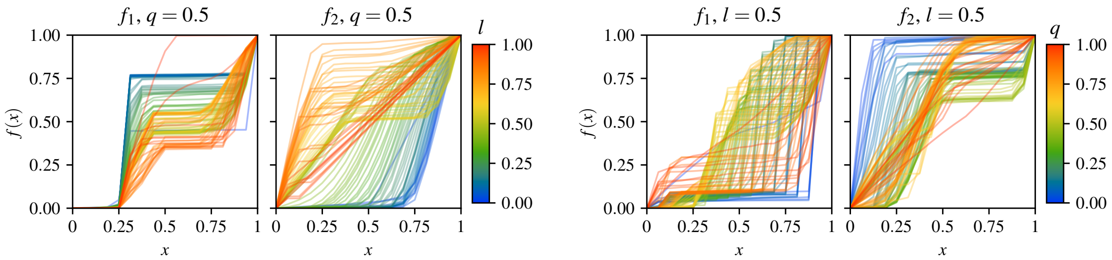


Figure 3.7: Shapes of trained activation functions f_1 (attention weights) and f_2 (bounding box distances) for varying bounding box side lengths (left) or the noise (right) in the dataset, varied in 0.01 step sizes. Best viewed in colour.

When increasing the side length, the height of the "step" in f_1 decreases to compensate for the generally greater degree of overlapping bounding boxes. A similar effect is seen with f_2: it varies over requiring a high pairwise distance when l is low – when partial overlaps are most likely spurious – and judging small distances enough for proposals to be considered different when l is high. At the highest values for l, there is little signal in the overlaps left since everything overlaps with everything, which explains why f_2 returns to its default linear initialisation for those parameters.

When varying the amount of noise, without noise f_1 resembles a step function where the step starts close to $x = 1$ and takes a value of close to 1 after the step. Since a true proposal will always have a weight of 1 when there is no noise, anything below this can be safely zeroed out. With increasing noise, this step moves away from 1 for both x and $f_1(x)$, capturing the uncertainty when a bounding box belongs to a true object. With lower q, f_2 considers a pair of proposals to be distinct for lower distances, whereas with higher q, f_2 follows a more sigmoidal shape. This can be explained by the model taking the increased uncertainty of the precise bounding box placements into account by requiring higher distances for proposals to be considered completely different.

3.5.2 VQA

VQA v2 [38] is the updated version of VQA v1 [6] where greater care has been taken to reduce dataset biases through balanced pairs: for each question, a pair of images is identified where the answer to that question differs. The standard accuracy metric on this dataset accounts for disagreements in human answers by averaging $\min(\frac{1}{3}\ \textit{agreeing}, 1)$ over all 10-choose-9 subsets of human answers, where *agreeing* is the number of human answers that agree with the given answer. This can be shown to be equal to $\min(0.3\ \textit{agreeing}, 1)$ without averaging.

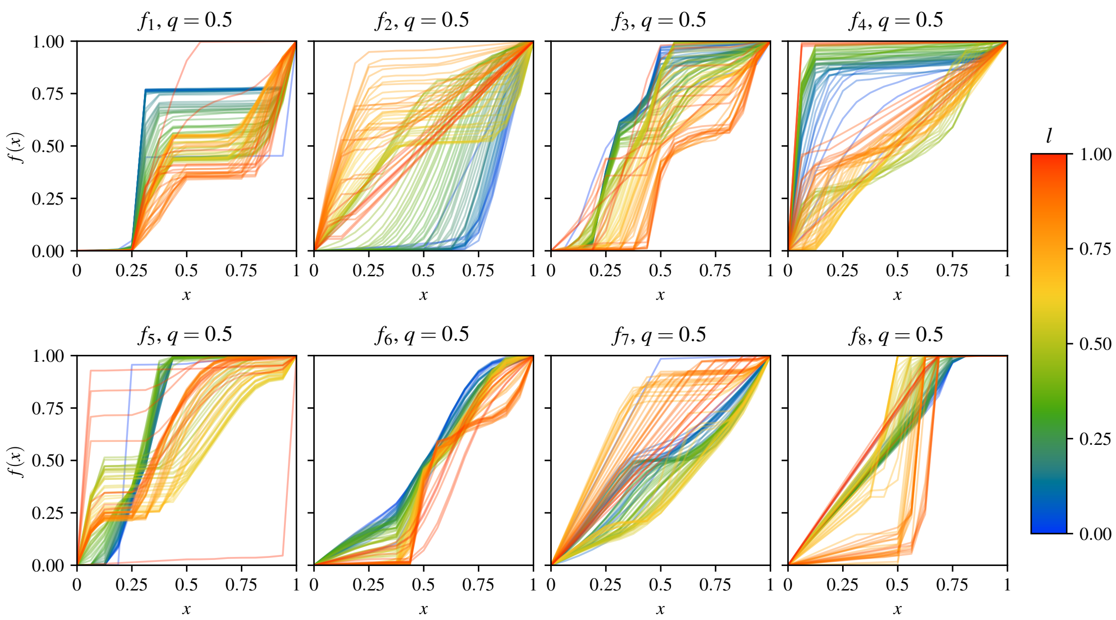


Figure 3.8: Shape of activation functions as l is varied for $q = 0.5$ on the toy dataset. Each line shows the shape of the activation function when l is set to the value associated to its colour. Best viewed in colour.

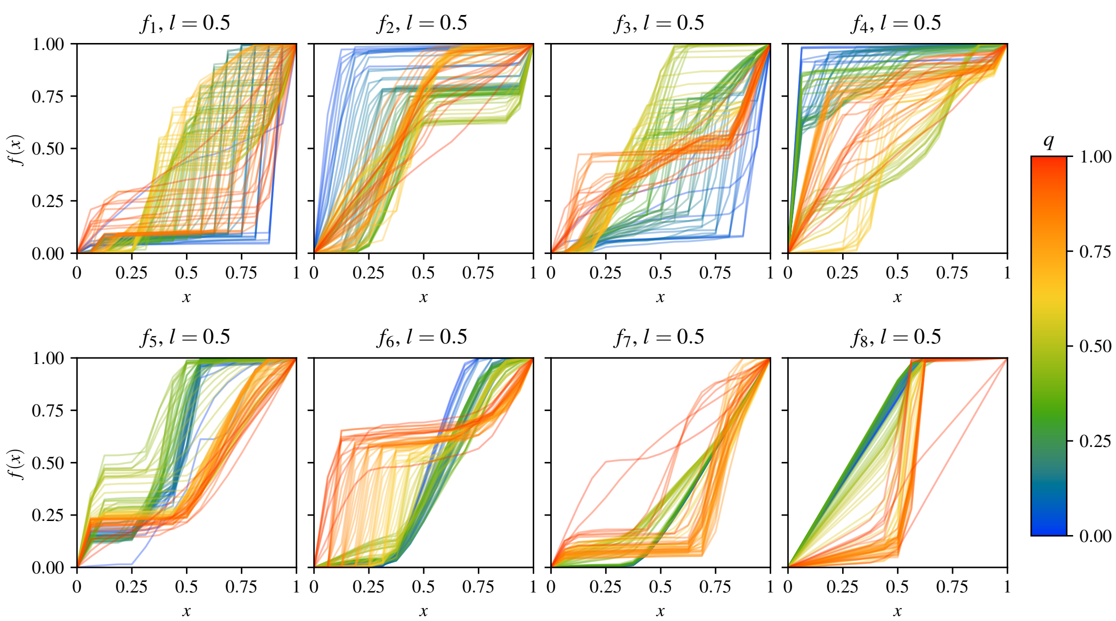


Figure 3.9: Shape of activation functions as q is varied for $l = 0.5$ on the toy dataset. Each line shows the shape of the activation function when q is set to the value associated to its colour. Best viewed in colour.

Aside: Simplifying the evaluation metric

We managed to simplify the official metric of averaging over the 10-choose-9 subsets as follows.

There are two cases for the one answer to be discarded:

1. the discarded answer *is not* the predicted answer
   $\implies$ accuracy stays the same
2. the discarded answer *is* the predicted answer
   $\implies$ we have to subtract 1 from the number of agreeing answers.

We use # to denote the number of human answers agreeing with the prediction. There are $10-\#$ of case 1 and # of case 2, therefore the accuracy is:

$$0.1 \cdot \left((10 - \#) \min\left(\frac{\#}{3}, 1\right) + \# \min\left(\frac{\# - 1}{3}, 1\right) \right) \tag{3.13}$$

We know that when $\# = 0$, the accuracy is 0, and when $\# \geq 4$, the accuracy is 1. In the remaining cases $1 \leq \# \leq 3$, we know that $\frac{\#-1}{3} < \frac{\#}{3} \leq 1$, so all the mins can be removed. This allows us to move the # outside:

$$0.1 \cdot \frac{1}{3} \cdot \# \left((10 - \#) + (\# - 1) \right) \tag{3.14}$$

which simplifies to $0.3\#$. Lastly, we can combine all the cases together to get $\min(0.3\#, 1)$ as accuracy metric.

Model Our baseline model is based on the work of Kazemi et al. [54], which outperformed most previous VQA models on the VQA v1 dataset with a simple baseline architecture. We adapt the model to the VQA v2 dataset and make various tweaks that improve validation accuracy slightly, which we describe in full in Section A.1. The architecture is illustrated in Figure 3.10.

We have not performed any tuning of this baseline to maximise the performance difference between it and the baseline with counting module. To augment this model with the counting module, we extract the attention weights of the first attention glimpse (there are two in the baseline) before softmax normalisation, and feed them into the counting module after applying a logistic function. Since object proposal features from Anderson et al. [4] vary from 10 to 100 per image, a natural choice for the number of top-n proposals to use is 10. The output of the module

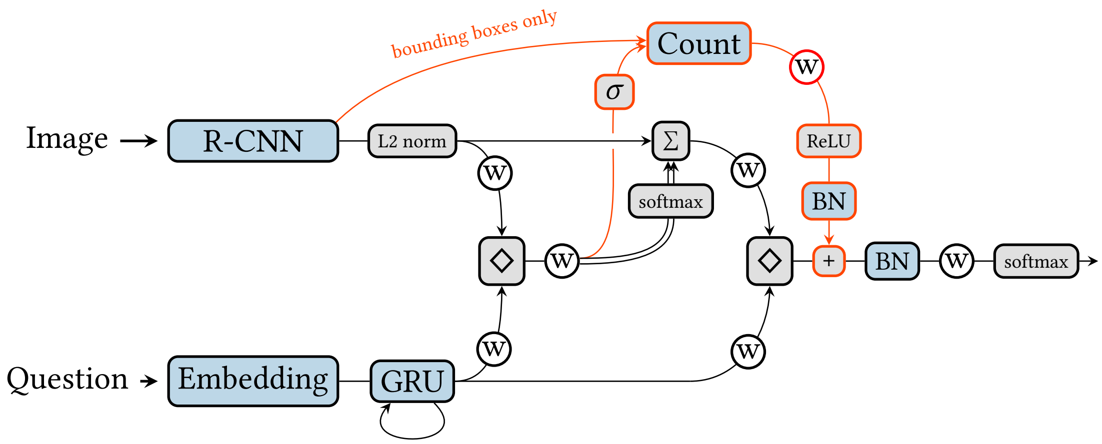


Figure 3.10: Schematic view of a model using our counting module. The modifications made to the baseline model when including the counting module are marked in red. Blue blocks mark modules with trainable parameters, grey blocks mark modules without trainable parameters. White ⓦ mark linear layers, either linear projections or convolutions with a spatial size of 1 depending on the context. Dropout with drop probability 0.5 is applied before the GRU and every ⓦ, except before the ⓦ after the counting module. ◇ stands for the fusion function we define in Section A.1, BN stands for batch normalisation, σ stands for a logistic, and Embedding is a word embedding that has been fed through a tanh function. The two glimpses of the attention mechanism are represented with the two lines exiting the ⓦ. Note that one of the two glimpses is shared with the counting module.

is linearly projected into the same space as the hidden layer of the classifier, followed by ReLU activation, batch normalisation, and addition with the features in the hidden layer.

Results Table 3.1 shows the results on the official VQA v2 leader board. The baseline with our module has a significantly higher accuracy on number questions without compromising accuracy on other categories compared to the baseline result. Despite our single-model baseline being substantially worse than the state-of-the-art, by simply adding the counting module we outperform even the 8-model ensemble in Zhou et al. [119] on the number category. We expect further improvements in number accuracy when incorporating their techniques to improve the quality of attention weights, especially since the current state-of-the-art models suffer from the problems with counting that we mention in Section 3.3. Some qualitative examples of inputs and activations within the counting module are shown in Figure 3.11.

Since the writing of this chapter, further VQA challenges have been run. The winners of the 2018 challenge were Kim et al. [56], who used our counting model to improve the counting results in their model. They achieved a 54.04% accuracy on the number category of the VQA v2 test set, which is a 2.65% improvement over our results. Without our counting module, they obtained 50.66% accuracy in the number category, so a significant part of their improved number results are

Table 3.1: Results on VQA v2 of the top models along with our results. Entries marked with (Ens.) are ensembles of models. At the time of writing, our model with the counting module places third among all entries. All models listed here use object proposal features and are trained on the training and validation sets. The top-performing ensemble models use additional pre-trained word embeddings, which we do not use.

	VQA v2 test-dev				VQA v2 test			
Model	Yes/No	**Number**	Other	All	Yes/No	**Number**	Other	All
Teney et al. [98]	81.82	44.21	56.05	65.32	82.20	43.90	56.26	65.67
Teney et al. [98] (Ens.)	86.08	48.99	60.80	69.87	86.60	48.64	61.15	70.34
Zhou et al. [119]	84.27	49.56	59.89	68.76	–	–	–	–
Zhou et al. [119] (Ens.)	–	–	–	–	86.65	51.13	61.75	70.92
Baseline	82.98	46.88	58.99	67.50	83.21	46.60	59.20	67.78
+ counting	83.14	**51.62**	58.97	68.09	83.56	**51.39**	59.11	68.41

Table 3.2: Results on the VQA v2 validation set with models trained only on the training set. Reported are the mean accuracies and sample standard deviations (±) over 4 random initialisations.

	VQA accuracy			Balanced pair accuracy		
Model	Number	**Count**	All	Number	**Count**	All
Baseline	$44.83_{\pm 0.2}$	$51.69_{\pm 0.2}$	$64.80_{\pm 0.0}$	$17.34_{\pm 0.2}$	$20.02_{\pm 0.2}$	$36.44_{\pm 0.1}$
+ NMS	$44.60_{\pm 0.1}$	$51.41_{\pm 0.1}$	$64.80_{\pm 0.1}$	$17.06_{\pm 0.1}$	$19.72_{\pm 0.1}$	$36.44_{\pm 0.2}$
+ counting	$49.36_{\pm 0.1}$	$\mathbf{57.03}_{\pm 0.0}$	$65.42_{\pm 0.1}$	$23.10_{\pm 0.2}$	$\mathbf{26.63}_{\pm 0.2}$	$37.19_{\pm 0.1}$

due to our model. In other words, they independently confirmed that a better attention model (which they developed) leads to even better counting results.

We also evaluate our models on the validation set of VQA v2, shown in Table 3.2. This allows us to consider only the counting questions within number questions, since number questions include questions such as "what time is it?" as well. We treat any question starting with the words "how many" as a counting question. As we expect, the benefit of using the counting module on the counting question subset is higher than on number questions in general. Additionally, we try an approach where we simply replace the counting module with NMS, using the average of the attention glimpses as scoring, and one-hot encoding the number of proposals left. The NMS-based approach, using an IoU threshold of 0.5 and no score thresholding based on validation set performance, does not improve on the baseline, which suggests that the piecewise gradient of NMS is a major problem for learning to count in VQA and that conversely, there is a substantial benefit to being able to differentiate through the counting module.

Additionally, we can evaluate the accuracy over balanced pairs as proposed by Teney et al. [98]: the ratio of balanced pairs on which the VQA accuracy for both questions is 1.0. This is a much more difficult metric, since it requires the model to find the subtle details between images

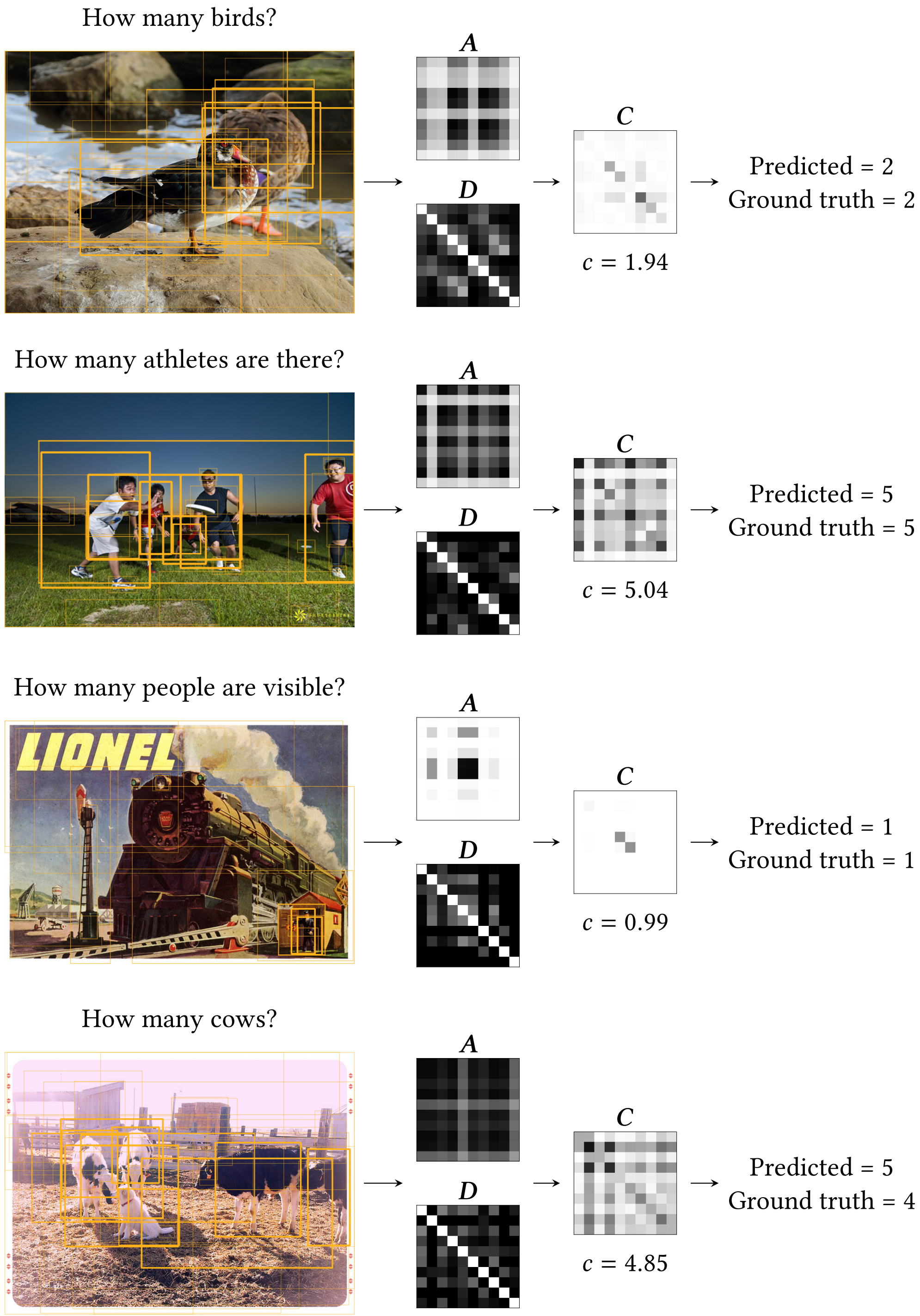


Figure 3.11: Selection of validation images with overlaid bounding boxes, values of the attention matrix $\boldsymbol{A}$, distance matrix $\boldsymbol{D}$, and the resulting count matrix $\boldsymbol{C}$. White entries represent values close to 1, black entries represent values close to 0. The count c is the usual square root of the sum over the elements of $\boldsymbol{C}$. Notice how particularly in the third example, $\boldsymbol{A}$ clearly contains more rows/columns with high activations than there are actual objects (a sign of overlapping bounding boxes) and the counting module successfully removes intra- and inter-object edges to arrive at the correct prediction regardless. The prediction is not necessarily – though often is – the rounded value of c.

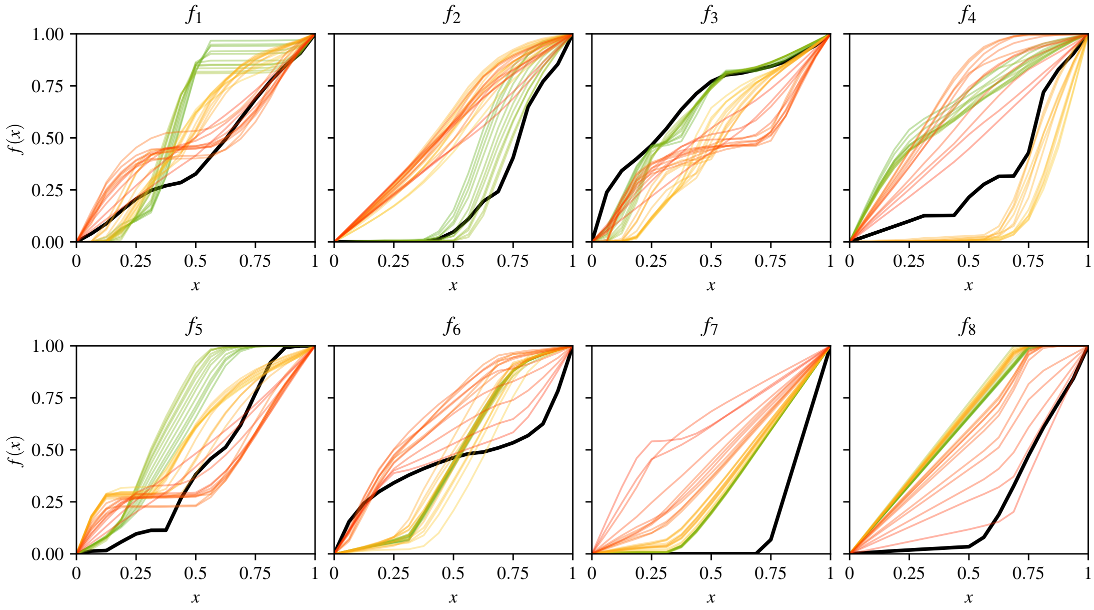


Figure 3.12: Shape of activation functions for a model trained on the train and validation sets of VQA v2 (thick black), compared against the shapes when parametrising the toy dataset with q around 0.4 (green), 0.7 (orange), or 1.0 (red) with fixed $l = 0.2$. Best viewed in colour.

instead of being able to rely on question biases in the dataset. First, notice how all balanced pair accuracies are greatly reduced compared to their respective VQA accuracy. More importantly, the absolute accuracy improvement of the counting module is still fully present with the more challenging metric, which is further evidence that the module can properly count rather than simply fitting better to dataset biases.

When looking at the activation functions of the trained model, shown in Figure 3.12, we find that some characteristics of them are shared with high-noise parametrisations of the toy dataset. This suggests that the current attention mechanisms and object proposal network are still very inaccurate, which explains the perhaps small-seeming increase in counting performance. This provides further evidence that the balanced pair accuracy is maybe a more reflective measure of how well current VQA models perform than the overall VQA accuracies of over 70% of the current top models.

3.6 Conclusion

After understanding why VQA models struggle to count, we designed a counting module that alleviates this problem through differentiable bounding box deduplication. The module can readily be used alongside any future improvements in VQA models, as long as they still use soft attention as all current top models on VQA v2 do. It has uses outside of VQA as well: for many counting tasks, it can allow an object-proposal-based approach to work without ground-truth objects available as long as there is a – possibly learned – per-proposal scoring (for example using a classification score) and a notion of how dissimilar a pair of proposals are. Since each step in the module has a clear purpose and interpretation, the learned weights of the activation functions are also interpretable. The design of the counting module is an example showing how by encoding inductive biases into a deep learning model, challenging problems such as counting of arbitrary objects can be approached when only relatively little supervisory information is available.

For future research, it should be kept in mind that VQA v2 requires a versatile skill set that current models do not have. To make progress on this dataset, we advocate focusing on *understanding* of what the current shortcomings of models are and finding ways to mitigate them.

Chapter 4

Set encoder: Permutation-optimisation

After having motivated the usefulness of sets through the VQA benchmark task, we move on to developing a method for extracting information from sets in a more general setting. Based on the idea that lists are easier to work with than sets, we aim to let the model *learn* how to turn a set into a list.

These contributions have been published as [117] in the International Conference on Learning Representations (ICLR) 2019.

4.1 Introduction

Consider a task where each input sample is a *set* of feature vectors with each feature vector describing an object in an image (for example: {person, table, cat}). Because there is no a priori ordering of these objects, it is important that the model is invariant to the order that the elements appear in the set. However, this puts restrictions on what can be learned efficiently. As we covered in Section 2.3, the typical approach is to compose elementwise operations with permutation-invariant reduction operations, such as summing [112] or taking the maximum [84] over the whole set. Since the reduction operator compresses a set of any size down to a single descriptor, this can be a significant bottleneck in what information about the set can be represented efficiently.

We take an alternative approach based on an idea explored in Vinyals et al. [101], where they find that some permutations of sets allow for easier learning on a task than others. They do this by ordering the set elements in some predetermined way and feeding the resulting sequence into a recurrent neural network. For instance, it makes sense that if the task is to output the top-n numbers from a set of numbers, it is useful if the input is already sorted in descending order before being fed into an RNN. This approach leverages the representational capabilities of traditional sequential models such as LSTMs, but requires some prior knowledge of what order might be useful.

Our idea is to learn such a permutation purely from data *without requiring a priori knowledge* (Section 4.2). The key aspect is to turn a set into a sequence in a way that is both permutation-invariant, as well as differentiable so that it is learnable. Our main contribution is a Permutation-optimisation (PO) module that satisfies these requirements: it optimises a permutation in the forward pass of a neural network using pairwise comparisons. By feeding the resulting sequence into a traditional model such as an LSTM, we can learn a flexible, permutation-invariant representation of the set while avoiding the bottleneck that a simple reduction operator would introduce. Techniques used in our model may also be applicable to other set problems where permutation-invariance is desired, building on the literature of approaches to dealing with permutation-invariance (Section 4.3).

In four different experiments, we show improvements over existing methods (Section 4.4). The former two tasks measure the ability to learn a particular permutation as target: number sorting and image mosaics. We achieve state-of-the-art performance with our model, which shows that our method is suitable for representing permutations in general. The latter two tasks test whether a model can learn to solve a task that requires it to come up with a suitable permutation implicitly: classification from image mosaics and visual question answering. We provide no supervision of what the permutation should be; the model has to learn by itself what permutation is most useful for the task at hand. Here, our model also beats the existing models and we improve the performance of a state-of-the-art model in visual question answering (VQA) with it. This shows that our PO module is able to learn good permutation-invariant representations of sets using our approach.

4.2 Permutation-optimisation module

We will now describe a differentiable, and thus learnable model to turn an *unordered* set $\{\boldsymbol{x}_i\}_N$ with feature vectors as elements into an *ordered* sequence of these feature vectors. An overview of the algorithm is shown in Figure 4.1 and pseudo-code is shown in Algorithm 4.1. The input set is represented as a matrix $\boldsymbol{X} = [\boldsymbol{x}_1, \dots, \boldsymbol{x}_N]^\top$ with the feature vectors $\boldsymbol{x}_i$ as rows in some arbitrary order. Note that this representation of sets is transposed compared to Chapter 2 ($\mathbb{R}^{n \times d}$ instead of $\mathbb{R}^{d \times n}$) to ease the exposition in this chapter. In the algorithm, it is important to not rely on the arbitrary order so that $\boldsymbol{X}$ is correctly treated as a set. The goal is then to learn a permutation matrix $\boldsymbol{P}$ such that when permuting

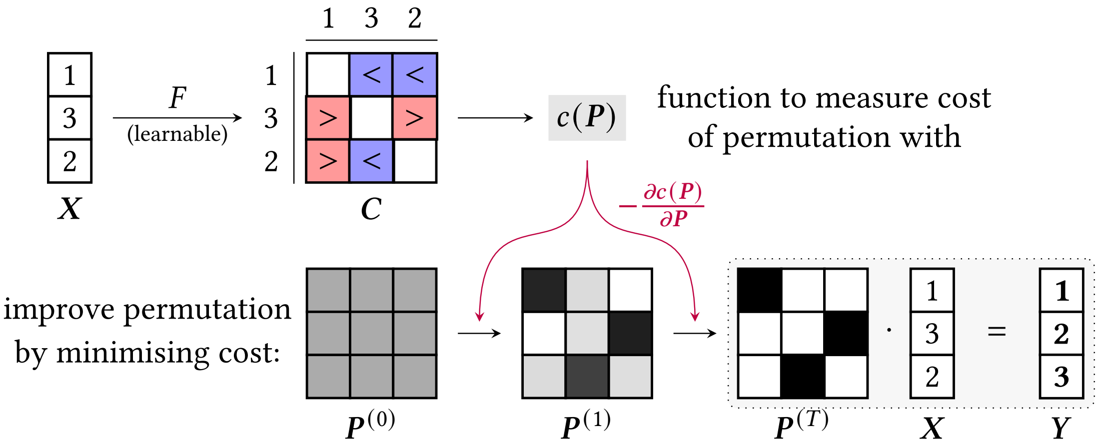


Figure 4.1: Overview of Permutation-optimisation module. In the ordering cost $\boldsymbol{C}$, elements of $\boldsymbol{X}$ are compared to each other (blue represents a negative value, red represents a positive value). Gradients are applied to unnormalised permutations $\widetilde{\boldsymbol{P}}^{(t)}$, which are normalised to proper permutations $\boldsymbol{P}^{(t)}$.

Algorithm 4.1 Forward pass of permutation-optimisation algorithm

1: **Input**: $\boldsymbol{X} \in \mathbb{R}^{N \times M}$ with $\boldsymbol{x}_i$ as rows in arbitrary order
2: **Learnable parameters**: weights that parametrise F, step size η
3:
4: $C_{ij} \leftarrow \text{normed}(F(\boldsymbol{x}_i, \boldsymbol{x}_j))$ ▹ ordering costs (Equation 4.15)
5: initialise $\widetilde{\boldsymbol{P}}$ ▹ uniform or linear assignment init (Equation 4.11)
6: **for** t $\leftarrow 1, T$ **do**
7: $\quad \boldsymbol{P} \leftarrow S(\widetilde{\boldsymbol{P}})$ ▹ normalise assignment (Equation 4.8
8: $\quad \boldsymbol{G} \leftarrow \partial c(\boldsymbol{P}) / \partial \boldsymbol{P}$ ▹ compute gradient of assignment (Equation 4.9)
9: $\quad \widetilde{\boldsymbol{P}} \leftarrow \widetilde{\boldsymbol{P}} - \eta \boldsymbol{G}$ ▹ gradient descent step on assignment (Equation 4.12)
10: **end for**
11: $\boldsymbol{P} \leftarrow S(\widetilde{\boldsymbol{P}})$
12: $\boldsymbol{Y} \leftarrow \boldsymbol{P}\boldsymbol{X}$ ▹ permute rows of $\boldsymbol{X}$ to obtain output $\boldsymbol{Y}$

the rows of the input through $\boldsymbol{Y} = \boldsymbol{P}\boldsymbol{X}$, the output is ordered correctly according to the task at hand. When an entry P_{ik} takes the value 1, it can be understood as assigning the ith element to the kth position in the output.

Our main idea is to first relate pairs of elements through an *ordering cost*, parametrised with a neural network. This pairwise cost tells us whether an element i should preferably be placed before or after element j in the output sequence. Using this, we can define a *total cost* that measures how good a given permutation is (Subsection 4.2.1). The second idea is to optimise this total cost in each forward pass of the module (Subsection 4.2.2). By minimising the total cost of a permutation, we improve the quality of a permutation with respect to the current ordering costs. Crucially, the ordering cost function – and thus also the total cost function – is learned. In doing so, the module is able to learn how to generate a permutation as is desired.

In order for this to work, it is important that the optimisation process itself is differentiable so that the ordering cost is learnable. Because permutations are inherently discrete objects, a continuous relaxation of permutations is necessary. For optimisation, we perform gradient descent on the total cost for a fixed number of steps and unroll the iteration, similar to how recurrent neural networks are unrolled to perform backpropagation-through-time. Because the *inner gradient* (total cost differentiated with respect to permutation) is itself differentiable with respect to the ordering cost, the whole model is kept differentiable and we can train it with a standard supervised learning loss.

Note that as long as the ordering cost is computed appropriately (Subsection 4.2.3), all operations used turn out to be permutation-equivariant or invariant. Thus, we have a model that respects the symmetries of sets while producing an output without those symmetries: a sequence. This can be naturally extended to outputs where the target is not a sequence, but grids and lattices (Subsection 4.2.4).

4.2.1 Total cost function

The total cost function measures the quality of a given permutation and should be lower for better permutations. Because this is the function that will be optimised, it is important to understand what it expresses precisely.

The main ingredient for the total cost of a permutation is the pairwise ordering cost (details in Subsection 4.2.3). By computing it for all pairs, we obtain a cost matrix $\boldsymbol{C}$ where the entry C_{ij} represents the ordering cost between i and j: the cost of placing element i anywhere before j in the output sequence. An important constraint that we put on $\boldsymbol{C}$ is that $C_{ij} = -C_{ji}$. In other words, if one ordering of i and j is "good" (negative cost), then the opposite ordering obtained by swapping them is "bad" (positive cost). Additionally, this constraint means that $C_{ii} = 0$. This makes sure that two very similar feature vectors in the input will be similarly ordered in the output because their pairwise cost goes to 0.

In this chapter we use a straightforward definition of the total cost function: a sum of the ordering costs over all pairs of elements i and j. When considering the pair i and j, if the permutation maps i to be before j in the output sequence, this cost is simply C_{ij}. Vice versa, if the permutation maps i to be after j in the output sequence, the cost has to be flipped to C_{ji}. To express this idea, we define the total cost $c\colon \mathbb{R}^{N\times N} \mapsto \mathbb{R}$ of a permutation $\boldsymbol{P}$ as:

$$c(\boldsymbol{P}) = \sum_{ij} C_{ij} \sum_{k} P_{ik} \left(\sum_{k'>k} P_{jk'} - \sum_{k'<k} P_{jk'} \right) \tag{4.1}$$

This can be understood as follows: If the permutation assigns element i to position u (so $P_{iu} = 1$) and element j to position v (so $P_{jv} = 1$), the sums over k and k' simplify to 1 when $v > u$ and -1 when $v < u$; permutation matrices are binary and only have one 1 in any row and column, so all other terms in the sums are 0. That means that the term for each i and j becomes C_{ij} when $v > u$ and $-C_{ij} = C_{ji}$ when $v < u$, which matches what we described previously.

4.2.2 Optimisation problem

Now that we can compute the total cost of a permutation, we want to optimise this cost with respect to a permutation. After including the constraints to enforce that $\boldsymbol{P}$ is a valid permutation matrix, we obtain the following optimisation problem:

$$\begin{aligned} \underset{\boldsymbol{P}}{\text{minimize}} \quad & c(\boldsymbol{P}) \\ \text{subject to} \quad & \forall i, k\colon P_{ik} \in \{0, 1\}, \\ & \forall i\colon \sum_k P_{ik} = 1, \sum_k P_{ki} = 1 \end{aligned} \tag{4.2}$$

Optimisation over $\boldsymbol{P}$ directly is difficult due to the discrete and combinatorial nature of permutations. To make optimisation feasible, a common relaxation is to replace the constraint that $P_{ik} \in \{0, 1\}$ with $P_{ik} \in [0, 1]$ [33]. With this change, the feasible set for $\boldsymbol{P}$ expands to the set of doubly-stochastic matrices, known as the Birkhoff or assignment polytope. Rather than hard permutations, we now have soft assignments of elements to positions, analogous to the latent assignments when fitting a mixture of Gaussians model using Expectation-Maximisation.

Note that we do not need to change our total cost function after this relaxation. Instead of discretely flipping the sign of C_{ij} depending on whether element i comes before j or not, the sums over k and k' give us a weight for each C_{ij} that is based on how strongly i and j are assigned to positions. This weight is positive when i is on average assigned to earlier positions than j and negative vice versa.

In order to perform optimisation of the cost under our constraints, we reparametrise $\boldsymbol{P}$ with the Sinkhorn operator S from Adams et al. [2] so that the constraints are always satisfied.

We found this to lead to better solutions than projected gradient descent in initial experiments. After first exponentiating all entries of a matrix, S repeatedly normalises all rows, then all columns of the matrix to sum to 1, which converges to a doubly-stochastic matrix in the limit.

$$\boldsymbol{P} = S(\widetilde{\boldsymbol{P}}) \tag{4.3}$$

This ensures that $\boldsymbol{P}$ is always approximately a doubly-stochastic matrix. $\widetilde{\boldsymbol{P}}$ can be thought of as the unnormalised permutation while $\boldsymbol{P}$ is the normalised permutation. By changing our optimisation to minimise $\widetilde{\boldsymbol{P}}$ instead of $\boldsymbol{P}$ directly, all constraints are always satisfied and we can simplify the optimisation problem to $\min_{\widetilde{\boldsymbol{P}}} c(\boldsymbol{P})$ without any constraints.

Aside: Sinkhorn operator

The Sinkhorn operator S as defined in Adams et al. [2] is:

$$\mathcal{T}_r(\boldsymbol{X})_{ij} = X_{ij} / \sum_k X_{ik} \tag{4.4}$$

$$\mathcal{T}_c(\boldsymbol{X})_{ij} = X_{ij} / \sum_k X_{kj} \tag{4.5}$$

$$S^{(0)}(\boldsymbol{X}) = \exp(\boldsymbol{X}) \tag{4.6}$$

$$S^{(l+1)}(\boldsymbol{X}) = \mathcal{T}_c\left(\mathcal{T}_r\left(S^{(l)}(\boldsymbol{X})\right)\right) \tag{4.7}$$

$$S(\boldsymbol{X}) = S^{(L)}(\boldsymbol{X}) \tag{4.8}$$

$\mathcal{T}_r$ normalises each row, $\mathcal{T}_c$ normalises each column of a square matrix $\boldsymbol{X}$ to sum to one. This formulation is different from the normal Sinkhorn operator by Sinkhorn [94] by exponentiating all entries first and running for a fixed number of steps L instead of for steps approaching infinity. Mena et al. [71] include a temperature parameter on the exponentiation, which acts analogously to temperature in the softmax function. In this chapter, we fix L to 4.

It is now straightforward to optimise $\widetilde{\boldsymbol{P}}$ with standard gradient descent. First, we compute the gradient:

$$\frac{\partial c(\boldsymbol{P})}{\partial P_{pq}} = 2 \sum_j C_{pj} \left(\sum_{k'>q} P_{jk'} - \sum_{k'<q} P_{jk'} \right) \tag{4.9}$$

$$\frac{\partial c(\boldsymbol{P})}{\partial \widetilde{P}_{pq}} = \frac{\partial \boldsymbol{P}}{\partial \widetilde{P}_{pq}} \cdot \frac{\partial c(\boldsymbol{P})}{\partial \boldsymbol{P}} \tag{4.10}$$

From Equation 4.9, it becomes clear that this gradient is itself differentiable with respect to the ordering cost C_{ij}, which allows it to be learned. In practice, both $\partial c(\boldsymbol{P})/\partial \widetilde{\boldsymbol{P}}$ as well as $\partial[\partial c(\boldsymbol{P})/\partial \widetilde{\boldsymbol{P}}]/\partial \boldsymbol{C}$ can be computed with automatic differentiation. However, some implementations of automatic differentiation require the computation of $c(\boldsymbol{P})$ which we do not use. In this case, implementing $\partial c(\boldsymbol{P})/\partial \widetilde{\boldsymbol{P}}$ explicitly can be more efficient. Also notice that if we define $B_{jq} = \sum_{k'>q} P_{jk'} - \sum_{k'<q} P_{jk'}$, Equation 4.9 is just the matrix multiplication $\boldsymbol{CB}$ and is thus efficiently computable.

For the optimisation, $\boldsymbol{P}$ has to be initialised in a permutation-equivariant way to preserve permutation-invariance of the algorithm. In this chapter, we consider a uniform initialisation so that all $P_{ik} = 1/N$ (**PO-U** model, left) and an initialisation that linearly assigns [71] each element to each position (**PO-LA** model, right).

$$\widetilde{P}_{ik}^{(0)} = 0 \quad \text{or} \quad \widetilde{P}_{ik}^{(0)} = \boldsymbol{w}_k \boldsymbol{x}_i \tag{4.11}$$

where $\boldsymbol{w}_k$ is a different weight vector for each position k. Then, we perform gradient descent for a fixed number of steps T. The iterative update using the gradient and a (learnable) step size η converges to the optimised permutation $\boldsymbol{P}^{(T)}$:

$$\widetilde{\boldsymbol{P}}^{(t+1)} = \widetilde{\boldsymbol{P}}^{(t)} - \eta \, \frac{\partial c(\boldsymbol{P}^{(t)})}{\partial \boldsymbol{P}^{(t)}} \tag{4.12}$$

One peculiarity of this is that we update $\widetilde{\boldsymbol{P}}$ with the gradient of the normalised permutation $\boldsymbol{P}$, not of the unnormalised permutation $\widetilde{\boldsymbol{P}}$ as normal. In other words, we do gradient descent on $\widetilde{\boldsymbol{P}}$ but in Equation 4.10 we set $\partial P_{uv}/\partial \widetilde{P}_{pq} = 1$ when $u = p, v = q$, and 0 everywhere else. We found that this results in significantly better permutations experimentally; we believe that this is because $\partial \boldsymbol{P}/\partial \widetilde{\boldsymbol{P}}$ vanishes too quickly from the Sinkhorn normalisation, which biases $\boldsymbol{P}$ away from good permutation matrices wherein all entries are close to 0 and 1. We justify this in more detail in Subsection 4.2.5.

The runtime of this algorithm is dominated by the computation of gradients of $c(\boldsymbol{P})$, which involves a matrix multiplication of two $N \times N$

matrices. In total, the time complexity of this algorithm is T times the complexity of this matrix multiplication, which is $\Theta(N^3)$ in practice. We found that typically, small values for T such as 4 are enough to get good permutations.

4.2.3 Ordering cost function

The ordering cost C_{ij} is used in the total cost and tells us what the pairwise cost for placing i before j should be. The key property to enforce is that the function F that produces the entries of $\boldsymbol{C}$ is anti-symmetric ($F(\boldsymbol{x}_i, \boldsymbol{x}_j) = -F(\boldsymbol{x}_j, \boldsymbol{x}_i)$). A simple way to achieve this is to define F as:

$$F(\boldsymbol{x}_i, \boldsymbol{x}_j) = f(\boldsymbol{x}_i, \boldsymbol{x}_j) - f(\boldsymbol{x}_j, \boldsymbol{x}_i) \tag{4.13}$$

We can then use a small neural network for f to obtain a learnable F that is always anti-symmetric.

Lastly, $\boldsymbol{C}$ is normalised to have unit Frobenius norm. This results in simply scaling the total cost obtained, but it also decouples the scale of the outputs of F from the step size parameter η to make optimisation more stable at inference time. $\boldsymbol{C}$ is then defined as:

$$\widetilde{C}_{ij} = F(\boldsymbol{x}_i, \boldsymbol{x}_j) \tag{4.14}$$

$$C_{ij} = \widetilde{C}_{ij} / \|\widetilde{\boldsymbol{C}}\|_F \tag{4.15}$$

4.2.4 Extending permutations to lattices

In some tasks, it may be natural to permute the set into a lattice structure instead of a sequence. For example, if it is known that the set contains parts of an image, it makes sense to arrange these parts back to an image by using a regular grid. We can straightforwardly adapt our model to this by considering each row and column of the target grid as an individual permutation problem. The total cost of an assignment to a grid is the sum of the total costs over all individual rows and columns of the grid. The gradient of this new cost is then the sum of the gradients of these individual problems. This results in a model that considers both row-wise and column-wise pairwise relations when permuting a set of inputs into a grid structure, and more generally, into a lattice structure.

4.2.5 Justification for alternative update

We mentioned previously that we perform gradient descent with the post-Sinkhorn gradients on the pre-Sinkhorn matrix. Here, we justify why this is a reasonable thing to do.

First, the gradient of $S(\boldsymbol{X})$ is:

$$\frac{\partial S(\boldsymbol{X})}{\partial \boldsymbol{X}} = \frac{\partial \exp(\boldsymbol{X})}{\partial \boldsymbol{X}} \frac{\partial \mathcal{T}_r\big(\exp(\boldsymbol{X})\big)}{\partial \exp(\boldsymbol{X})} \frac{\partial \mathcal{T}_c\Big(\mathcal{T}_r\big(\exp(\boldsymbol{X})\big)\Big)}{\partial \mathcal{T}_r\big(\exp(\boldsymbol{X})\big)} \cdots \tag{4.16}$$

$$\frac{\partial \mathcal{T}_r(\boldsymbol{X})_{uv}}{\partial X_{ij}} = \mathbb{1}_{u=i} \frac{\mathbb{1}_{v=j} \sum_k X_{uk} - X_{uv}}{(\sum_k X_{uk})^2} \tag{4.17}$$

$$\frac{\partial \mathcal{T}_c(\boldsymbol{X})_{uv}}{\partial X_{ij}} = \mathbb{1}_{v=j} \frac{\mathbb{1}_{u=i} \sum_k X_{kv} - X_{uv}}{(\sum_k X_{kv})^2} \tag{4.18}$$

where $\mathbb{1}$ is the indicator function that returns 1 if the condition is true and 0 otherwise.

We compared the entropy of the permutation matrices obtained with and without using the “proper” gradient with $\partial S(\widetilde{\boldsymbol{P}})/\partial \widetilde{\boldsymbol{P}}$ as term in it and found that our version has a significantly lower entropy. To understand this, it is enough to focus on the first two terms in Equation 4.16, which is essentially the gradient of a softmax function applied row-wise to $\boldsymbol{P}$.

Let $\boldsymbol{x}$ be a row in $\boldsymbol{P}$ and s_i be the ith entry in the softmax function applied to $\boldsymbol{x}$. Then, the gradient is:

$$\frac{\partial s_i}{\partial x_j} = s_i(\mathbb{1}_{i=j} - s_j) \tag{4.19}$$

Since this is a product of entries in a probability distribution, the gradient vanishes quickly as we move towards a proper permutation matrix (all entries very close to 0 or 1). By using our alternative update and thus removing this term from our gradient, we can avoid the vanishing gradient problem.

Gradient descent is not efficient when the gradient vanishes towards the optimum and the optimum – in our case a permutation matrix with exact ones and zeros as entries – is infinitely far away. Since we prefer to use a small number of steps in our algorithm for efficiency, we want to reach a good solution as quickly as possible. This justifies effectively ignoring the step size that the gradient suggests and simply taking a step in a similar direction as the gradient in order to be able to saturate the Sinkhorn normalisation sufficiently, thus obtaining a doubly stochastic matrix that is closer to a proper permutation matrix in the end.

4.2.6 Quadratic programming formulation

We can write our total cost function as a quadratic program in the standard $\boldsymbol{x}^\mathsf{T}\boldsymbol{Q}\boldsymbol{x}$ form with linear constraints. (We leave out the constraints here as they are not particularly interesting.) This connects our work to the extensive quadratic programming literature.

First, we can define $\boldsymbol{O} \in \mathbb{R}^{N \times N}$ as:

$$O_{kk'} = \begin{cases} -1 & \text{if } k > k' \\ 0 & \text{if } k = k' \\ 1 & \text{if } k < k' \end{cases} \tag{4.20}$$

and with it, $\boldsymbol{Q} \in \mathbb{R}^{N^2 \times N^2}$ as:

$$Q_{(ik)(jk')} = C_{ij} O_{kk'} \tag{4.21}$$

Then we can write the cost function as:

$$\begin{aligned} c(\boldsymbol{P}) &= \sum_{ij} C_{ij} \sum_{kk'} P_{ik} P_{jk'} O_{kk'} && (4.22) \\ &= \sum_{ik} \sum_{jk'} P_{ik} (C_{ij} O_{kk'}) P_{jk'} && (4.23) \\ &= \sum_{(ik)} \sum_{(jk')} P_{(ik)} Q_{(ik)(jk')} P_{(jk')} && (4.24) \\ &= \boldsymbol{p}^\mathsf{T} \boldsymbol{Q} \boldsymbol{p} && (4.25) \end{aligned}$$

where there is some bijection between a pair of indices (i, k) and the index l, and $\boldsymbol{p}$ is a flattened version of $\boldsymbol{P}$ with $p_l = P_{ik}$. $\boldsymbol{Q}$ is indefinite because the total cost can be negative: a uniform initialisation for $\boldsymbol{P}$ has a cost of 0, better permutations have negative cost, worse permutations have positive cost. Thus, the problem is non-convex and the problem is possibly NP-hard. Also, since we have flattened $\boldsymbol{P}$ into $\boldsymbol{p}$, the number of optimisation variables is quadratic in the set size N. Even if this were a convex quadratic program, methods such as OptNet [3] have cubic time complexity in the number of optimisation variables, which makes it $O(N^6)$ for our case.

4.3 Related work

The most relevant work to ours is the inspiring study by Mena et al. [71], where they discuss the reparametrisation that we use and propose a model that can also learn permutations implicitly in principle. Their model uses a simple elementwise linear map from each of the N elements of the set to the N positions, normalised by the Sinkhorn operator. This can be understood as classifying each element individually into one of the N classes corresponding to positions, then normalising the predictions so that each class only occurs once within this set. However, processing the elements individually means that their model does not take relations between elements into account properly; elements are placed in absolute positions, not relative to other elements. Our model differs from theirs by considering pairwise relations when creating the permutation. By basing the cost function on pairwise comparisons, it is able to order elements such that local relations in the output are taken into account. We believe that this is important for learning from permutations implicitly, because networks such as CNNs and RNNs rely on local ordering more than absolute positioning of elements. It also allows our model to process variable-sized sets, which their model is not able to do.

Our work is closely related to the set function literature, where the main constraint is invariance to ordering of the set. While it is always possible to simply train using as many permutations of a set as possible, using a model that is naturally permutation-invariant increases learning and generalisation capabilities through the correct inductive bias in the model. There are some similarities with relation networks [89] in considering all pairwise relations between elements as in our pairwise ordering function. However, they sum over all non-linearly transformed pairs, which can lead to the bottleneck we mention in Section 4.1. Meanwhile, by using an RNN on the output of our model, our approach can encode a richer class of functions: it can still learn to simply sum the inputs, but it can also learn more complex functions where the learned order between elements is taken into account. The concurrent work by Murphy et al. [76] discusses various approximations of averaging the output of a neural network over all possible permutations, with our method falling under their categorisation of a learned canonical input ordering. Our model is also relevant to neural networks operating on graphs such as graph convolutional networks [59]. Typically, a set function is applied to the set of neighbours for each node, with which the state of the node is updated. Our module combined with an RNN is thus an alternative set function to perform this state update with.

Noroozi et al. [79] and Cruz et al. [26] show that it is possible to use permutation learning for representation learning in a self-supervised setting. The model in Cruz et al. [26] is very similar to Mena et al. [71], including use of a Sinkhorn operator, but they perform significantly more processing on images with a large CNN (AlexNet) beforehand with the main goal of learning good representations for that CNN. We instead focus on using the permuted set itself for representation learning in a supervised setting.

We are not the first to explore the usefulness of using optimisation in the forward pass of a neural network (for example, Stoyanov et al. [97], Domke [29], and Belanger et al. [13]). However, we believe that we are the first to show the potential of optimisation for processing sets because – with an appropriate cost function – it is able to preserve permutation-invariance. In OptNet [3], exact solutions to convex quadratic programs are found in a differentiable way through various techniques. Unfortunately, our quadratic program is non-convex, which makes finding an optimal solution possibly NP-hard [81]. We thus fall back to the simpler approach of gradient descent on the reparametrised problem to obtain a non-optimal, but reasonable solution.

Note that our work differs from learning to rank approaches such as Burges et al. [18] and Severyn et al. [90], as there the end goal is the permutation itself. This usually requires supervision on what the target permutation should be, producing a permutation with hard assignments at the end. We require our model to produce soft assignments so that it is easily differentiable, since the main goal is not the permutation itself, but processing it further to form a representation of the set being permuted. This means that other approaches that produce hard assignments such as Ptr-Net [102] are also unsuitable for implicitly learning permutations, although using a variational approximation through Mena et al. [71] to obtain a differentiable permutation with hard assignments is a promising direction to explore for the future. Due to the lack of differentiability, existing literature on solving minimum feedback arc set problems [21] can not be easily used for set representation learning either.

4.4 Experiments

Throughout the text, we will refer to our model with uniform assignment as PO-U, with linear assignment initialisation as PO-LA, and the model from Mena et al. [71] as LinAssign. We perform a qualitative analysis of what comparisons are learned in Section 4.5. Precise experimental details can be found in Section A.2 and our implementation for all experiments is available at `https://github.com/Cyanogenoid/perm-optim` for full reproducibility.

An interesting aspect we observed throughout all experiments is how the learned step size η changes during training. At the start of training, it decreases from its initial value of 1, thus reducing the influence of the permutation mechanism. Then, η starts rising again, usually ending up at a value above 1 at the end of training. This can be explained by the ordering cost being very inaccurate at the start of training, since it has

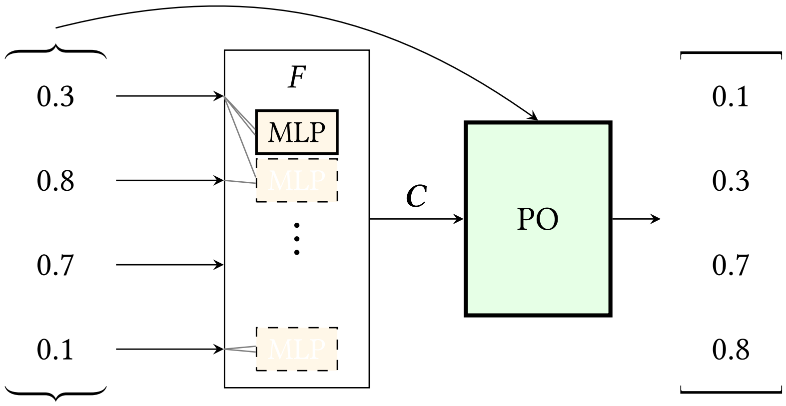


Figure 4.2: Network architecture for number sorting. The MLP in F computes f and is shared across pairs of set elements. The PO block performs the optimisation with the given costs and permutes the input set.

not been trained yet. Through training, the ordering cost improves and it becomes more beneficial for the influence of the PO module on the permutation to increase.

4.4.1 Sorting numbers

We start with the toy task of turning a set of random unsorted numbers into a sorted list. For this problem, we train with fixed-size sets of numbers drawn uniformly from the interval $[0, 1]$ and evaluate on different intervals to determine generalisation ability (for example: $[0, 1]$, $[0, 1000]$, $[1000, 1001]$). We use the correctly ordered sequence as training target and minimise the mean squared error. Following Mena et al. [71], during evaluation we use the Hungarian algorithm for solving a linear assignment problem with $-\boldsymbol{P}$ as the assignment costs. This is done to obtain a permutation with hard assignments from our soft permutation. We show our model architecture in Figure 4.2.

Results Our PO-U model is able to sort all sizes of sets that we tried – 5 to 1024 numbers – perfectly, including generalising to all the different evaluation intervals without any mistakes. This is in contrast to all existing end-to-end learning-based approaches such as Mena et al. [71], which starts to make mistakes on $[0, 1]$ at 120 numbers and no longer generalises to sets drawn from $[1000, 1001]$ at 80 numbers. Vinyals et al. [101] already starts making mistakes on 5 numbers. Our stark improvement over existing results is evidence that the inductive biases due to the learned pairwise comparisons in our model are suitable for learning permutations, at least for this particular toy problem. In Subsection 4.5.1, we investigate what it learns that allows it to generalise this well.

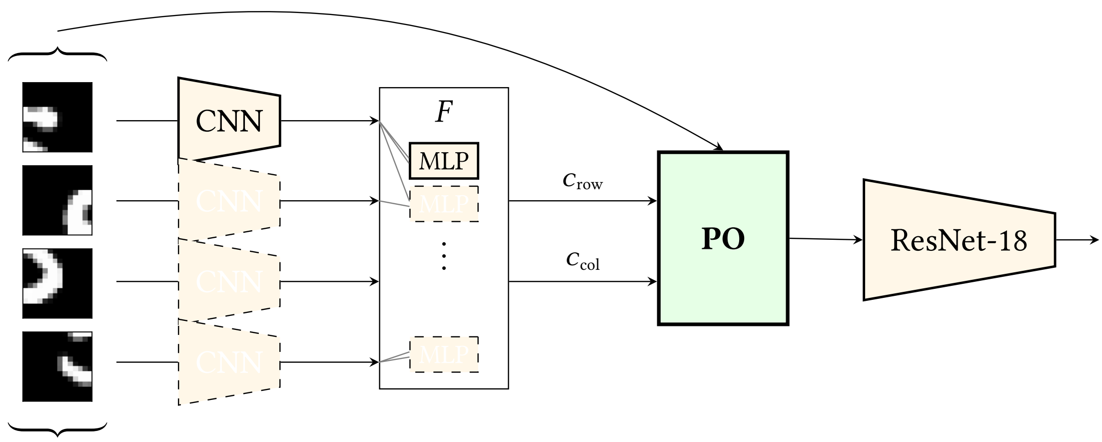


Figure 4.3: Network architecture for image mosaic tasks. The small CNN and the MLP in F is shared across set elements and pairs of set elements respectively. The PO block performs the optimisation with the given row and column costs and permutes the input set. The ResNet-18 network at the end is only present in the implicit permutation setting.

Table 4.1: Mean squared error of image mosaic reconstruction for different datasets and number of tiles an image is split into. Lower is better. LinAssign* is the model by Mena et al. [71], LinAssign is our reproduction of their model, PO-U and PO-LA are our models with uniform and linear assignment initialisation respectively.

	MNIST				CIFAR10				ImageNet 64×64			
Model	2×2	3×3	4×4	5×5	2×2	3×3	4×4	5×5	2×2	3×3	4×4	5×5
*LinAssign**	*0.00*	*0.00*	*0.26*	*0.18*	–	–	–	–	*0.22*	*0.31*	–	–
LinAssign	0.00	0.00	0.33	0.08	0.37	0.49	1.34	1.12	0.60	1.10	1.33	1.44
PO-U	**0.00**	0.02	0.46	0.45	**0.11**	0.44	1.23	1.26	**0.14**	0.69	1.20	**1.31**
PO-LA	0.00	**0.00**	**0.07**	**0.01**	0.18	**0.16**	**1.07**	**0.70**	0.16	**0.62**	**1.13**	1.32

4.4.2 Re-assembling image mosaics

As second task, we consider a problem where the model is given images that are split into $n \times n$ equal-size tiles and the goal is to re-arrange this set of tiles back into the original image. We take these images from either MNIST, CIFAR10, or a version of ImageNet with images resized down to 64×64 pixels. For this task, we use the alternative cost function described in Subsection 4.2.4 to arrange the tiles into a grid rather than a sequence; this lets our model take relations within rows and columns into account. Again, we minimise the mean squared error to the correctly permuted image and use the Hungarian algorithm during evaluation, matching the experimental setup in Mena et al. [71]. Due to the lack of reference implementation of their model for this experiment, we use our own implementation of their model, which we verified to reproduce their MNIST results closely. Unlike them, we decide to not arbitrarily upscale MNIST images to get improved results for all models. We show our model architecture in Figure 4.3.

Table 4.2: Accuracy of image mosaic reconstruction. Higher is better. A permutation is considered correct if all tiles are placed correctly. Because of indistinguishable tiles at higher tile counts (for example multiple completely blank tiles on MNIST) it becomes very unlikely to guess the correct ground-truth matching at higher tile counts. LinAssign* results come from Mena et al. [71].

	MNIST				CIFAR10				ImageNet 64×64			
Model	2×2	3×3	4×4	5×5	2×2	3×3	4×4	5×5	2×2	3×3	4×4	5×5
*LinAssign**	*100*	*72*	*3*	*0*	–	–	–	–	*81*	*47*	*0*	*0*
LinAssign	99.7	66.9	0.8	0.0	68.8	30.3	0.0	0.0	47.7	4.0	0.0	**0.0**
PO-U	**100.0**	65.9	0.2	0.0	86.2	34.4	0.0	0.0	**85.9**	19.2	0.1	0.0
PO-LA	99.9	**73.1**	**1.8**	**0.0**	**87.3**	**66.0**	**0.6**	**0.3**	84.2	**28.6**	**0.1**	0.0

Results The mean squared errors for the different image datasets and different number of tiles an image is split into are shown in Table 4.1. The corresponding accuracies when training a classifier on these reconstructions are shown in Table 4.2. First, notice that in essentially all cases, our model with linear assignment initialisation (PO-LA) performs best, often significantly so. On the two more complex datasets CIFAR10 and ImageNet, this is followed by our PO-U model, then the LinAssign model. We analyse what types of comparisons PO-U learns in Subsection 4.5.2.

On MNIST, LinAssign performs better than PO-U on higher tile counts because images are always centred on the object of interest. That means that many tiles only contain the background and end up completely blank; these tiles can be more easily assigned to the borders of the image by the LinAssign model than our PO-U model because the absolute position is much more important than the relative positioning to other tiles. This also points towards an issue for these cases in our cost function: because two tiles that have the same contents are treated the same by our model, it is unable to place one blank tile on one side of the image and another blank tile on the opposite side, as this would require treating the two tiles differently. This issue with backgrounds is also present on CIFAR10 to a lesser extent: notice how for the 3×3 case, the error of PO-U is much closer to LinAssign on CIFAR10 than on ImageNet, where PO-U is much better comparatively. This shows that the PO-U model is more suitable for more complex images when relative positioning matters more. PO-LA is able to combine the best of both methods.

In Figure 4.4, Figure 4.5, and Figure 4.6, we show some example reconstructions that have been learnt by our PO-U model. Starting from a uniform assignment at the top, the figures show reconstructions as a permutation is being optimised. Generally, it is able to reconstruct most images fairly well.

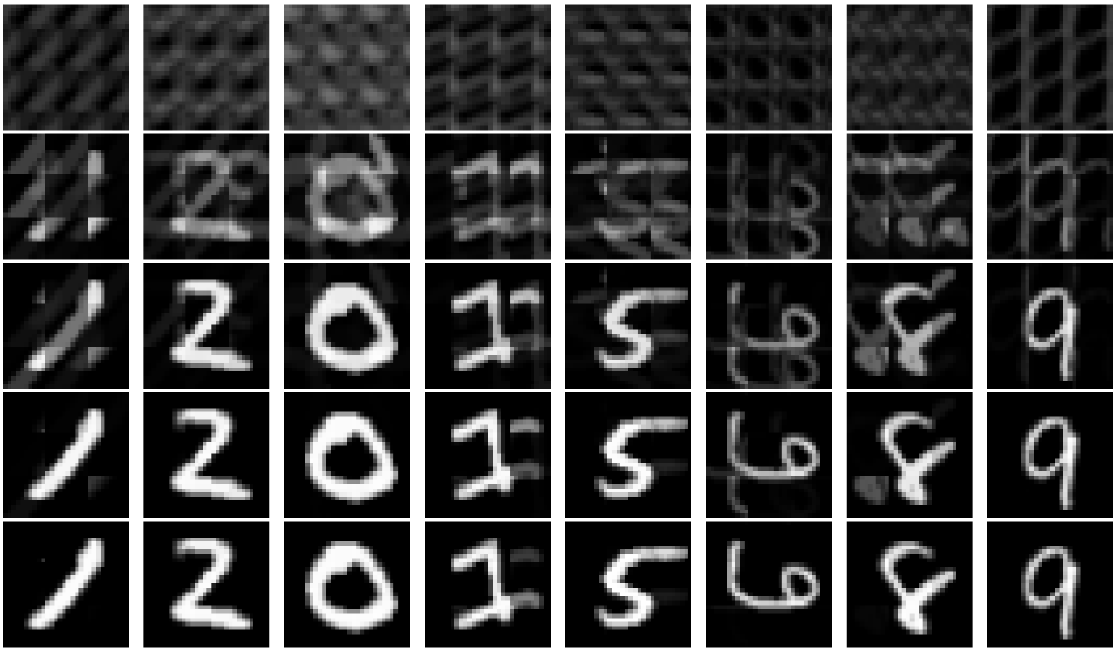


Figure 4.4: Example reconstructions of PO-U as they are being optimised on MNIST 3×3 with explicit supervision. These examples have not been cherry-picked.

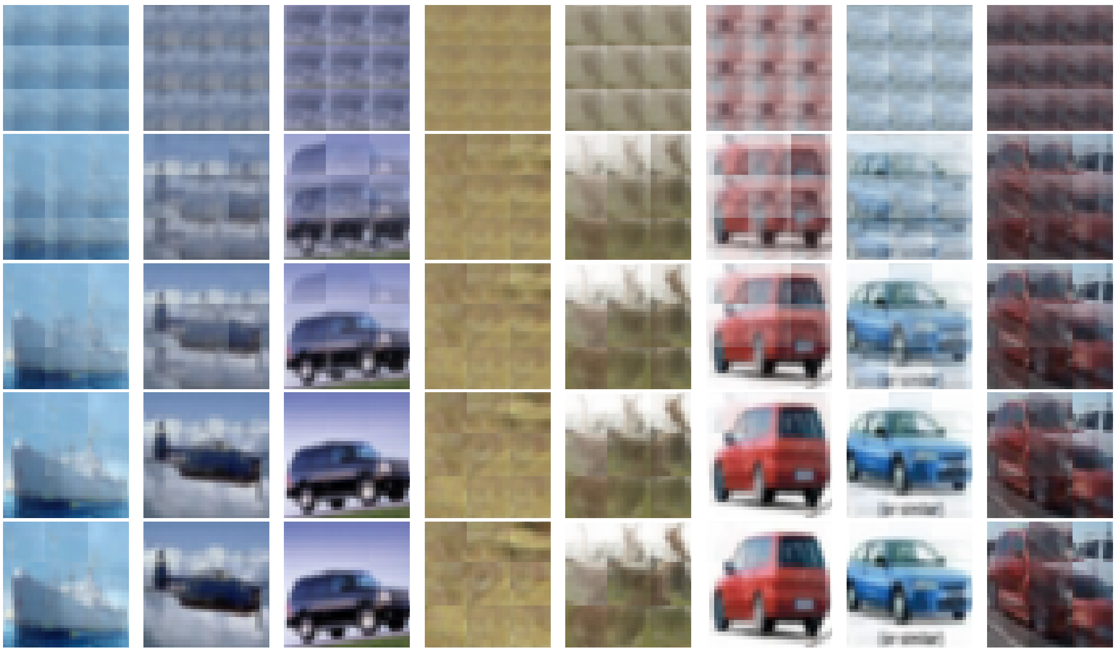


Figure 4.5: Example reconstructions of PO-U as they are being optimised on CIFAR10 3×3 with explicit supervision. These examples have not been cherry-picked.

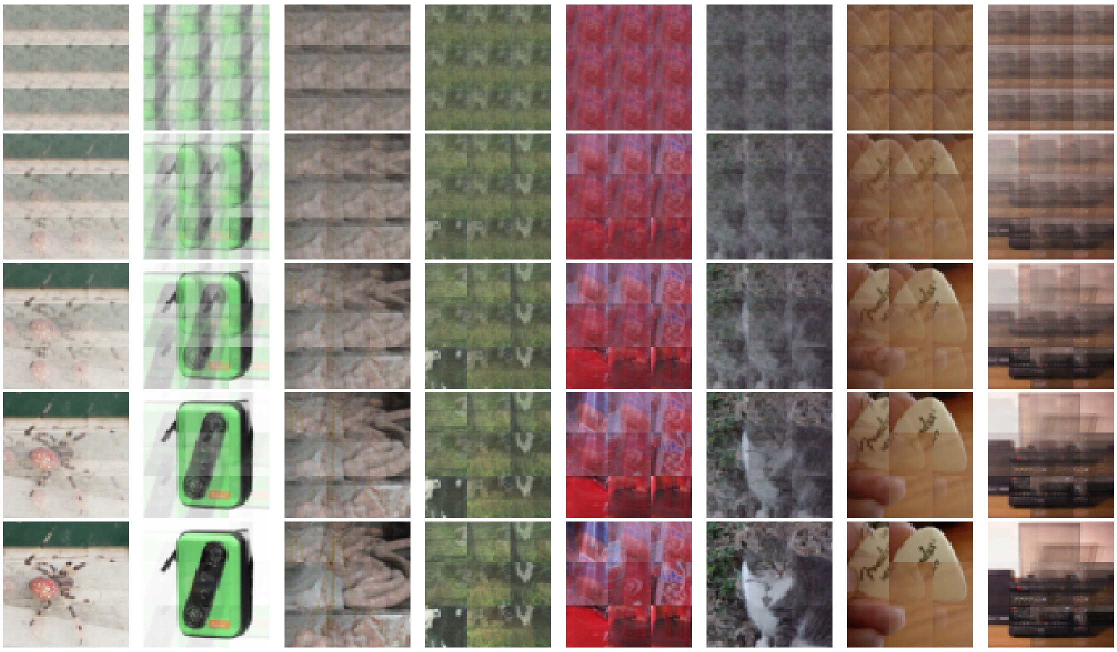


Figure 4.6: Example reconstructions of PO-U as they are being optimised on ImageNet 3×3 with explicit supervision. These examples have not been cherry-picked.

4.4.3 Implicit permutations through classification

We now turn to tasks where the goal is not producing the permutation itself, but learning a suitable permutation for a different task. For these tasks, we do not provide explicit supervision on what the permutation should be; an appropriate permutation is learned implicitly while learning to solve another task.

As the dataset, we use a straightforward modification of the image mosaic task. The image tiles are assigned to positions on a grid as before, which are then concatenated into a full image. This image is fed into a standard image classifier (ResNet-18 [42]) which is trained with the usual cross-entropy loss to classify the image. The idea is that the network has to learn *some* permutation of the image tiles so that the classifier can classify it accurately. This is not necessarily the permutation that restores the original image faithfully.

One issue with this set-up we observed is that with big tiles, it is easy for a CNN to ignore the artefacts on the tile boundaries, which means that simply permuting the tiles randomly gets to almost the same test accuracy as using the original image. To prevent the network from avoiding to solve the task, we first pre-train the CNN on the original dataset without permuting the image tiles. Once it is fully trained, we freeze the weights of this CNN and train only the permutation mechanism.

Results We show the classification results in Table 4.3 and the corresponding MSE reconstruction losses in Table 4.4. Note that the models

Table 4.3: Accuracy of classification from implicitly-learned image reconstructions through permutations. *max* shows the accuracy of the pre-trained model on the original images, *min* shows the accuracy of the pre-trained model on images with randomly permuted tiles.

	MNIST				CIFAR10				ImageNet 64×64			
Model	2×2	3×3	4×4	5×5	2×2	3×3	4×4	5×5	2×2	3×3	4×4	5×5
max	*99.5*	*99.5*	*99.5*	*99.3*	*81.0*	*81.0*	*81.9*	*80.2*	*31.2*	*33.4*	*31.2*	*33.5*
min	*36.6*	*22.5*	*17.1*	*14.6*	*36.5*	*26.4*	*22.9*	*18.0*	*11.4*	*7.8*	*3.5*	*2.7*
LinAssign	**99.4**	99.2	86.0	84.2	64.6	33.8	33.4	**32.5**	13.1	5.8	5.3	3.3
PO-U	99.3	98.7	67.9	69.2	70.8	**41.6**	33.3	29.7	**24.6**	**12.1**	**7.3**	**5.1**
PO-LA	99.3	**99.4**	**93.3**	**89.8**	**71.6**	40.7	**34.2**	32.3	23.4	10.9	6.3	4.4

Table 4.4: Mean squared error of implicitly-learned reconstruction. Lower is better.

	MNIST				CIFAR10				ImageNet 64×64			
Model	2×2	3×3	4×4	5×5	2×2	3×3	4×4	5×5	2×2	3×3	4×4	5×5
max	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
min	*1.59*	*1.63*	*1.91*	*1.72*	*1.76*	*1.91*	*2.11*	*2.01*	*1.48*	*1.72*	*1.79*	*1.83*
LinAssign	0.02	0.05	0.73	1.11	0.56	1.29	1.53	1.62	0.88	1.33	1.32	**1.39**
PO-U	0.02	0.15	1.38	1.24	0.33	1.03	**1.44**	**1.34**	0.44	1.16	**1.28**	1.47
PO-LA	**0.01**	**0.03**	**0.50**	**0.93**	**0.28**	**1.00**	1.47	1.55	**0.41**	**1.15**	1.41	1.43

are not trained to minimise MSE loss. Generally, a similar trend to the image mosaic task with explicit supervision can be seen. Our PO-LA model usually performs best, although for ImageNet PO-U is consistently better. This is evidence that for more complex images, the benefits of linear assignment decrease (and can actually detract from the task in the case of ImageNet) and the importance of the optimisation process in our model increases. With higher number of tiles on MNIST, even though PO-U does not perform well, PO-LA is clearly superior to only using LinAssign. This is again due to the fully black tiles not being able to be sorted well by the cost function with uniform initialisation.

In Figure 4.7, Figure 4.8, and Figure 4.9, we show some example reconstructions that have been learnt by our PO-U model on 3×3 versions of the image datasets. Because the quality of implicit CIFAR10 and ImageNet reconstructions are relatively poor, we also include Figure 4.10, and Figure 4.11 on 2×2 versions. Starting from a uniform assignment at the top, the figures show reconstructions as a permutation is being optimised. The reconstructions here are clearly noisier than before due the supervision only being implicit. This is evidence that while our method is superior to existing methods in terms of reconstruction error and accuracy of the classification, there is still plenty of room for improvement to allow for better implicitly learned permutations. Keep in mind that it is not necessary for the permutation to produce the original image exactly, as long as the CNN can consistently recognise

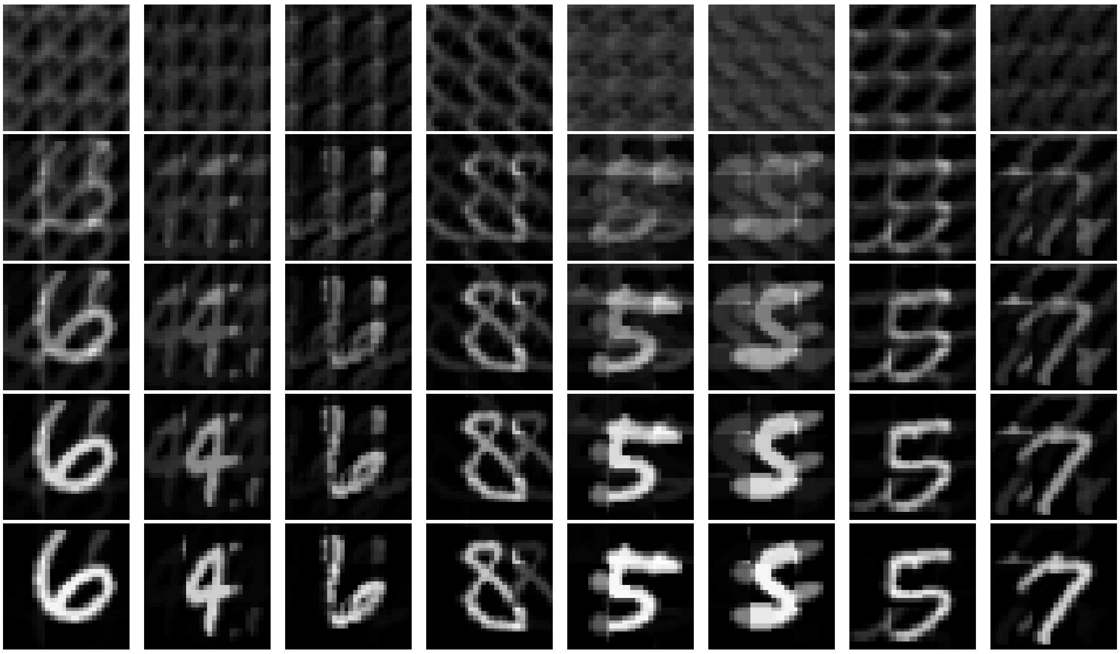


Figure 4.7: Example reconstructions of PO-U as they are being optimised on MNIST 3×3 with implicit supervision. These examples have not been cherry-picked.

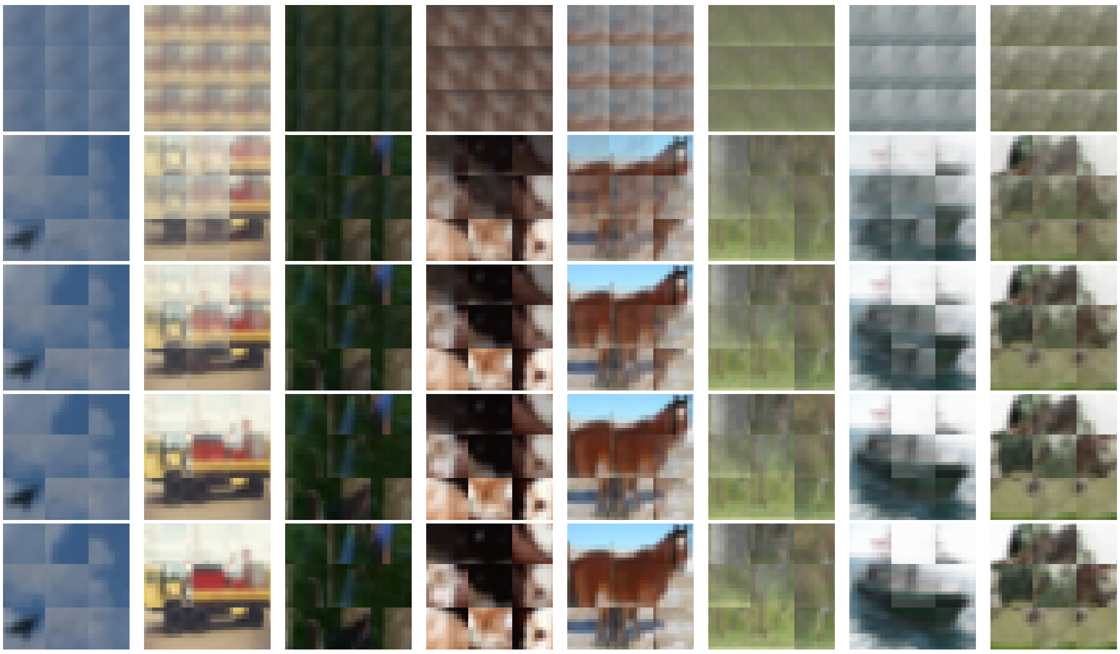


Figure 4.8: Example reconstructions of PO-U as they are being optimised on CIFAR10 3×3 with implicit supervision. These examples have not been cherry-picked.

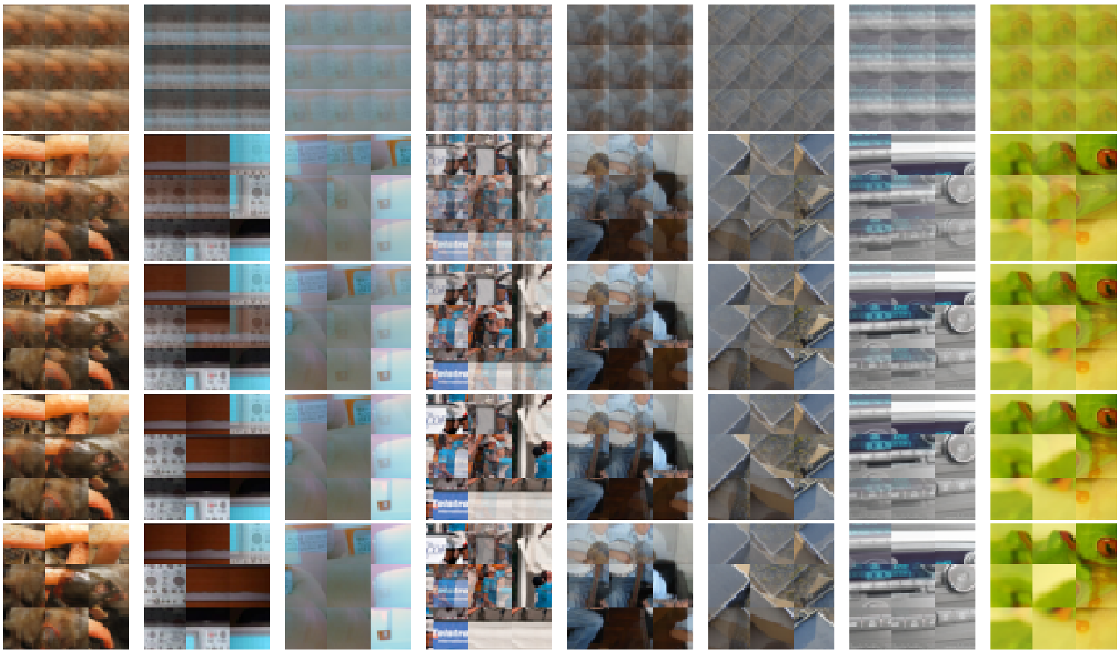


Figure 4.9: Example reconstructions of PO-U as they are being optimised on ImageNet 3×3 with implicit supervision. These examples have not been cherry-picked.

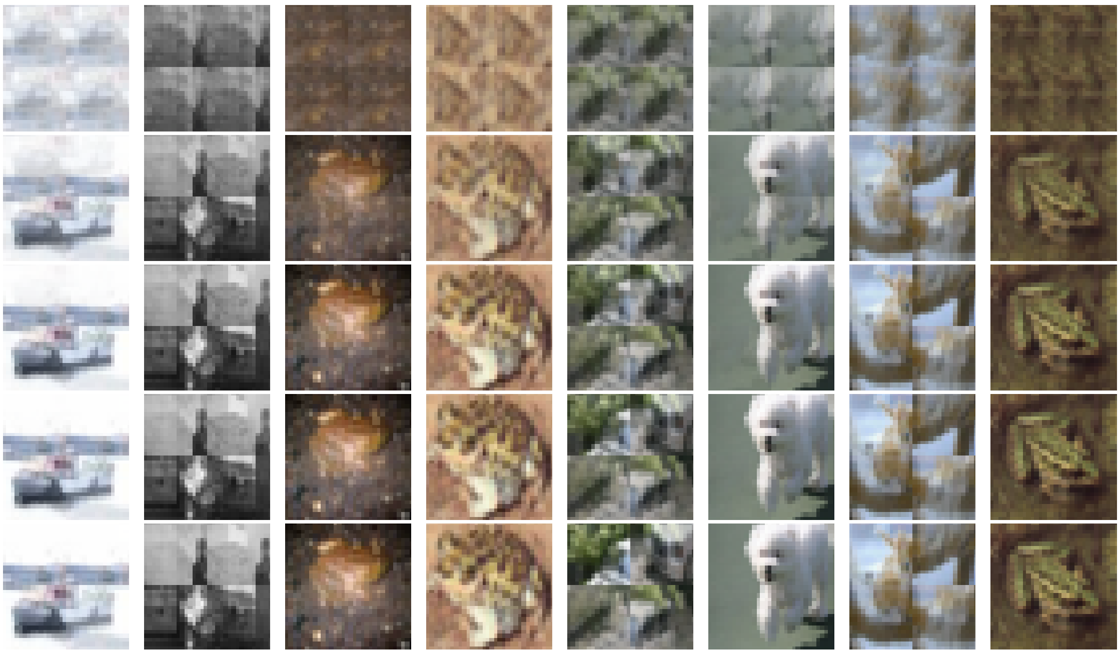


Figure 4.10: Example reconstructions of PO-U as they are being optimised on CIFAR10 2×2 with implicit supervision. These examples have not been cherry-picked.

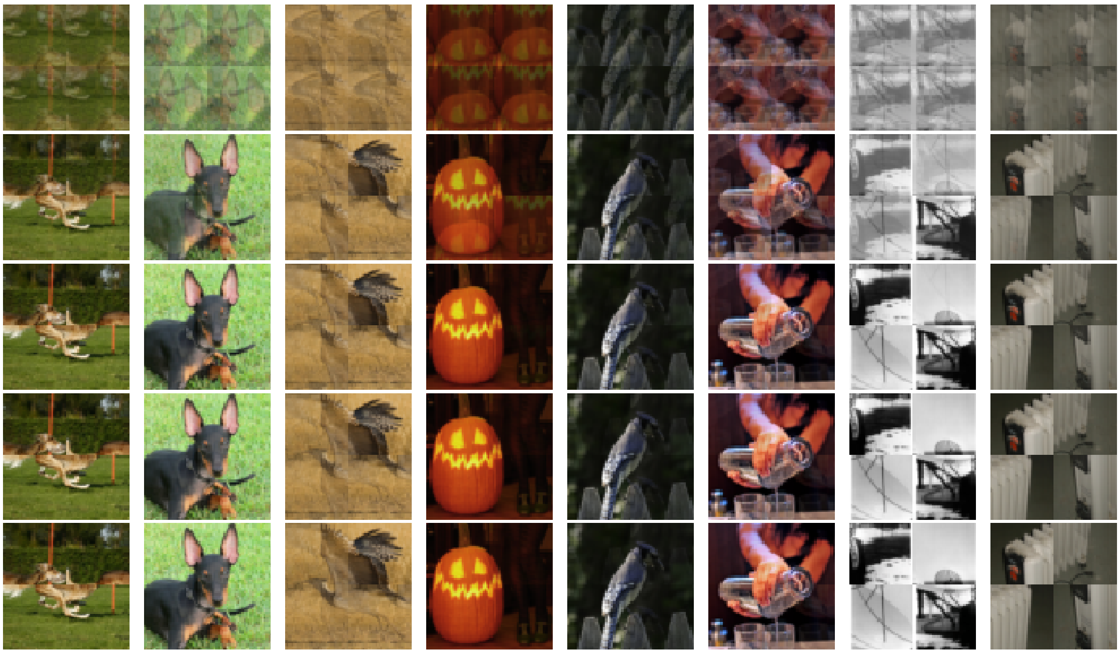


Figure 4.11: Example reconstructions of PO-U as they are being optimised on ImageNet 2×2 with implicit supervision. These examples have not been cherry-picked.

what the permutation method has learned. Our models tend to naturally learn reconstructions that are more similar to the original image than the LinAssign model.

4.4.4 Visual question answering

As the last task, we consider the much more complex problem of visual question answering (VQA): answering questions about images. We use the VQA v2 dataset [6, 38], which in total contains around 1 million questions about 200,000 images from MS-COCO with 6.5 million human-provided answers available for training. We use bottom-up attention features [4] as representation for objects in the image, which for each image gives us a *set* (size varying from 10 to 100 per image) of bounding boxes and the associated feature vector that encodes the contents of the bounding box. These object proposals have no natural ordering a priori.

We use the state-of-the-art BAN model [56] as baseline and perform a straightforward modification to it to incorporate our module (see Figure 4.12). For each element in the set of object proposals, we concatenate the bounding box coordinates, features, and the attention value that the baseline model generates. Our model learns to permute this set into a sequence, which is fed into an LSTM. We take the last cell state of the LSTM to be the representation of the set, which is fed back into the baseline model. This is done for each of the eight attention glimpses in the BAN model. We include another baseline model (BAN + LSTM)

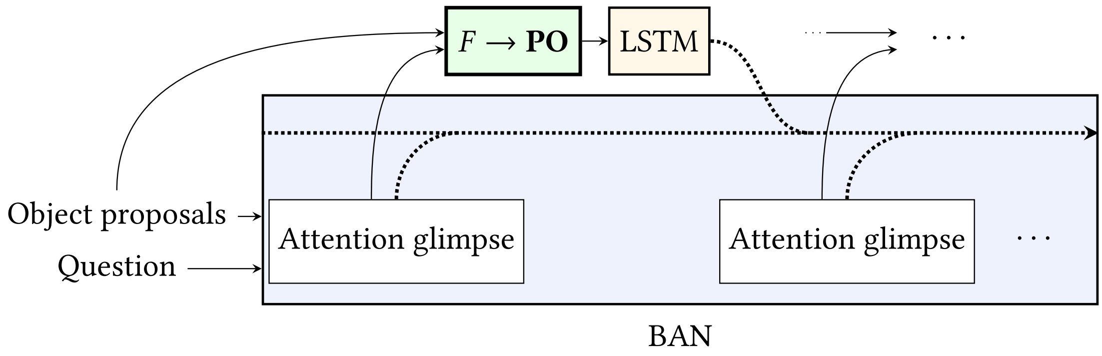


Figure 4.12: Network architecture for visual question answering task using BAN with 8 glimpses (2 shown) as baseline. We add a shared PO-U module with an LSTM to process its output for each glimpse. The outputs of the BAN attention and the LSTM are added into the hidden state of the BAN network.

Table 4.5: Accuracy on VQA v2 validation set, mean of 10 runs. The sample standard deviation over these runs is shown after the ± symbol. Overall includes the other three question categories.

Model	Overall	Yes/No	Number	Other
BAN	65.96 ± 0.16	83.34 ± 0.09	49.24 ± 0.56	57.17 ± 0.14
BAN + LSTM	66.06 ± 0.13	83.29 ± 0.13	49.64 ± 0.37	57.30 ± 0.13
BAN + PO-U	**66.33** ± 0.09	**83.50** ± 0.10	**50.42** ± 0.46	**57.48** ± 0.10

that skips the permutation learning, directly processing the set with the LSTM.

Our results on the validation set of VQA v2 are shown in Table 4.5. We improve on the overall performance of the state-of-the-art model by 0.37% – a significant improvement for this dataset – with 0.27% of this improvement coming from the learned permutation. This shows that there is a substantial benefit to learning an appropriate permutation through our model in order to learn better set representations. Our model significantly improves on the number category, despite the inclusion of our counting module from Chapter 3 specifically targeted at number questions in the baseline. This is evidence that the representation learned through the permutation is non-trivial. Note that the improvement through our model is not simply due to increased model size and computation: Kim et al. [56] found that significantly increasing BAN model size, increasing computation time similar in scale to including our model, does not yield any further gains.

4.5 Analysis of learned comparisons

4.5.1 Number sorting

First, we investigate what comparison function F is learned for the number sorting task. We start with plotting the outputs of F for different

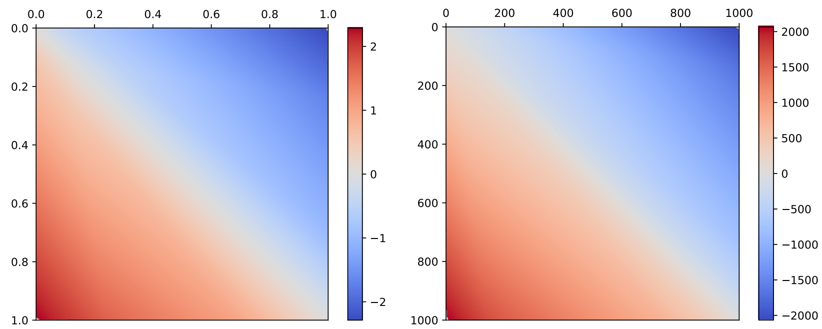


Figure 4.13: Outputs of F for different pairs of numbers as input. Red indicates that the number on the left should be ordered after the number at the top, blue indicates the opposite. Evaluation intervals are $[0, 1]$ (left) and $[0, 1000]$ (right).

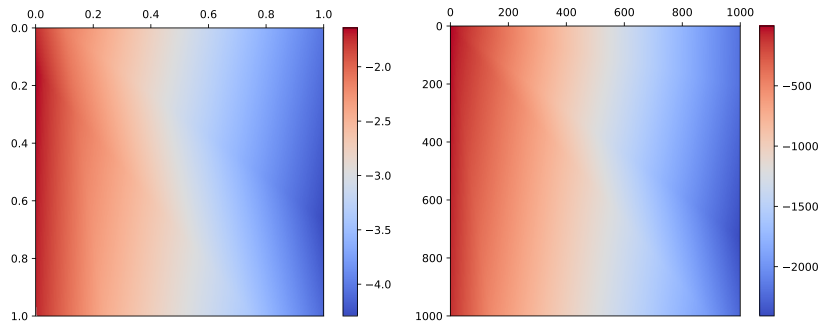


Figure 4.14: Outputs of f for different pairs of numbers as input. Evaluation intervals are $[0, 1]$ (left) and $[0, 1000]$ (right).

pairs of inputs in Figure 4.13. From this, we can see that it learns a sensible comparison function where it outputs a negative number when the first argument is lower than the second, and a positive number vice versa.

The easiest way to achieve this is to learn $f(x_i, x_j) = x_i$, which results in $F(x_i, x_j) = x_i - x_j$. By plotting the outputs of the learned f in Figure 4.14 we can see that something close to this has indeed been learned. The learned f mostly depends on the second argument and is a scaled and shifted version of it. It has not learned to completely ignore the first argument, but the deviations from it are small enough that the cost function of the permutation is able to compensate for it. We can see that there is a faint grey diagonal area going from $(0, 0)$ to $(1, 1)$ and to $(1000, 1000)$, which could be an artefact from F having small gradients due to its skew-symmetry when two numbers are close to each other.

4.5.2 Image mosaics

Next, we investigate the behaviour of F on the image mosaic task. Since our model uses the outputs of F in the optimisation process, we find it easier to interpret F over f in the subsequent analysis.

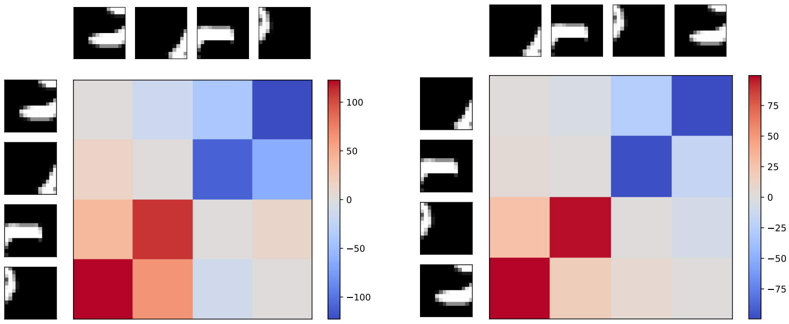


Figure 4.15: Outputs of F_1 (left half, row comparisons) and F_2 (right half, column comparisons) for pairs of tiles from an image in MNIST. For F_1, the tiles are sorted left-to-right if only F_1 was used as cost. For F_2, the tiles are sorted top-to-bottom if only F_2 was used as cost. Blue indicates that the tile to the left of this entry should be ordered left of the tile at the top for F_1, the tile on the left should be ordered above the tile at the top for F_2. The opposite applies for red. The saturation of the colour indicates how strong this ordering is.

Outputs

We start by looking at the output of F_1 (costs for left-to-right ordering) and F_2 (costs for top-to-bottom ordering) for MNIST 2×2, shown in Figure 4.15. First, there is a clear entry in each row and column of both F_1 and F_2 that has the highest absolute cost (high colour saturation) whenever the corresponding tiles fit together correctly. This shows that it successfully learned to be confident what order two tiles should be in when they fit together. From the two 2-by-2 blocks of red and blue on the anti-diagonal, we can also see that it has learned that for the per-row comparisons (F_1), the tiles that should go into the left column should generally compare to less than (i.e. should be permuted to be to the left of) the tiles that go to the right. Similarly, for the per-column comparisons (F_2) tiles that should be at the top compare to less than tiles that should be at the bottom. Lastly, F_1 has a low absolute cost when comparing two tiles that belong in the same column. These are the entries in the matrix at the coordinates (1, 2), (2, 1), (4, 3), and (3, 4). This makes sense, as F_1 is concerned with whether one tile should be to the left or right of another, so tiles that belong in the same column should not have a preference either way. A similar thing applies to F_2 for tiles that belong in the same row.

Sensitivity to positions

Next, we investigate what positions within the tiles F_1 and F_2 are most sensitive to. This illustrates what areas of the tiles are usually important for making comparisons. We do this by computing the gradients of the absolute values of F with respect to the input tiles and averaging over many inputs. For MNIST 2×2 (Figure 4.16, left), it learns no particular spatial pattern for F_1 and puts slightly more focus away from

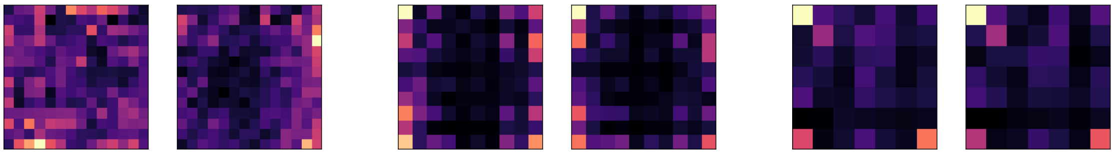


Figure 4.16: Sensitivity to positions within a tile for MNIST 2×2 (left), 3×3 (middle), and 4×4 (right). The left plot of each pair shows F_1, the right plot shows F_2.

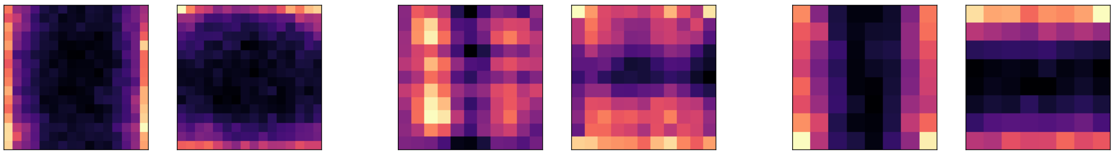


Figure 4.17: Sensitivity to positions within a tile of the comparisons for CIFAR10 2×2 (left), 3×3 (middle), and 4×4 (right). The left plot of each pair shows F_1, the right plot shows F_2.

the centre of the tile for F_2. As we will see later, it learns something that is very content-dependent rather than spatially-dependent. With increasing numbers of tiles on MNIST, it tends to focus more on edges, and especially on corners. For the CIFAR10 dataset (Figure 4.17), there is a much clearer distinction between left-right comparisons for F_1 and top-bottom comparisons for F_2. For the 2×2 and 4×4 settings, it relies heavily on the pixels on the left and right borders for left-to-right comparisons, and top and bottom edges for top-to-bottom comparisons. Interestingly, F_1 in the 3×3 setting (middle pair) on CIFAR10 focuses on the left and right halves of the tiles, but specifically avoids the borders. A similar thing applies to F_2, where a greater significance is given to pixels closer to the middle of the image rather than only focusing on the edges. This suggests that it learns to not only match up edges as with the other tile numbers, but also uses the content within the tile to do more sophisticated content-based comparisons.

Per-tile gradients

Lastly, we can look at the gradients of F with respect to the input tiles for specific pairs of tiles, shown in Figure 4.18 and Figure 4.19. This gives us a better insight into what changes to the input tiles would affect the cost of the comparison the most. These figures can be understood as follows: for each pair of tiles, we have the corresponding two gradient maps next to them. Brightening the pixels for the blue entries in these gradient maps would order the corresponding tile more strongly towards the left for F_1 and towards the top for F_2. The opposite applies to brightening the pixels with red entries. Vice versa, darkening pixels with blue entries orders the tile more strongly towards the right for F_1 and the bottom for F_2. More saturated colours in the gradient maps correspond to greater effects on the cost when changing those pixels.

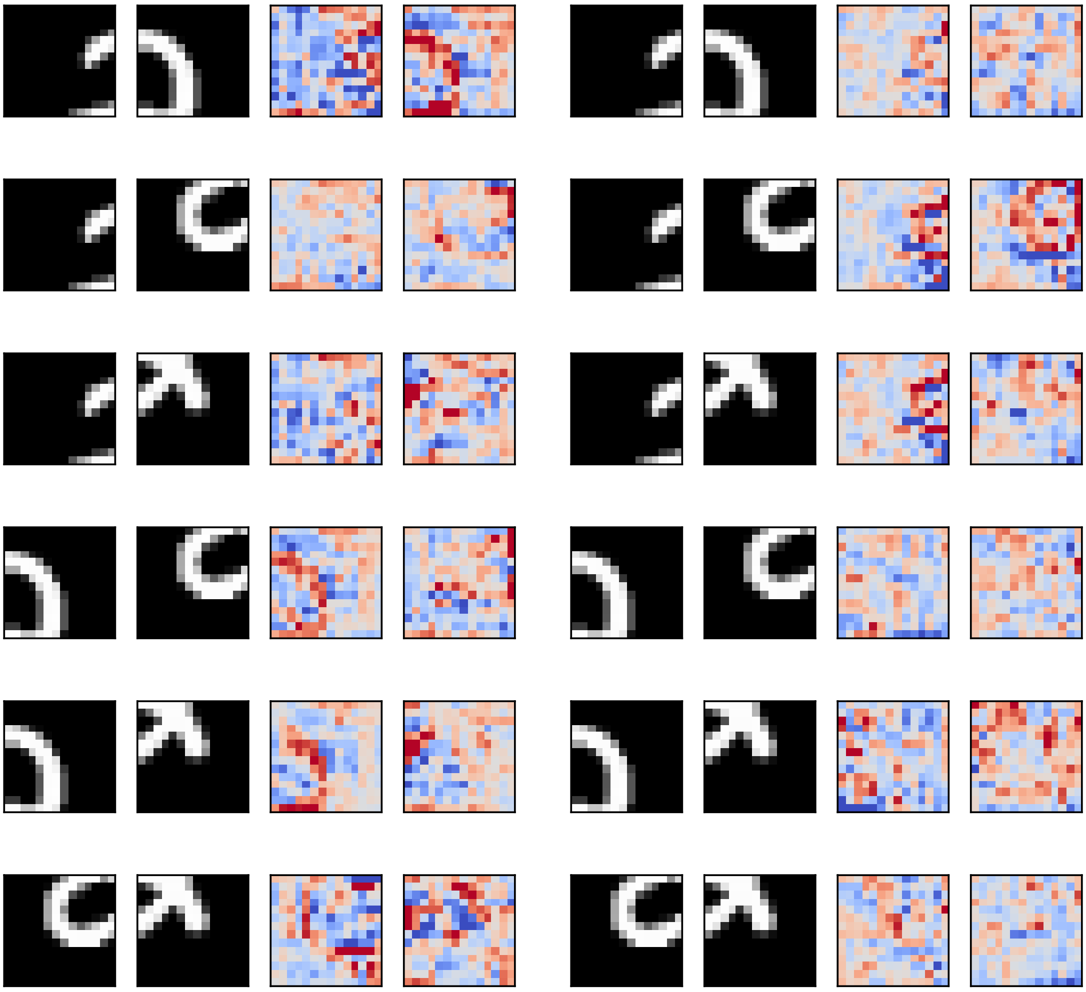


Figure 4.18: Gradient maps of pairs of tiles from MNIST for F_1 (left half) and F_2 (right half). Each group of four consists of: tile 1, tile 2, gradient of $F(t_1, t_2)$ with respect to tile 1, gradient of $F(t_2, t_1)$ with respect to tile 2.

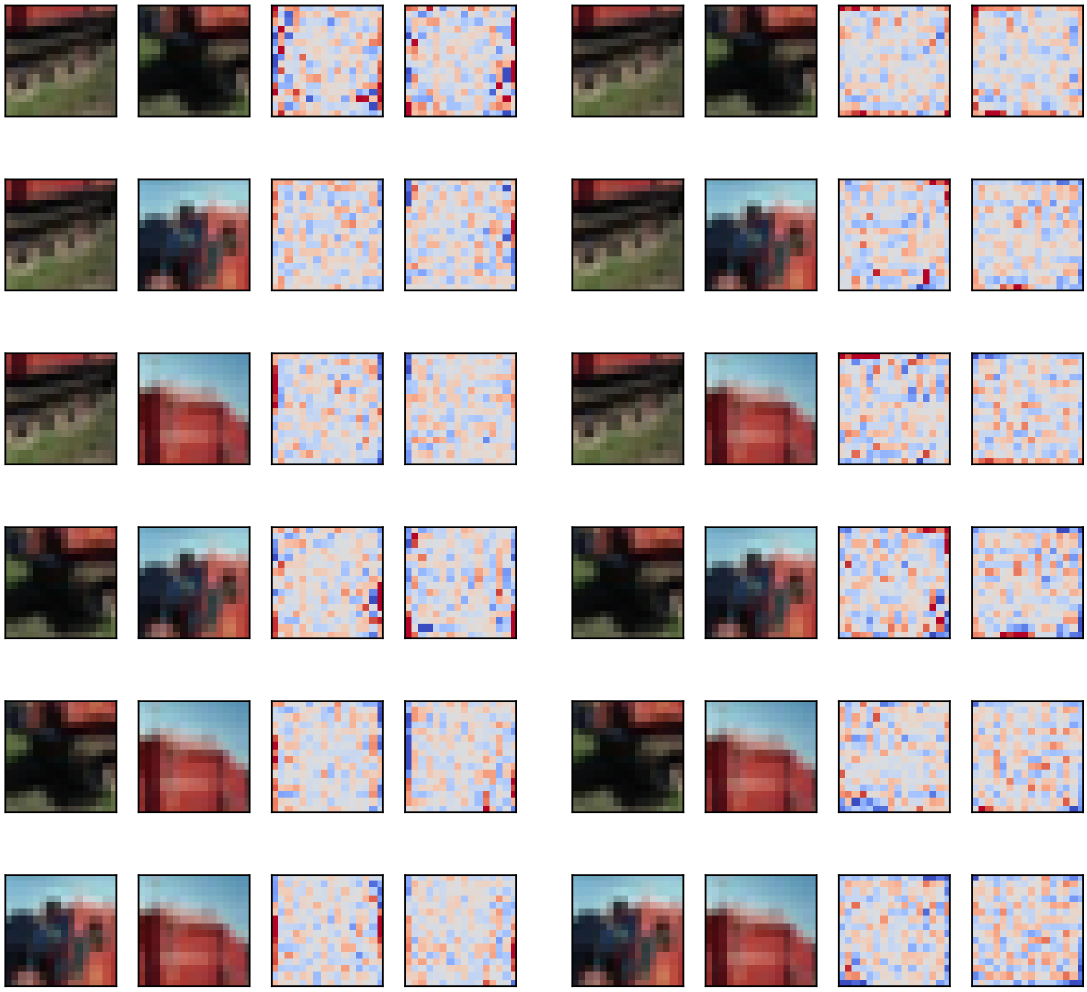


Figure 4.19: Gradient maps of pairs of tiles from CIFAR10 for F_1 (left half) and F_2 (right half). Each group of four consists of: tile 1, tile 2, gradient of $F(t_1, t_2)$ with respect to tile 1, gradient of $F(t_2, t_1)$ with respect to tile 2.

We start with gradients on the tiles for an input showing the digit 2 on MNIST 2×2 in Figure 4.18. We focus on the first row, left side, which shows a particular pair of tiles from this image and their gradients of F_1 (left-to-right ordering), and we share some of our observations here:

- The gradients of the second tile show that to encourage the permutation to place it to the right of the first tile, it is best to increase the brightness of the curve in tile 2 that is already white (red entries in tile 2 gradient map) and decrease the black pixels around it (blue entries). This means that it recognised that this type of curve is important in determining that it should be placed to the right, perhaps because it matches up with the start of the curve from tile 1. We can imagine the curve in the gradient map of tile 2 roughly forming part of a 7 rather than a 2 as well, so it is not necessarily looking for the curve of a 2 specifically.

- In the gradient map of the first tile, we can see that to encourage it to be placed to the left of tile 2, increasing the blue entries would form a curve that would make the first tile look like part of an 8 rather than a 2, completing the other half of the curve from tile 2. This means that it has learned that to match something with the shape in tile 2, a loop that completes it is best, but the partial loop that we have in tile 1 satisfies part of this too.

- Notice how the gradient of tile 1 changes quite a bit when going from row 1 to row 3, where it is paired up with different tiles. This suggests that the comparison has learned something about the specific comparison between tiles being made, rather than learning a general trend of where the tile should go. The latter is what a linear assignment model is limited to doing because it does not model pairwise interactions.

- In the third row, we can see that even though the two tiles do not match up, there is a red blob on the left side of the tile 2 gradient map. This blob would connect to the top part of the line in tile 1, so it makes sense that making the two tiles match up more on the border would encourage tile 2 to be ordered to the right of tile 1.

Similar observations apply to the right half of Figure 4.18, such as row 5, where tile 1 (which should go above tile 2) should have its pixels in the bottom left increased and tile 2 should have its pixels in the top left increased in order for tile 1 to be ordered before (i.e. above) tile 2 more strongly.

On CIFAR10 2×2 in Figure 4.19, it is enough to focus on the borders of the tiles. Here, it is striking how specifically it tries to match edge colours between tiles. For example, consider the blue sky in the left half (F_1), row 6. To order tile 1 to the left of tile 2, we should change tile 1 to have brighter sky and darker red on the right border, and also darken the black on the left border so that it matches up less well with the right border of tile 2, where more of the bright sky is visible. For tile 2, the gradient shows that it should also match up more on the left border, and have increase the amount of bright pixels, i.e. sky, on the right border, again so that it matches up less well with the left border of tile 1 if they were to be ordered the opposing way.

4.6 Discussion

In this chapter, we proposed our Permutation-optimisation module for learning permutations of sets using an optimisation-based approach. In various experiments, we verified the merit of our approach for learning permutations and, from them, set representations. We think that the optimisation-based approach to processing sets is currently underappreciated and hope that the techniques and results in this chapter will inspire new algorithms for processing sets in a permutation-invariant manner. Of course, there is plenty of work to be done. For example, we have only explored one possible function for the total cost; different functions capturing different properties may be used. The main drawback of our approach is the cubic time complexity in the set size compared to the quadratic complexity of Mena et al. [71], which limits our model to tasks where the number of elements is relatively small. While this is acceptable on the real-world dataset that we used – VQA with up to 100 object proposals per image – with only a 30% increase in computation time, our method does not scale to the much larger set sizes encountered in domains such as point cloud classification. Improvements in the optimisation algorithm may improve this situation, perhaps through a divide-and-conquer approach.

We believe that going beyond tensors as basic data structures is important for enabling higher-level reasoning. As a fundamental mathematical object, sets are a natural step forward from tensors for modelling unordered collections. The property of permutation invariance lends itself to greater abstraction by allowing data that has no obvious ordering to be processed, and we took a step towards this by learning an ordering that existing neural networks are able to take advantage of.

Chapter 5

Set auto-encoder: Featurewise sort pooling

In this chapter, we will present a multitude of contributions to the set neural network literature. We identify the responsibility problem in existing set prediction models. We then develop a set encoder that is simpler and faster than our model in Chapter 4. It is based on a similar idea of turning the set into an ordered representation; we use numerical sorting instead of trying to learn the ordering. To avoid the responsibility problem, we develop a set decoder that is paired with our set encoder.

These contributions have been presented as [116] at the Sets & Partitions workshop, hosted at the Neural Information Processing Systems (NeurIPS) 2019 conference, and have been submitted to the International Conference on Learning Representations (ICLR) 2020.

5.1 Introduction

Consider the following task: you have a dataset wherein each data point is a *set* of 2-d points that form the vertices of a regular polygon, and the goal is to learn an auto-encoder on this dataset. The only variable is the rotation of this polygon around the origin, with the number of points, size, and centre of it fixed. Because the inputs and outputs are sets, this problem has some unique challenges.

Encoder: This turns the set of points into a latent space. The order of the elements in the set is irrelevant, so the feature vector the encoder produces should be invariant to permutations of the elements in the set. While there has been recent progress on learning such functions [112, 84], they compress 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 [76, 117].

Decoder: This turns the latent space back into a set. The elements in the target set have an arbitrary order, so a standard reconstruction loss cannot be used naïvely – the decoder would have to somehow output the elements in the same arbitrary order. Methods like those in Achlioptas et al. [1] therefore use an assignment mechanism to match up elements (Section 5.2), after which a usual reconstruction loss can be computed. Surprisingly, their model is still unable to solve the polygon reconstruction task with close-to-zero reconstruction error, despite the apparent simplicity of the dataset.

In this chapter, we introduce a set pooling method for neural networks that addresses both the encoding bottleneck issue and the decoding failure issue. We make the following contributions:

1. We identify the *responsibility problem* (Section 5.3). This is a fundamental issue with existing set prediction models that has not been considered in the literature before, explaining why these models struggle to model even the simple polygon dataset.

2. We introduce FSPool: a differentiable, sorting-based pooling method for variable-size sets (Section 5.4). By using our pooling in the encoder of a set auto-encoder and *inverting the sorting* in the decoder, we can train it with the usual MSE loss for reconstruction *without* the need for an assignment-based loss. This avoids the responsibility problem.

3. We show that our auto-encoder can learn polygon reconstructions with close-to-zero error, which is not possible with existing set auto-encoders (Subsection 5.6.1). This benefit transfers over to a set version of MNIST, where the quality of reconstruction and learned representation is improved (Subsection 5.6.2). In further classification experiments on CLEVR (Subsection 5.6.4) and several graph classification datasets (Subsection 5.6.5), using FSPool in a set encoder improves over many non-trivial baselines.

5.2 Background

The problem with predicting sets is that the output order of the elements is arbitrary, so computing an elementwise mean squared error does not make sense; there is no guarantee that the elements in the target set happen to be in the same order as they were generated. The existing solution around this problem is an assignment-based loss, which assigns each predicted element to its "closest" neighbour in the target set first, after which a traditional pairwise loss can be computed.

We have a predicted set $\hat{Y}$ with feature vectors as elements and a ground-truth set Y, and we want to measure how different the two sets are. These sets can be represented as matrices with the feature vectors placed in the columns in some arbitrary order, so $\hat{Y} = [\hat{\boldsymbol{y}}_1, \ldots, \hat{\boldsymbol{y}}_n]$ and $Y = [\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n]$ with n as the set size (columns) and d as the number of features per element (rows). In this work, we assume that these two sets have the same size. The usual way to produce $\hat{Y}$ is with a multi-layer perceptron (MLP) that has $d \times n$ outputs.

Hungarian loss One way to do this assignment is to find a linear assignment that minimises the total loss, which can be solved with the Hungarian algorithm in $O(n^3)$ time. With Π as the space of all n-length permutations:

$$\mathcal{L}_H(\hat{Y}, Y) = \min_{\pi \in \Pi} \sum_i^n \left\| \hat{\boldsymbol{y}}_i - \boldsymbol{y}_{\pi(i)} \right\|^2 \tag{5.1}$$

Chamfer loss Alternatively, we can assign each element directly to the closest element in the target set. To ensure that all points in the target set are covered, a term is added to the loss wherein each element in the target set is also assigned to the closest element in the predicted set. This has $O(n^2)$ time complexity and can be run efficiently on GPUs.

$$\mathcal{L}_C(\hat{Y}, Y) = \sum_i \min_j \left\| \hat{\boldsymbol{y}}_i - \boldsymbol{y}_j \right\|^2 + \sum_j \min_i \left\| \hat{\boldsymbol{y}}_i - \boldsymbol{y}_j \right\|^2 \tag{5.2}$$

Both of these losses are examples of permutation-invariant functions: the loss is the same regardless of how the columns of Y and $\hat{Y}$ are permuted.

5.3 Responsibility problem

It turns out that standard neural networks struggle with modeling symmetries that arise because there are $n!$ different list representations of the same set, which we highlight here with an example. Suppose we want to train an auto-encoder on our polygon dataset and have a square (so a set of 4 points with the x-y coordinates as features) with some arbitrary initial rotation (see Figure 5.2). Each pair in the 8 outputs of the MLP decoder is *responsible* for producing one of the points in this square (Figure 5.1). We mark each such pair with a different colour in the figure.

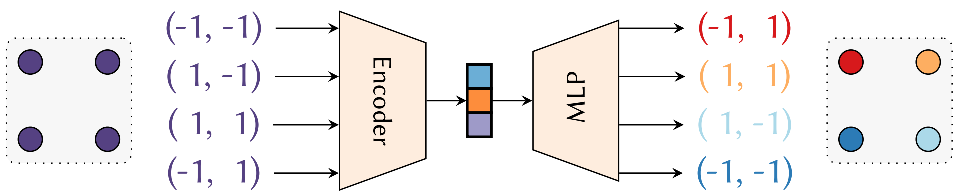


Figure 5.1: Each pair of outputs of the auto-encoder is *responsible* for one of the points of the set. This responsibility is marked with the different colours of the outputs.

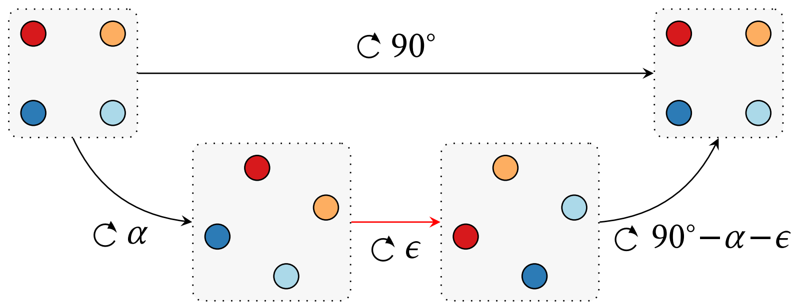


Figure 5.2: Discontinuity (red arrow) when rotating the set of points. The coloured points denote which output of the network is responsible for which point. In the top path, the set rotated by 90° is the same set (exactly the same shape before and after rotation) and encodes to the same feature vector, so the output responsibility (colouring) must be the same too. In this example, after 30° and a further small clockwise rotation by ϵ, the point that each output pair is responsible for has to suddenly change.

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 decode to 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 (bottom path in figure), *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 rotation 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. *Even though we change the set continuously, the list representation (MLP or RNN 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 increase the number of vertices in the polygon (number of set elements), it must learn an increasing frequency of situations where all the outputs must discontinuously change at once, which becomes very difficult to model. Our experiment in Subsection 5.6.1 confirms this.

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 set of 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 (like in Rezatofighi et al. [87] and Stewart et al. [96]) must handle which of its outputs is responsible for what element in a discontinuous way. Note that traditional object detectors like Faster R-CNN do not have this responsibility problem, because they do not treat object detection as a proper set prediction task with their anchor-based approach. When the set elements come from a finite domain (often a set of labels) and not $\mathbb{R}^d$, it does not make sense to interpolate set elements. Thus, the responsibility problem does not apply to methods for such problems, for example Welleck et al. [104] and Hamid Rezatofighi et al. [41].

5.3.1 Formal statement

The following theorem is a more formal treatment of the responsibility problem resulting in discontinuities.

Theorem 1 *For any set function $f : \mathcal{S}_n^d \to \mathbb{R}^{d\times n}$ ($d \geq 2$, $n \geq 2$, $\mathcal{S}_n^d$ is the set of all sets of size n with elements in $\mathbb{R}^d$) from a set of points $\boldsymbol{S} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ to a list representation of that set $\boldsymbol{L} = [\boldsymbol{x}_{\sigma(1)}, \boldsymbol{x}_{\sigma(2)}, \ldots, \boldsymbol{x}_{\sigma(n)}]$ with some fixed permutation $\sigma \in \Pi$, there will be a discontinuity in f: there exists an $\varepsilon > 0$ such that for all $\delta > 0$, there exist two sets $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ where:*

$$d_s(\boldsymbol{S}_1, \boldsymbol{S}_2) < \delta \quad \textit{and} \quad d_l(f(\boldsymbol{S}_1), f(\boldsymbol{S}_2)) \geq \varepsilon. \tag{5.3}$$

d_s is a measure of the distance between two sets (e.g. Chamfer loss) and d_l is the sum of Euclidean distances ($d_l(\boldsymbol{A}, \boldsymbol{B}) = \sum_j \left\|\boldsymbol{a}_j - \boldsymbol{b}_j\right\|_2$).

Proof. We prove the theorem by considering mappings from a set of two points in two dimensions. For larger sets or sets with more dimensions, we can isolate two points and two dimensions and ignore the remaining points and dimensions.

Let us consider the set of two points $\boldsymbol{S}(\theta) = \left\{\begin{bmatrix} -\cos(\theta) \\ -\sin(\theta) \end{bmatrix}, \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}\right\}$ (see Figure 5.3). This is mapped to a list $\boldsymbol{L}(\theta) = f(\boldsymbol{S}(\theta))$. Without loss of generality, we can assume that our list representation for $\theta = 0$ is $\boldsymbol{L}(0) = \begin{bmatrix} -\cos(0) & \cos(0) \\ -\sin(0) & \sin(0) \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix}$. Since the order of set elements is irrelevant and f is a (permutation-invariant) set function, $\boldsymbol{S}(\pi) = \boldsymbol{S}(0)$ and therefore $\boldsymbol{L}(\pi) = \boldsymbol{L}(0) = \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix}$. This implies that for at least one value of $\theta = \theta^*$, there is a change in responsibility such that for $\theta \leq \theta^*$, the list representation will be $\boldsymbol{L}_1(\theta) = \begin{bmatrix} -\cos(\theta) & \cos(\theta) \\ -\sin(\theta) & \sin(\theta) \end{bmatrix}$ while for $\theta > \theta^*$, the list representation will be $\boldsymbol{L}_2(\theta) = \begin{bmatrix} \cos(\theta) & -\cos(\theta) \\ \sin(\theta) & -\sin(\theta) \end{bmatrix}$ in order to satisfy $\boldsymbol{L}(\pi) = \boldsymbol{L}(0)$. For any θ, $d_l(\boldsymbol{L}_1(\theta), \boldsymbol{L}_2(\theta)) = 4$.

Let $\varepsilon = 3.9$ and δ be given. We can find a sufficiently small $\alpha > 0$ so that $d_s(\boldsymbol{S}(\theta^*), \boldsymbol{S}(\theta^* + \alpha)) < \delta$ and $d_l(\boldsymbol{L}(\theta^*), \boldsymbol{L}(\theta^* + \alpha)) > \varepsilon$. □

5.4 Featurewise sort pooling

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 5.2, because rotating the square by 90° also permutes the outputs of 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. This also means that $\boldsymbol{L}(\pi) \neq \boldsymbol{L}(0)$, so the proof no longer applies.

We describe the model for the simplest case of encoding fixed-size sets in Subsection 5.4.1, extend it to variable-sized sets in Subsection 5.4.2, then discuss how to use this in an auto-encoder in Subsection 5.4.3.

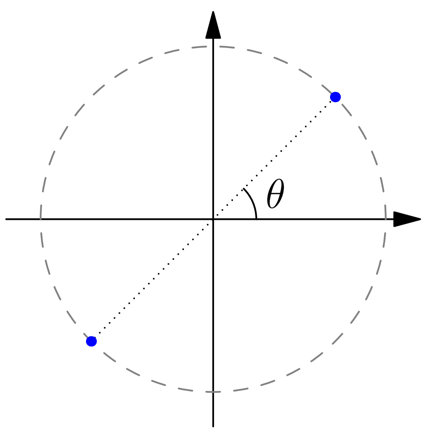


Figure 5.3: Example of the set containing two points.

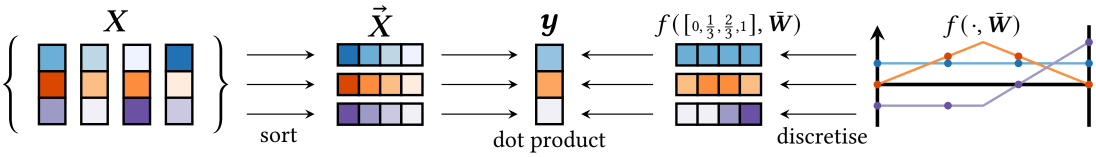


Figure 5.4: Overview of our FSPool model for variable-sized sets. In this example, the weights define piecewise linear functions with two pieces. The four dots on each line correspond to the positions where f is evaluated for a set of size four.

5.4.1 Fixed-size sets

We are given a set of n feature vectors $\boldsymbol{X} = [\boldsymbol{x}_1, \dots, \boldsymbol{x}_n]$ 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 $\vec{\boldsymbol{X}}$:

$$\vec{X}_{i,j} = \textsc{Sort}(\boldsymbol{X}_{i,:})_j \tag{5.4}$$

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 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. Also, if we were to sort whole elements by one feature, there would be discontinuities whenever two elements swap order. This problem is avoided by our featurewise sorting.

Efficient parallel implementations of Sort are available in Deep Learning frameworks such as PyTorch, which uses a bitonic sort ($O(\log^2 n)$ parallel time, $O(n \log^2 n)$ comparisons). While the permutation that the sorting applies is not differentiable, gradients can still be propagated pathwise according to this permutation in a similar way as for max pooling.

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).

$$y_i = \sum_j^n W_{i,j} \vec{X}_{i,j} \tag{5.5}$$

$\boldsymbol{y} \in \mathbb{R}^d$ 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 [47]. This is a generalisation of both max and sum pooling, since max pooling can be obtained with the weight vector $[1, 0, \dots, 0]$ and sum pooling can be obtained with the $\mathbf{1}$ vector. Thus, it is also a maximally powerful pooling method for multisets [105] while being potentially more flexible [76] in what it can represent.

5.4.2 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 $f : [0, 1] \rightarrow \mathbb{R}$, also known as calibrator function [51]. 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 $r \in [0, 1]$,

$$f(r, \bar{\boldsymbol{w}}) = \sum_{i=1}^{k} \max\left(0, 1 - |r(k-1) - (i-1)|\right) \bar{w}_i \tag{5.6}$$

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 $(1 - 2r)w_1 + 2rw_2$.

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:

$$y_i = \sum_{j=1}^{n} f\left(\frac{j-1}{n-1}, \bar{\boldsymbol{W}}_{i,:}\right) \vec{X}_{i,j} \tag{5.7}$$

$\boldsymbol{y}$ is now the pooled representation with a potentially varying set size n as input. When $n = k$, this reduces back to Equation 5.5. For most experiments, we simply set $k = 20$ without tuning it.

5.4.3 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 5.7 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 5.7 as

$$\vec{X}'_{i,j} = f\left(\frac{j-1}{n-1}, \bar{\boldsymbol{W}}'_{i,:}\right) y'_i \tag{5.8}$$

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. To avoid this issue, we need the permutation that the sort produces to be differentiable. To achieve this, we use the recently proposed sorting networks [40], which is 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.

$$X'_{i,j} = (\vec{\boldsymbol{X}}'_{i,:} \boldsymbol{P}_i^{\top})_j \tag{5.9}$$

where $\boldsymbol{P}_i^{\top}$ 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 [69] 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.

5.5 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 [112] and PointNet [84], 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 by Murphy et al. [76]. 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 [114] 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. [19] and Gao et al. [34] 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 et al. [34] propose a graph auto-encoder where the decoder use the “inverse” of what the top-k operator does in the encoder, similar to our approach. Instead of numerically sorting, Mena et al. [71] and our Permutation-optimisation model (Chapter 4) *learn* an ordering of set

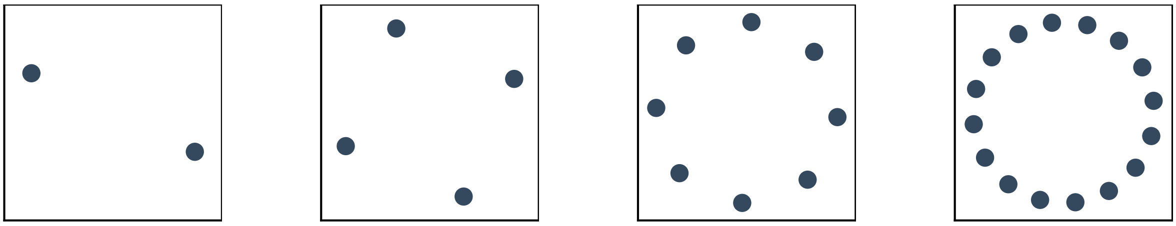


Figure 5.5: Examples from polygon dataset for $n \in \{2, 4, 8, 16\}$.

elements instead. Our FSPool model introduced here has the benefit of only $\Theta(n \log n)$ time complexity, compared to Permutation-optimisation with $\Theta(n^3)$, so it is much easier to use with larger sets and thus a greater variety of tasks.

Outside of the set learning literature, rank-based pooling in a convolutional neural network has been used in Shi et al. [92], 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 [5] and improved robustness to adversarial examples [24].

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 [30, 107, 1]. An interesting alternative approach is to perform the set generation sequentially [96, 52, 110]. The difficulty lies in how to turn the set into one or multiple sequences, which these papers try to solve in different ways.

5.6 Experiments

We start with two auto-encoder experiments, then move to tasks where we replace the pooling in an established model with FSPool. Full results can be found in the appendices, experimental details can be found in Section A.3, and we provide our code for reproducibility at `https://github.com/Cyanogenoid/fspool`.

5.6.1 Rotating polygons

We start with our simple dataset of auto-encoding regular polygons (Section 5.3), 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. We show some examples of different set sizes in Figure 5.5.

Table 5.1: Direct mean squared error (in hundredths) on Polygon dataset with different number of points in the set. Lower is better.

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 5.2: Chamfer loss (in hundredths) on Polygon dataset with different number of points in the set. Lower is better.

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 5.3: Hungarian loss (in hundredths) on Polygon dataset with different number of points in the set. Lower is better.

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

Model The encoder contains a 2-layer MLP applied to each set element, FSPool, and a 2-layer MLP to produce the latent space. The decoder contains a 2-layer MLP, FSUnpool, and a 2-layer MLP applied on each set element. We train this model to minimise the mean squared error. As baseline, we use a model where the decoder has been replaced with an MLP and train it with either the linear assignment or Chamfer loss (equivalent to AE-EMD and AE-CD models in Achlioptas et al. [1]). We also include a random baseline that outputs a polygon with the correct size and centre, but random rotation.

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. In Table 5.1, Table 5.2, and Table 5.3, we show the results of various model and training loss combinations. For our training runs with set sizes up to 128, our auto-encoder is able to reconstruct the point set close to perfectly. Meanwhile, the baseline converges significantly slower with high reconstruction error when the number of points is 8 or fewer and outputs the same set irrespective of input above that, regardless of loss function. This shows that FSPool with the direct MSE training loss is clearly better than the baseline trained with either Hungarian or Chamfer loss.

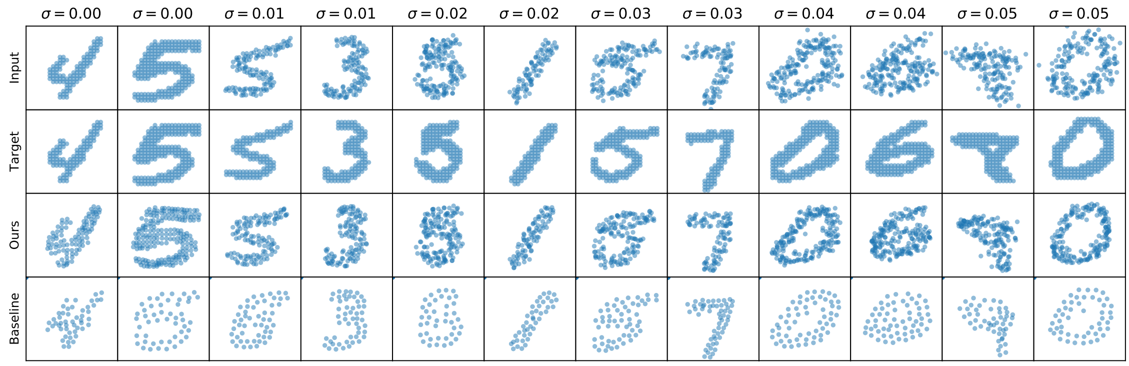


Figure 5.6: MNIST as point sets with different amounts of Gaussian noise (σ) and their reconstructions. The baseline uses sum pooling and an MLP decoder, which had the best quantitative results among the baselines. We used the best network for our model (0.28×10^4 average Chamfer loss) and the best network for the baseline model (0.20×10^4 average Chamfer loss). The examples are not cherry-picked.

Even when significantly increasing the latent size, dimensionality of layers, tweaking the learning rate, and replacing FSPool in the encoder with sum, mean, or max, the baseline trained with the Hungarian or Chamfer loss fails completely at 16 points. We verified that for 4 points, the baseline shows the discontinuous jump behaviour in the outputs as we predict in Figure 5.2.

This experiment highlights the difficulty of learning this simple dataset with traditional approaches due to the responsibility problem, while our model is able to fit this dataset with ease.

5.6.2 Noisy MNIST reconstruction

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.

Model We use exactly the same architecture as on the polygon dataset. As baseline models, we combine sum/mean/max pooling encoders with MLP/LSTM decoders and train with the Chamfer loss. This closely corresponds to the AE-CD approach [1] with the MLP decoder and the model by Stewart et al. [96] with the LSTM decoder.

Results We show the quantitative results for the default MNIST setting in Table 5.4. Interestingly, the sum pooling baseline has a lower Chamfer reconstruction error than our model, despite the example outputs in Figure 5.6 looking clearly worse. This demonstrates the

Table 5.4: Test Chamfer loss (in 1000ths) for MNIST for different input noise levels σ over 6 runs. Lower is better.

Noise σ	0.00	0.01	0.02	0.03	0.04	0.05
FSPool + FSUnpool	0.42$_{\pm 0.06}$	0.34$_{\pm 0.05}$	0.36$_{\pm 0.02}$	0.38$_{\pm 0.03}$	0.41$_{\pm 0.00}$	0.44$_{\pm 0.01}$
Sum + MLP	**0.30**$_{\pm 0.04}$	**0.28**$_{\pm 0.03}$	**0.28**$_{\pm 0.03}$	**0.28**$_{\pm 0.03}$	**0.27**$_{\pm 0.01}$	**0.31**$_{\pm 0.04}$
Sum + RNN	0.76$_{\pm 0.46}$	0.58$_{\pm 0.06}$	0.57$_{\pm 0.09}$	0.54$_{\pm 0.11}$	0.64$_{\pm 0.13}$	0.78$_{\pm 0.39}$
Max + MLP	1.29$_{\pm 0.23}$	1.37$_{\pm 0.28}$	1.23$_{\pm 0.16}$	1.74$_{\pm 0.32}$	1.27$_{\pm 0.19}$	1.43$_{\pm 0.30}$
Mean + MLP	1.41$_{\pm 0.12}$	1.22$_{\pm 0.18}$	1.33$_{\pm 0.29}$	1.25$_{\pm 0.09}$	1.31$_{\pm 0.15}$	1.49$_{\pm 0.31}$

Table 5.5: Test Chamfer loss (in 1000ths) for MNIST with additional mask features on every element for different input noise levels σ over 6 runs. Lower is better.

Noise σ	0.00	0.01	0.02	0.03	0.04	0.05
FSPool + FSUnpool	**0.28**$_{\pm 0.03}$	**0.21**$_{\pm 0.01}$	**0.25**$_{\pm 0.03}$	**0.25**$_{\pm 0.01}$	**0.27**$_{\pm 0.01}$	**0.30**$_{\pm 0.01}$
Sum + MLP	0.87$_{\pm 0.43}$	0.90$_{\pm 0.39}$	0.61$_{\pm 0.40}$	0.99$_{\pm 0.84}$	0.63$_{\pm 0.27}$	0.61$_{\pm 0.24}$
Sum + RNN	0.58$_{\pm 0.13}$	0.69$_{\pm 0.16}$	0.60$_{\pm 0.18}$	1.35$_{\pm 1.53}$	0.73$_{\pm 0.12}$	0.63$_{\pm 0.13}$
Max + MLP	5.91$_{\pm 3.10}$	4.78$_{\pm 3.05}$	7.10$_{\pm 2.40}$	5.05$_{\pm 2.87}$	5.85$_{\pm 3.11}$	4.57$_{\pm 2.08}$
Mean + MLP	5.92$_{\pm 3.10}$	4.55$_{\pm 3.17}$	6.84$_{\pm 2.84}$	7.11$_{\pm 2.39}$	3.04$_{\pm 0.78}$	6.34$_{\pm 2.53}$

weakness of the Chamfer loss we mentioned in Subsection 2.4.1 when comparing multisets. Our model avoids this weakness by being trained with a normal MSE loss (with the trade-off of a potentially higher Chamfer loss), which is not possible with the baselines. This MSE loss ensures that our model always has to predict the correct set size. The sum pooling baseline has a better test Chamfer loss because it is trained to minimise it, but it is also solving an easier task, since it does not need to distinguish padding from non-padding elements.

The main reason for this difference comes from the shortcoming of the Chamfer loss in distinguishing sets with duplicates or near-duplicates. For example, the Chamfer loss between $[1, 1.001, 9]$ and $[1, 9, 9.001]$ is close to 0. Most points in an MNIST set are quite close to many other points and there are many duplicate padding elements, so this problem with the Chamfer loss is certainly present on MNIST. That is why minimising MSE can lead to different results with higher Chamfer loss than minimising Chamfer loss directly, even though the qualitative results seem worse for the latter.

We can make the comparison between our model and the baselines more similar by forcing the models to predict an additional "mask feature" for each set element. This takes the value 1 when the point is present (non-padding element) and 0 (padding element) when not. This setting is useful for tasks where the predicted set size matters, as it allows points at the coordinates (0, 0) to be distinguished from padding elements. The results of this variant are shown in Table 5.5. Now, our model is clearly better: even though our auto-encoder minimises an MSE loss, the test Chamfer loss is also much better than all the baselines. Having to predict this additional mask feature does not affect our model predictions much because our model structure lets our model “know” which elements are padding elements, while this is much more challenging for the baselines.

We can also qualitatively compare our FSPool-FSUnpool model against the best baseline, which uses the sum pooling encoder and MLP decoder (Figure 5.6). 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 also makes big errors such as turning 5s into 8s (first $\sigma = 0.01$ example), which we have not observed with our model.

5.6.3 Noisy MNIST classification

Instead of 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 1 or 10 epochs and $\sigma = 0.05$ in Table 5.6, for $\sigma = 0.00$ in Table 5.7, and for 100 epochs and both σ values in Table 5.8. These are based on pre-trained models from the default MNIST setting without mask feature.

The FSPool-based models are consistently superior to all the baselines in all training settings. Even though our model can store information in the permutation that skips the latent space (the model could “cheat” the reconstruction by somehow storing everything in the permutation matrix), our latent space contains more information to correctly classify a set, even when the weights are fixed. Our model with fixed encoder weights already performs better after 1 epoch of training than the baseline models with unfrozen weights after 10 epochs of training. This shows the benefit of the FSPool-FSUnpool auto-encoder to the representation. When allowing the encoder weights to change (Unfrozen and Random init), our results again improve significantly over the baselines.

Table 5.6: Classification accuracy (mean ± stdev) 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	**82.2**%$_{\pm 2.1}$	**86.9**%$_{\pm 1.3}$	**84.7**%$_{\pm 1.9}$	**84.3**%$_{\pm 1.8}$	**91.5**%$_{\pm 0.5}$	**91.9**%$_{\pm 0.5}$
Sum	76.6%$_{\pm 1.3}$	68.7%$_{\pm 3.5}$	30.3%$_{\pm 5.6}$	79.0%$_{\pm 1.0}$	77.7%$_{\pm 2.3}$	72.7%$_{\pm 3.4}$
Mean	25.7%$_{\pm 3.6}$	32.2%$_{\pm 10.5}$	30.1%$_{\pm 1.6}$	36.8%$_{\pm 5.0}$	75.0%$_{\pm 2.7}$	73.0%$_{\pm 1.7}$
Max	73.6%$_{\pm 1.3}$	73.0%$_{\pm 3.5}$	56.1%$_{\pm 5.6}$	77.3%$_{\pm 0.9}$	80.4%$_{\pm 1.8}$	76.9%$_{\pm 1.3}$

Table 5.7: Classification accuracy (mean ± stdev) on MNIST $\sigma = 0.00$ over 6 runs.

	1 epoch of training			10 epochs of training		
	Frozen	Unfrozen	Random init	Frozen	Unfrozen	Random init
FSPool	**86.3**%$_{\pm 1.6}$	**92.3**%$_{\pm 1.1}$	**90.5**%$_{\pm 1.2}$	**88.2**%$_{\pm 1.4}$	**96.0**%$_{\pm 0.3}$	**96.1**%$_{\pm 0.3}$
Sum	82.3%$_{\pm 1.2}$	77.9%$_{\pm 3.4}$	35.3%$_{\pm 8.3}$	85.0%$_{\pm 0.8}$	84.2%$_{\pm 2.5}$	78.4%$_{\pm 3.9}$
Mean	27.0%$_{\pm 3.3}$	43.5%$_{\pm 7.1}$	31.2%$_{\pm 1.0}$	42.0%$_{\pm 7.7}$	76.7%$_{\pm 2.6}$	77.2%$_{\pm 2.2}$
Max	82.0%$_{\pm 1.8}$	84.1%$_{\pm 1.4}$	62.9%$_{\pm 3.5}$	86.8%$_{\pm 0.9}$	91.9%$_{\pm 1.3}$	87.7%$_{\pm 1.2}$

Table 5.8: Classification accuracy (mean ± stdev) on MNIST for 100 epochs over 6 runs.

	$\sigma = 0.05$, 100 epochs			$\sigma = 0.00$, 100 epochs		
	Frozen	Unfrozen	Random init	Frozen	Unfrozen	Random init
FSPool	**84.9**%$_{\pm 1.7}$	**93.9**%$_{\pm 0.4}$	**94.0**%$_{\pm 0.3}$	88.6%$_{\pm 1.6}$	**97.4**%$_{\pm 0.3}$	**97.5**%$_{\pm 0.3}$
Sum	79.8%$_{\pm 1.0}$	85.3%$_{\pm 1.1}$	83.1%$_{\pm 1.9}$	85.6%$_{\pm 0.9}$	89.5%$_{\pm 2.5}$	88.3%$_{\pm 1.4}$
Mean	48.2%$_{\pm 6.9}$	86.5%$_{\pm 0.8}$	84.1%$_{\pm 2.3}$	57.0%$_{\pm 7.7}$	90.3%$_{\pm 1.3}$	91.1%$_{\pm 0.8}$
Max	78.8%$_{\pm 0.8}$	84.7%$_{\pm 1.0}$	84.6%$_{\pm 0.9}$	**89.2**%$_{\pm 0.8}$	95.3%$_{\pm 0.7}$	95.1%$_{\pm 1.5}$

Interestingly, switching the relaxed sort to the unrelaxed sort in our model when using the fixed auto-encoder weights does not hurt accuracy. Training the FSPool model takes 45 seconds per epoch on a GTX 1080 GPU, only slightly more than the baselines with 37 seconds per epoch.

Note that while Qi et al. [84] report an accuracy of ~99% on a similar set version of MNIST, our model uses noisy sets as input and is much smaller and simpler: we have 3820 parameters, while their model has 1.6 million parameters. Our model also does not use dropout, batch norm, a branching network architecture, nor a stepped learning rate schedule. When we try to match their model size, our accuracies for $\sigma = 0.00$ increase to ~99% as well.

5.6.4 CLEVR

CLEVR [52] 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.

Model For this dataset, we compare against relation networks (RN) [89] – explicitly modeling all pairwise relations –, Janossy pooling [76], and regular pooling functions. While the original RN paper reports a result of 96.4% for this dataset, we use a tuned implementation by Messina et al. [72] with 2.6% better accuracy. For our model, we modify this to not operate on pairwise relations and replace the existing sum pooling with FSPool. We use the same hyperparameters for our model as the strong RN baseline without further tuning them.

Results Over 10 runs, Table 5.9 shows that our FSPool model reaches the best accuracy and also reaches the listed accuracy milestones in fewer epochs than all baselines. The difference in accuracy is statistically significant (two-tailed t-tests against sum, mean, RN, all with $p \approx 0.01$). Also, FSPool reaches 99% accuracy in 5.3 h, while the fastest baseline, mean pooling, reaches the same accuracy in 6.2 h. Surprisingly, RNs do not provide any benefit here, despite the hyperparameters being explicitly tuned for the RN model. We show some of the functions $f(\cdot, \bar{\boldsymbol{W}})$ that FSPool has learned in Figure 5.7. These confirm that FSPool uses more complex functions than just sums or maximums, which

Table 5.9: CLEVR results over 10 runs: mean ± stdev of accuracy after 350 epochs, epochs to reach an accuracy milestone, and wall time required with a 1080 Ti GPU. * averages over only 8 runs because 2 runs did not reach 99%. MAC [48] is a model specifically designed for CLEVR and the state-of-the-art for *image inputs* and without program supervision.

		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
Janossy	97.00%$_{\pm 0.54}$	–	–	–	11.5 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
MAC	99.0 %	–	–	–	–

allow it to capture more information about the set than other pooling functions.

5.6.5 Graph classification

We perform a large number of experiments on various graph classification datasets from the TU repository [55]: 4 graph datasets from bioinformatics (for example with the graph encoding the structure of a molecule) and 5 datasets from social networks (for example with the graph encoding connectivity between people who worked with each other). The task is to classify the whole graph into one of multiple classes such as positive or negative drug response.

Model We use the state-of-the-art graph neural network GIN [105] as baseline. This involves a series of graph convolutions (which includes aggregation of features from each node's set of neighbours into the node), a readout (which aggregates the set of all nodes into one feature vector), and a classification with an MLP. We replace the usual sum or mean pooling readout with FSPool $k = 5$ for our model. We repeat 10-fold cross-validation on each dataset 10 times and use the same hyperparameter ranges as Xu et al. [105] for our model and the GIN baseline.

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. The hyperparameter sweep is done based on best validation accuracy for each fold in the cross-validation individually and over the same combinations as specified in GIN.

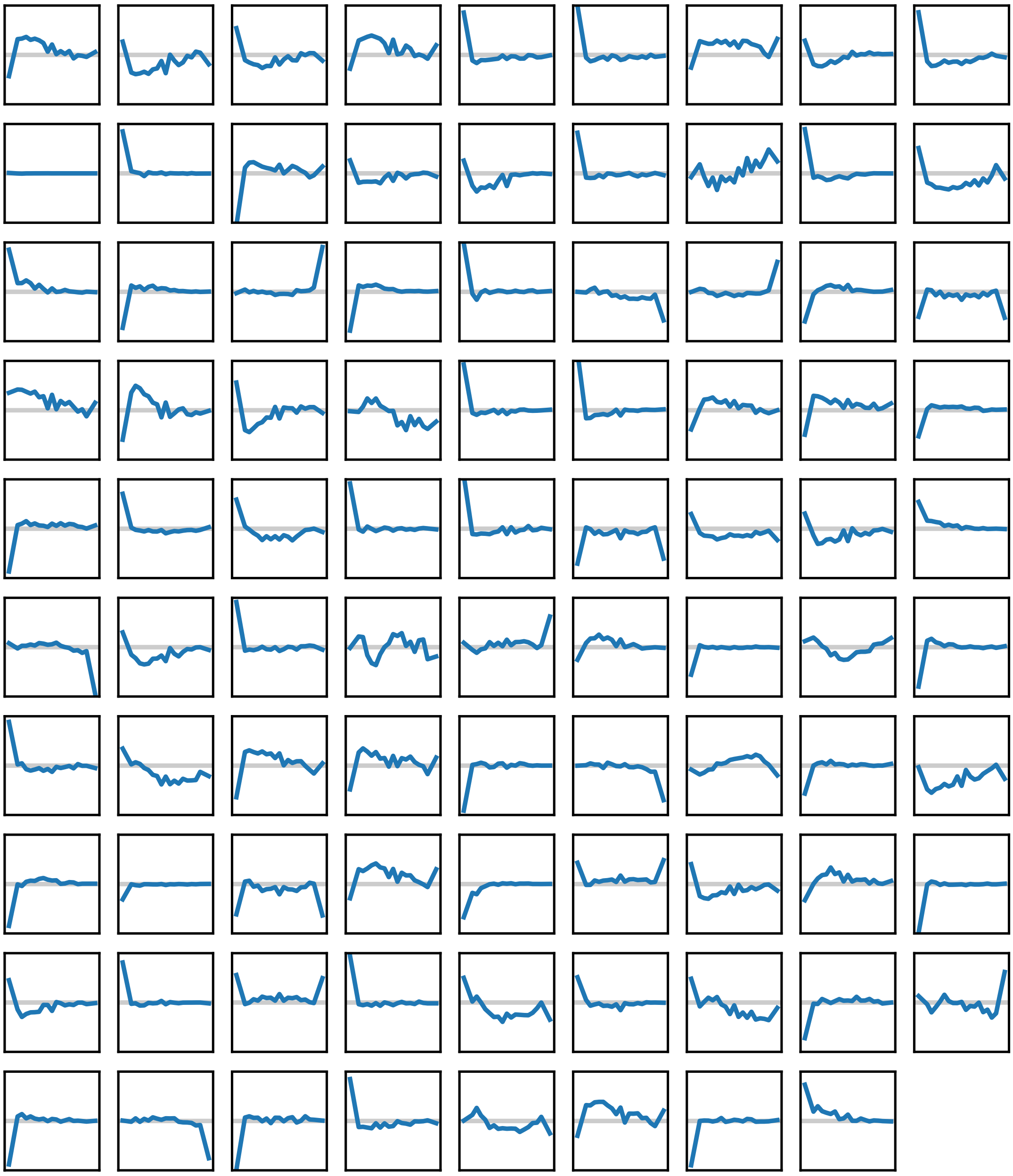


Figure 5.7: Shapes of piecewise linear functions learned by the FSPool model on CLEVR. These show $r \in [0, 1]$ on the x-axis and $f(r, \bar{\boldsymbol{w}})$ on the y-axis for a particular $\bar{\boldsymbol{w}}$ of a fully-trained model. A common shape among these functions are variants of max pooling: 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.

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 73.8% to 77.1% if we were to select based on best test accuracy, which would be better than their 76.2%.

While we initially also used $k = 20$ in FSPool for this experiment, we found that $k = 5$ was consistently an improvement. The $k = 20$ model was still better than the baseline on average by a smaller margin.

Results We show the results in Table 5.10. On 6 out of 9 datasets, GIN-FSPool achieves better test accuracy than GIN-Base. On a different 6 datasets, it converges to the best validation accuracy faster.

The most significant improvements are on the two RDT datasets. 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 only 75 nodes. This is perhaps evidence that FSPool is helping to avoid the bottleneck problem of pooling a large set of feature vectors to a single feature vector. Mean and sum pooling are mostly able to keep up with FSPool on the smaller set sizes, but are bottlenecked for the RDT datasets with the much larger set sizes.

A Wilcoxon signed-rank test shows that the difference in accuracy to the standard GIN has $p \approx 0.07$ ($W = 7$) and the difference in convergence speed has $p \approx 0.11$ ($W = 9$). Keep in mind that just because the results have $p > 0.05$, it does not mean that the results are invalid.

We emphasise that the main comparison to be made is between the GIN-Sum and the GIN-FSPool model in the last four rows, 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, while our results are averaged over 10 repeats of 10-fold cross validation.

Table 5.10: 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
DCNN [7]	49.1	33.5	–	–	52.1
Patchy-san [78]	71.0 ±2.3	45.2 ±2.8	86.3 ±1.6	49.1 ±0.7	72.6 ±2.2
SortPool [114]	70.0 ±0.9	47.8 ±0.9	–	–	73.8 ±0.5
DiffPool [109]	–	–	–	–	75.5
WL* [105]	73.8	50.9	81.0	52.5	78.9
GIN-Base* [105]	75.1	52.3	92.4	57.5	80.2
GIN-FSPool	**72.1** ±2.0	**49.9** ±1.7	**89.1** ±1.2	**51.8** ±0.9	80.0 ±0.4
- *epochs*	95 ±70	**27** ±23	**124** ±64	**66** ±31	**124** ±56
GIN-Base	71.3 ±1.2	48.8 ±1.7	84.8 ±1.7	48.1 ±2.0	**80.3** ±0.4
- *epochs*	**83** ±73	57 ±59	156 ±58	211 ±27	204 ±26

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 [77]	76.0 ±2.7	73.7 ±0.7	59.5 ±2.4	82.5 ±0.5
DCNN [7]	67.0	61.3	56.6	62.6
Patchy-san [78]	92.6 ±4.2	75.9 ±2.8	60.0 ±4.8	78.6 ±1.9
SortPool [114]	85.8 ±1.7	75.5 ±0.9	58.6 ±2.5	74.4 ±0.5
DiffPool [109]	–	76.3	–	–
WL [91]	84.1 ±1.9	74.7 ±0.5	58.0 ±2.5	85.5 ±0.5
WL* [105]	90.4	75.0	59.9	86.0
GIN-Base* [105]	89.4	76.2	64.6	82.7
GIN-FSPool	**85.9** ±2.4	**73.8** ±0.9	59.3 ±1.8	79.2 ±0.6
- *epochs*	299 ±91	**69** ±23	214 ±110	**361** ±54
GIN-Base	85.0 ±1.5	73.2 ±1.2	**59.9** ±2.4	**79.4** ±0.6
- *epochs*	**244** ±95	160 ±123	**202** ±100	412 ±55

5.7 Discussion

In this chapter, we identified the responsibility problem with existing approaches for predicting sets and introduced FSPool, which provides a way around this issue in auto-encoders. In experiments on two datasets of point clouds, we showed that this results in much better reconstructions. We believe that this is an important step towards set prediction tasks with more complex set elements. However, because our decoder uses information from the encoder, it is not easily possible to turn it into a generative set model, which is the main limitation of our approach. Still, we find that using the auto-encoder to obtain better representations and pre-trained weights can be beneficial by itself. We will use our insights about the responsibility problem to create a model without the auto-encoder limitation in Chapter 6.

In classification experiments, we also showed that simply replacing the pooling function in an existing model with FSPool can give us better results and faster convergence. We showed that FSPool consistently learns better set representations at a relatively small computational cost, leading to improved results in the downstream task. Our model thus has immediate applications in various types of set models that have traditionally used sum or max pooling. For example, we will use FSPool in the next chapter (replacing the pooling in a Relation Network) to improve the quality of the learned representations. It would be useful to theoretically characterise what types of relations are more easily expressed by FSPool through an analysis like in Murphy et al. [76]. This may result in further insights into how to learn better set representations efficiently.

Chapter 6

Set decoder: Deep set prediction networks

In the previous chapter, we found that all existing approaches for predicting sets suffer from the responsibility problem. The solution we provided there only covered the auto-encoder setting. In this chapter, we will develop a solution without this restriction: a model that can predict sets without the responsibility problem in normal supervised learning tasks.

These contributions have been published as [115] in Advances in Neural Information Processing Systems 32 (NeurIPS), 2019, and have been presented at the Sets & Partitions workshop, hosted at the Neural Information Processing Systems (NeurIPS) 2019 conference.

6.1 Introduction

You are given a rotation angle and your task is to draw the four corner points of a square that is rotated by that amount. This is a structured prediction task where the output is a *set*, since there is no inherent ordering to the four points. Such sets are a natural representation for many kinds of data, ranging from the set of points in a point cloud, to the set of objects in an image (object detection), to the set of nodes in a molecular graph (molecular generation). Yet, existing machine learning models often struggle to solve even the simple square prediction task as we showed in Section 5.3.

The main difficulty in predicting sets comes from the ability to permute the elements in a set freely, which means that there are $n!$ equally good solutions for a set of size n. Models that do not take this set structure into account properly (such as MLPs or RNNs) result in discontinuities, which is the reason why they struggle to solve simple toy set prediction tasks. We quickly review the background on what the problem is in Section 6.2.

How can we build a model that properly respects the set structure of the problem so that we can predict sets without running into discontinuity issues? In this chapter, we aim to address this question. Concretely, we contribute the following:

1. We propose a model (Section 6.3, Algorithm 6.1) that can predict a set from a feature vector (vector-to-set) while properly taking the structure of sets into account. We explain what properties we make use of that enables this. Our model uses backpropagation through a set encoder to decode a set and works for variable-size sets. The model is applicable to a wide variety of set prediction tasks since it only requires a feature vector as input.

2. We evaluate our model on several set prediction datasets (Section 6.5). First, we demonstrate that the auto-encoder version of our model is sound on a set version of MNIST. Next, we use the CLEVR dataset to show that this works for general set prediction tasks. We predict the set of bounding boxes of objects in an image and we predict the set of object attributes in an image, both from a single feature vector. Our model is a completely different approach to usual anchor-based object detectors because we pose the task as a set prediction problem, which does not need complicated post-processing techniques such as non-maximum suppression.

6.2 Background

Representation We are interested in sets of feature vectors with the feature vector describing properties of the element, for example the 2d position of a point in a point cloud. A set of size n wherein each feature vector has dimensionality d is represented as a matrix $\boldsymbol{Y} \in \mathbb{R}^{n \times d}$ with the elements as rows in an arbitrary order, $\boldsymbol{Y} = [\boldsymbol{y}_1, \dots, \boldsymbol{y}_n]^\top$. Note that this representation of sets is transposed compared to Chapter 2 ($\mathbb{R}^{n \times d}$ instead of $\mathbb{R}^{d \times n}$) to ease the exposition in this chapter. To properly treat this as a set, it is important to only apply operations with certain properties to it [112]: *permutation-invariance* or *permutation-equivariance*. In other words, operations on sets should not rely on the arbitrary ordering of the elements.

Set encoders (which turn such sets into feature vectors) are usually built by composing permutation-equivariant operations with a permutation-invariant operation at the end. A simple example is the Deep Sets model by Zaheer et al. [112]: $f(\boldsymbol{Y}) = \sum_i g(\boldsymbol{y}_i)$ where g is a neural network.

Because g is applied to every element individually, it does not rely on the arbitrary order of the elements. We can think of this as turning the set $\{\boldsymbol{y}_i\}_{i=1}^n$ into $\{g(\boldsymbol{y}_i)\}_{i=1}^n$. This is permutation-equivariant because changing the order of elements in the input set affects the output set in a predictable way. Next, the set is summed to produce a single feature vector. Since summing is commutative, the output is the same regardless of what order the elements are in. In other words, summing is permutation-invariant. This gives us an encoder that produces the same feature vector regardless of the arbitrary order the set elements were stored in.

Loss In set prediction tasks, we need to compute a loss between a predicted set $\hat{Y} = [\hat{\boldsymbol{y}}_1, \ldots, \hat{\boldsymbol{y}}_n]^\top$ and the target set Y. The main problem is that the elements of each set are in an arbitrary order, so we cannot simply compute a pointwise distance. The usual solution to this is an assignment mechanism that matches up elements from one set to the other set. This gives us a loss function that is permutation-invariant in both its arguments.

One such loss is the $O(n^2)$ Chamfer loss, which matches up every element of $\hat{Y}$ to the closest element in Y and vice versa:

$$L_{\text{cha}}(\hat{Y}, Y) = \sum_i \min_j \left\|\hat{\boldsymbol{y}}_i - \boldsymbol{y}_j\right\|^2 + \sum_j \min_i \left\|\hat{\boldsymbol{y}}_i - \boldsymbol{y}_j\right\|^2 \tag{6.1}$$

Note that this does not work well for multisets: the loss between $[\boldsymbol{a}, \boldsymbol{a}, \boldsymbol{b}]$, $[\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{b}]$ is 0. A more sophisticated loss that does not have this problem involves the linear assignment problem with the pairwise losses as assignment costs:

$$L_{\text{hun}}(\hat{Y}, Y) = \min_{\pi \in \Pi} \sum_i \left\|\hat{\boldsymbol{y}}_i - \boldsymbol{y}_{\pi(i)}\right\|^2 \tag{6.2}$$

where Π is the space of permutations, which can be solved with the Hungarian algorithm in $O(n^3)$ time. This has the benefit that every element in one set is associated to exactly one element in the other set, which is not the case for the Chamfer loss.

Responsibility problem A widely-used approach is to simply ignore the set structure of the problem. A feature vector can be mapped to a set $\hat{Y}$ by using an MLP that takes the vector as input and directly produces $\hat{Y}$ with $n \times d$ outputs. Since the order of elements in $\hat{Y}$ does not matter, it appears reasonable to always produce them in a certain order based on the weights of the MLP.

While this seems like a promising approach, we pointed out Section 5.3 that this results in a discontinuity issue: there are points where a small change in set space requires a large change in the neural network outputs. The model needs to "decide" which of its outputs is responsible for producing which element, and this responsibility must be resolved discontinuously.

Here, we give a different example to give the intuition behind this problem. Consider an MLP that detects the colour of two otherwise identical objects present in an image, so it has two outputs with dimensionality 3 (R, G, B) corresponding to those two colours. We are given an image with a blue and red object, so let us say that output 1 predicts blue and output 2 predicts red; perhaps the weights of output 1 are more attuned to the blue channel and output 2 is more attuned to the red channel. We are given another image with a blue and green object, so it is reasonable for output 1 to again predict blue and output 2 to now predict green. When we now give the model an image with a red and green object, or two red objects, it is unclear which output should be responsible for predicting which object. Output 2 "wants" to predict both red and green, but has to decide between one of them, and output 1 now has to be responsible for the other object while previously being a blue detector. This responsibility must be resolved discontinuously, which makes modeling sets with MLPs difficult.

The main problem is that there is a notion of output 1 and output 2 – an ordered output representation – in the first place, which forces the model to give the set an order. Instead, it would be better if the outputs of the model were freely interchangeable – in the same way the elements of the set are interchangeable – to not impose an order on the outputs. This is exactly what our model accomplishes.

6.3 Deep set prediction networks

This section contains our primary contribution: a model for decoding a feature vector into a set of feature vectors. As we have previously established, it is important for the model to properly respect the set structure of the problem to avoid the responsibility problem.

Our main idea is based on the observation that the gradient of a set encoder with respect to the input set is permutation-equivariant (see Subsection 6.3.1): *to decode a feature vector into a set, we can use gradient descent to find a set that* encodes *to that feature vector.* Since each update of the set using the gradient is permutation-equivariant, we always properly treat it as a set and avoid the responsibility problem. This gives rise to a nested optimisation: an inner loop that changes a set to encode more similarly to the input feature vector, and an outer

Algorithm 6.1 One forward pass of the set prediction algorithm within the training loop.

$z = F(x)$ ▹ encode input with a model
$\hat{Y}^{(0)} \leftarrow$ init ▹ initialise set
for t $\leftarrow 1, T$ **do**
    $l \leftarrow L_{\text{repr}}(\hat{Y}^{(t-1)}, z)$ ▹ compute representation loss
    $\hat{Y}^{(t)} \leftarrow \hat{Y}^{(t-1)} - \eta \frac{\partial l}{\partial \hat{Y}^{(t-1)}}$ ▹ gradient descent step on the set
end for
predict $\hat{Y}^{(T)}$
$\mathcal{L} = \frac{1}{T} \sum_{t=0}^{T} L_{\text{set}}(\hat{Y}^{(t)}, Y) + \lambda L_{\text{repr}}(Y, z)$ ▹ outer optimisation loss

loop that changes the weights of the encoder to minimise a loss over a dataset.

With this idea in mind, we build up models of increasing usefulness for predicting sets. We start with the simplest case of auto-encoding fixed-size sets (Subsection 6.3.2), where a latent representation is decoded back into a set. This is modified to support variable-size sets, which is necessary for most sets encountered in the real-world. Lastly and most importantly, we extend our model to general set prediction tasks where the input no longer needs to be a set (Subsection 6.3.3). This gives us a model that can predict a set of feature vectors from a single feature vector. We give the pseudo-code of this method in Algorithm 6.1.

6.3.1 Proof of permutation-equivariance

Recall definitions of permutation-invariance and equivariance:

Definition 3 A function $f : \mathbb{R}^{n \times d} \rightarrow \mathbb{R}^{c}$ is permutation-invariant iff it satisfies:

$$f(\boldsymbol{X}) = f(\boldsymbol{PX}) \tag{6.3}$$

for all permutation matrices $\boldsymbol{P}$.

Definition 4 A function $g : \mathbb{R}^{n \times d} \rightarrow \mathbb{R}^{n \times c}$ is permutation-equivariant iff it satisfies:

$$\boldsymbol{P}g(\boldsymbol{X}) = g(\boldsymbol{PX}) \tag{6.4}$$

for all permutation matrices $\boldsymbol{P}$.

The definitions here use the transposed set representations to match the exposition in this chapter. With these definitions, we can prove the following:

Theorem 2 *The gradient of a permutation-invariant function $f : \mathbb{R}^{n\times d} \to \mathbb{R}^c$ with respect to its input is permutation-equivariant:*

$$P\frac{\partial f(X)}{\partial X} = \frac{\partial f(PX)}{\partial PX} \tag{6.5}$$

Proof. Using Definition 1, the chain rule, and the orthogonality of P:

$$\begin{aligned} P\frac{\partial f(X)}{\partial X} &= P\frac{\partial f(PX)}{\partial X} && (6.6)\\ &= P\frac{\partial PX}{\partial X}\frac{\partial f(PX)}{\partial PX} && (6.7)\\ &= PP^\top\frac{\partial f(PX)}{\partial PX} && (6.8)\\ &= \frac{\partial f(PX)}{\partial PX} && (6.9) \end{aligned}$$

□

This property ensures that our model is permutation-equivariant.

6.3.2 Auto-encoding fixed-size sets

In a set auto-encoder, the goal is to turn the input set Y into a small latent space $z = g_{\text{enc}}(Y)$ with the encoder g_{enc} and turn it back into the predicted set $\hat{Y} = g_{\text{dec}}(z)$ with the decoder g_{dec}. Using our main idea, we define a *representation loss* and the corresponding decoder as:

$$L_{\text{repr}}(\hat{Y}, z) = \left\|g_{\text{enc}}(\hat{Y}) - z\right\|^2 \tag{6.10}$$

$$g_{\text{dec}}(z) = \arg\min_{\hat{Y}} L_{\text{repr}}(\hat{Y}, z) \tag{6.11}$$

In essence, L_{repr} compares $\hat{Y}$ to Y in the latent space. To understand what the decoder does, first consider the simple, albeit not very useful case of the identity encoder $g_{\text{enc}}(Y) = Y$. Solving $g_{\text{dec}}(z)$ simply means setting $\hat{Y} = Y$, which perfectly reconstructs the input as desired.

When we instead choose g_{enc} to be a set encoder, the latent representation z is a permutation-invariant feature vector. If this representation is "good", $\hat{Y}$ will only encode to similar latent variables as Y if the two sets themselves are similar. Thus, the minimisation in Equation 6.11 should still produce a set $\hat{Y}$ that is the same (up to permutation) as Y, except this has now been achieved with z as a bottleneck.

Since the problem is non-convex when g_{enc} is a neural network, it is infeasible to solve Equation 6.11 exactly. Instead, we perform gradient

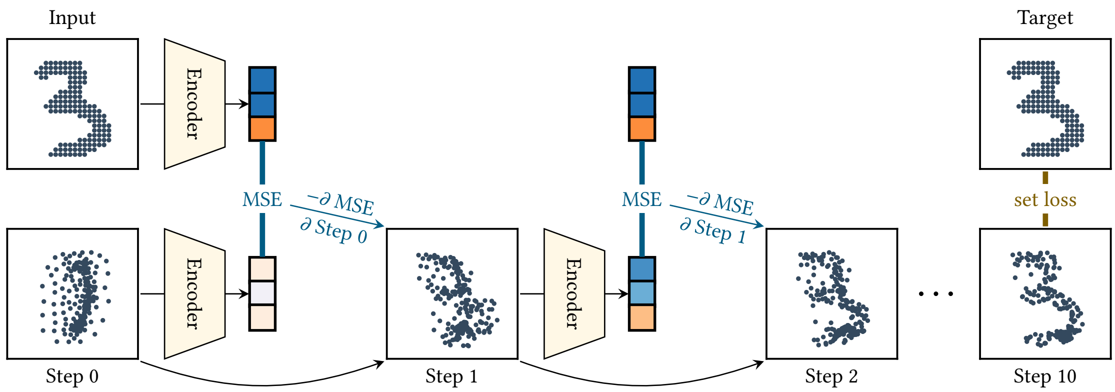


Figure 6.1: Overview of our algorithm for auto-encoding an MNIST set. The initial set at step 0 is iteratively refined, at each step improving the current prediction.

descent to approximate a solution. Starting from some initial set $\hat{Y}^{(0)}$, gradient descent is performed for a fixed number of steps T with the update rule:

$$\hat{Y}^{(t+1)} = \hat{Y}^{(t)} - \eta \cdot \frac{\partial L_{\text{repr}}(\hat{Y}^{(t)}, \boldsymbol{z})}{\partial \hat{Y}^{(t)}} \tag{6.12}$$

with η as the learning rate and the prediction being the final state, $g_{\text{dec}}(\boldsymbol{z}) = \hat{Y}^{(T)}$. This is the aforementioned inner optimisation loop. In practice, we let $\hat{Y}^{(0)}$ be a learnable $\mathbb{R}^{d \times n}$ matrix which is part of the neural network parameters.

To obtain a good representation $\boldsymbol{z}$, we still have to train the weights of g_{enc}. For this, we compute the auto-encoder objective $L_{\text{set}}(\hat{Y}^{(T)}, Y)$ – with $L_{\text{set}} = L_{\text{cha}}$ or L_{hun} – and differentiate with respect to the weights as usual, backpropagating through the steps of the inner optimisation. This is the aforementioned outer optimisation loop.

In summary, each forward pass of our auto-encoder first encodes the input set to a latent representation as normal. To decode this back into a set, gradient descent is performed on an initial guess with the aim to obtain a set that encodes to the same latent representation as the input. The same set encoder is used in the encoding and decoding stages.

Variable-size sets To extend this from fixed- to variable-size sets, we make a few modifications to this algorithm. First, we pad all sets to a fixed maximum size to allow for efficient batch computation. We then concatenate an additional mask feature m_i to each set element $\hat{\boldsymbol{y}}_i$ that indicates whether it is a regular element ($m_i = 1$) or padding ($m_i = 0$). With this modification to $\hat{Y}$, we can optimise the masks in the same way as the set elements are optimised. To ensure that masks stay in the valid range between 0 and 1, we simply clamp values above 1 to 1 and

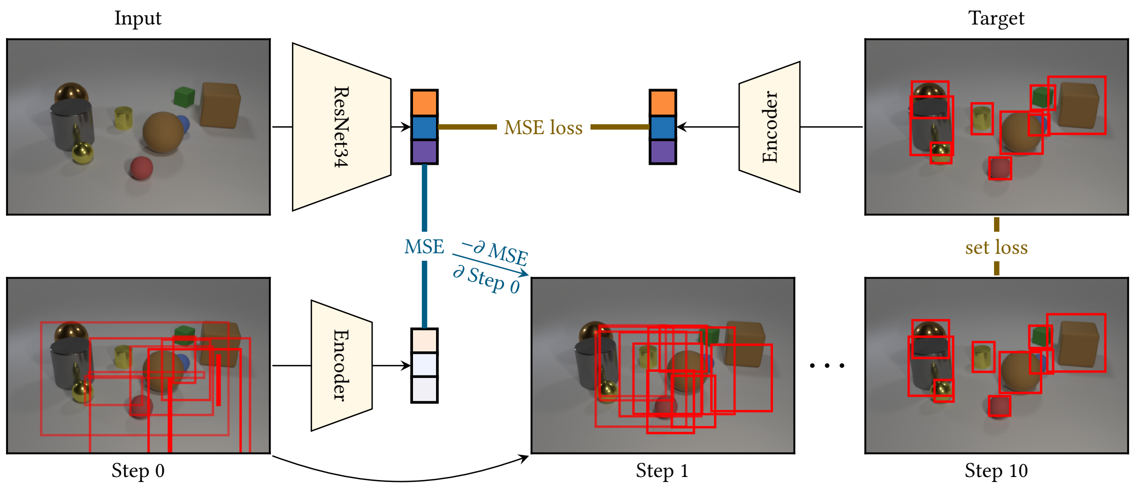


Figure 6.2: Overview of DSPN for supervised prediction. The main difference is the MSE loss between the input encoding and target encoding.

values below 0 to 0 after each gradient descent step. This performed better than using a sigmoid in our initial experiments, possibly because it allows exact 0s and 1s to be recovered.

6.3.3 Predicting sets from a feature vector

In our auto-encoder, we used an encoder to produce both the latent representation as well as to decode the set. This is no longer possible in the general set prediction setup, since the target representation z can come from a separate model (for example an image encoder F encoding an image x), so there is no longer a set encoder in the model.

When naïvely using $z = F(x)$ as input to our decoder, our decoding process is unable to predict sets correctly from it. Because the set encoder is no longer shared in our set decoder, there is no guarantee that optimising $g_{\text{enc}}(\hat{Y})$ to match z converges towards Y (or a permutation thereof). To fix this, we simply add a term to the loss of the outer optimisation that encourages $g_{\text{enc}}(Y) \approx z$ again. In other words, the target set should have a very low representation loss itself. This gives us an additional L_{repr} term in the loss function of the outer optimisation for supervised learning:

$$\mathcal{L} = L_{\text{set}}(\hat{Y}, Y) + \lambda L_{\text{repr}}(Y, z) \tag{6.13}$$

with L_{set} being either L_{cha} or L_{hun}. With this, minimising $L_{\text{repr}}(\hat{Y}, z)$ in the inner optimisation will converge towards Y. The additional term is not necessary in the pure auto-encoder because $z = g_{\text{enc}}(Y)$, so $L_{\text{repr}}(Y, z)$ is always 0 already.

Practical tricks For the outer optimisation, we can compute the set loss for not only $\hat{Y}^{(T)}$, but all $\hat{Y}^{(t)}$. That is, we use the average set loss $\frac{1}{T}\sum_t L_{\text{set}}(\hat{Y}^{(t)}, Y)$ as loss (similar to Belanger et al. [13]). This encourages $\hat{Y}$ to converge to Y quickly and not diverge with more steps, which increases the robustness of our algorithm.

We sometimes observed divergent training behaviour when the outer learning rate is set inappropriately. By replacing the instances of $||\cdot||^2$ in L_{set} and L_{repr} with the Huber loss (squared error for differences below 1 and absolute error above 1) – as is commonly done in object detection models – training became less sensitive to hyperparameter choices.

The inner optimisation can be modified to include a momentum term, which stops a prediction from oscillating around a solution. This gives us slightly better results, but we did not use this for any experiments to keep our method as simple as possible.

It is possible to explicitly include the sum of masks as a feature in the representation z for our model. This improves our results on MNIST – likely due to the explicit signal for the model to predict the correct set size – but again, we do not use this for simplicity.

6.4 Related work

The main approach we compare our method to is the simple method of using an MLP decoder to predict sets. This has been used for predicting point clouds [1, 30], bounding boxes [87, 11], and graphs (sets of nodes and edges) [20, 93]. These predict an ordered representation (list) and treat it as if it is unordered (set). As we discussed in Section 6.2, this approach runs into the responsibility problem. Some works on predicting 3d point clouds make domain-specific assumptions such as independence of points within a set [67] or grid-like structures [107]. To avoid inefficient graph matching losses, Yang et al. [106] compute a permutation-invariant loss between graphs by comparing them in the latent space (similar to our L_{repr}) in an adversarial setting.

An alternative approach is to use an RNN decoder to generate this list [68, 96, 101]. The problem can be made easier if it can be turned from a set into a sequence problem by giving a canonical order to the elements in the set through domain knowledge [101]. For example, You et al. [110] generate the nodes of a graph by ordering the set of nodes based on the node traversal order of a breadth-first search.

The closest work to ours is by Mordatch [75]. They also iteratively minimise a function (their energy function) in each forward pass of the neural network and differentiate through the iteration to learn the weights. They have only demonstrated that this works for modifying small sets of 2d elements in relatively simple ways, so it is unclear whether their approach scales to the harder problems such as object detection that we tackle in this chapter. In particular, minimising L_{repr} in our model has the easy-to-understand consequence of making the predicted set more similar to the target set, while it is less clear what minimising their learned energy function $E(\hat{Y}, z)$ does.

In Chapter 5, we constructed an auto-encoder that pools a set into a feature vector where information from the encoder is shared with their decoder. This is done to make the decoder permutation-equivariant, which we used to avoid the responsibility problem. However, this strictly limits the decoder to usage in auto-encoders – not set prediction – because it requires an encoder to be present during inference.

Greff et al. [39] construct an auto-encoder for images with a latent space that is set-structured. They are able to find latent sets of variables to describe an image composed of a set of objects with some task-specific assumptions. While interesting from a representation learning perspective, our model is immediately useful in practice because it works for general supervised learning tasks.

Our inspiration for using backpropagation through an encoder as a decoder comes from the line of introspective neural networks [62, 65] for image modeling. An important difference is that in these works, the two optimisation loops (generating predictions and learning the network weights) are performed in sequence, while ours are nested. The nesting allows our outer optimisation to differentiate through the inner optimisation. This type of nested optimisation to obtain structured outputs with neural networks was first studied by Belanger et al. [12, 13], of which our model can be considered an instance of. Note that Greff et al. [39] and Mordatch [75] also differentiate through an optimisation, which suggests that this approach is of general benefit when working with sets.By differentiating through a decoder rather than an encoder, Bojanowski et al. [16] learn a representation instead of a prediction.+

It is important to clearly separate the vector-to-set setting in this chapter from some related works on set-to-set mappings, such as the equivariant version of Deep Sets [112] and self-attention [100]. Tasks like object detection, where no set input is available, can not be solved with set-to-set methods alone; the feature vector from the image encoder has to be turned into a set first, for which a vector-to-set model like ours is necessary. Set-to-set methods do not have to deal with the responsibility problem, because the output usually has the same ordering as the input. Methods like the one in Mena et al. [71] and our method in Chapter 4 learn to predict a permutation matrix for a set (set-to-*set-of-position-assignments*). When this permutation is applied to the input set, the set is turned into a list (set-to-list). Again, our model is about producing a set as *output* while not necessarily taking a set as input.

6.5 Experiments

In the following experiments, we compare our set prediction network to a model that uses an MLP or RNN (LSTM) as set decoder. In all experiments, we fix the hyperparameters of our model to $T = 10, \eta = 800, \lambda = 0.1$. Further details about the model architectures, training settings, and hyperparameters are given in Section A.4. We provide the PyTorch [82] source code to reproduce all experiments at `https://github.com/Cyanogenoid/dspn`.

Note that there are two minor bugs (which do not affect the conclusions of the experiments) in the following results:

1. the CLEVR experiments computed the set loss only on the last step $\hat{Y}^{(T)}$,
2. the bounding box results are consistently overestimated.

Subsection A.4.3 shows our results with these bugs fixed.

6.5.1 MNIST

We begin with the task of auto-encoding a set version of MNIST. A set is constructed from each image by including all the pixel coordinates (x and y, scaled to the interval $[0, 1]$) of pixels that have a value above the mean pixel value. The size of these sets varies from 32 to 342 across the dataset.

Model In our model, we use a set encoder that processes each element individually with a 3-layer MLP, followed by FSPool (Chapter 5) as pooling function to produce 64 latent variables. These are decoded with our algorithm to predict the input set. We compare this against a baseline model with the same encoder, but with a traditional MLP or LSTM as decoder. This approach to decoding sets is used in models such as in Achlioptas et al. [1] (AE-CD variant) and Stewart et al. [96]; these baselines are representative of the best approaches for set prediction in the literature. Note that these baselines have significantly more parameters than our model, since our decoder has almost no additional parameters by sharing the encoder weights (ours: ~140 000 parameters, MLP: ~530 000, LSTM: ~470 000). For the baselines, we include a mask feature with each element to allow for variable-size sets. Due to the large maximum set size, use of Hungarian matching is too slow. Instead,

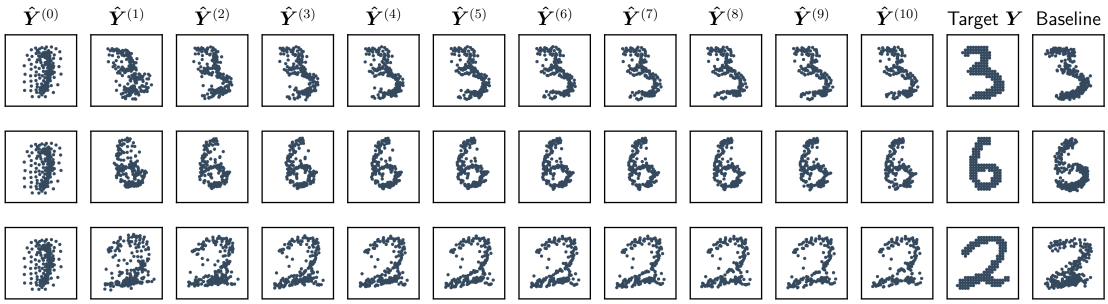


Figure 6.3: Progression of set prediction algorithm on MNIST ($\hat{Y}^{(t)}$). Our predictions come from our model with 0.08×10^{-3} loss, while the baseline predictions come from an MLP decoder model with 0.09×10^{-3} loss.

Table 6.1: Chamfer reconstruction loss on MNIST in 1000ths. Lower is better. Mean and standard deviation over 6 runs.

Model	Loss
MLP baseline	$0.21_{\pm 0.18}$
RNN baseline	$0.49_{\pm 0.19}$
Ours	$\mathbf{0.09}_{\pm 0.01}$

we use the Chamfer loss to compute the loss between predicted and target set in this experiment.

Results Table 6.1 shows that our model improves over the two baselines. In Figure 6.3 and Figure 6.4, we show the progression of $\hat{Y}$ throughout the minimisation with $\hat{Y}^{(10)}$ as the final prediction, the ground-truth set, and the baseline prediction of an MLP decoder. Observe how every optimisation starts with the same set $\hat{Y}^{(0)}$, but is transformed differently depending on the gradient of g_{enc}. Through this minimisation of L_{repr} by the inner optimisaton, the set is gradually changed into a shape that closely resembles the correct digit.

The types of errors of our model and the baseline are different, despite the use of models with similar losses in Figure 6.3. Errors in our model are mostly due to scattered points outside of the main shape of the digit, which is particularly visible in the third row. We believe that this is due to the limits of the encoder used: an encoder that is not powerful enough maps the slightly different sets to the same representation, so there is no L_{repr} gradient to work with. It still models the general shape accurately, but misses the fine details of these scattered points. The MLP decoder has less of this scattering, but makes mistakes in the shape of the digit instead. For example, in the third row, the baseline has a different curve at the top and a shorter line at the bottom. This difference in types of errors is also present in the extended examples in Figure 6.4.

Note that reconstructions shown in (Subsection 5.6.2) for the same auto-encoding task appear better because the decoder there uses additional information outside of the latent space: multiple $n \times n$ matrices are copied from the encoder into the decoder. In contrast, all information about the set is completely contained in our permutation-invariant latent space.

6.5.2 Bounding box prediction

Next, we turn to the task of object detection on the CLEVR dataset [53], which contains 70,000 training and 15,000 validation images. The goal is to predict the set of bounding boxes for the objects in an image. The target set contains at most 10 elements with 4 dimensions each: the (normalised) x-y coordinates of the top-left and bottom-right corners of each box. As the dataset does not contain bounding box information canonically, we use the processing method by Desta et al. [28] to calculate approximate bounding boxes. This causes the ground-truth bounding boxes to not always be perfect, which is a source of noise.

Model We encode the image with a ResNet34 [42] into a 512d feature vector, which is fed into the set decoder. The set decoder predicts the set of bounding boxes *from this single feature vector* describing the whole image. This is in contrast to existing region proposal networks [86] for bounding box prediction where the use of the entire feature map is required for the typical anchor-based approach. As the set encoder in our model, we use a 2-layer relation network [89] with FSPool (Chapter 5) as pooling. This is stronger than the FSPool-only model (without RN) we used in the MNIST experiment. We again compare this against a baseline that uses an MLP or LSTM as set decoder (matching AE-EMD [1] and [87] for the MLP decoder, [96] for the LSTM decoder). Since the sets are much smaller compared to our MNIST experiments, we can use the Hungarian loss as set loss. We perform no post-processing (such as non-maximum suppression) on the predictions of the model. The whole model is trained end-to-end.

Results We show our results in Table 6.2 using the standard average precision (AP) metric used in object detection with sample predictions in Figure 6.5 and Figure 6.6. Our model is able to very accurately localise the objects with high AP scores even when the intersection-over-union (IoU) threshold for a predicted box to match a ground truth box is very strict. In particular, our model using 10 iterations (the same it was trained with) has much better AP_{95} and AP_{98} than the baselines. The shown baseline model can predict bounding boxes in the close vicinity of objects, but fails to place the bounding box precisely on the object.

Figure 6.4: Progression of set prediction algorithm on MNIST.

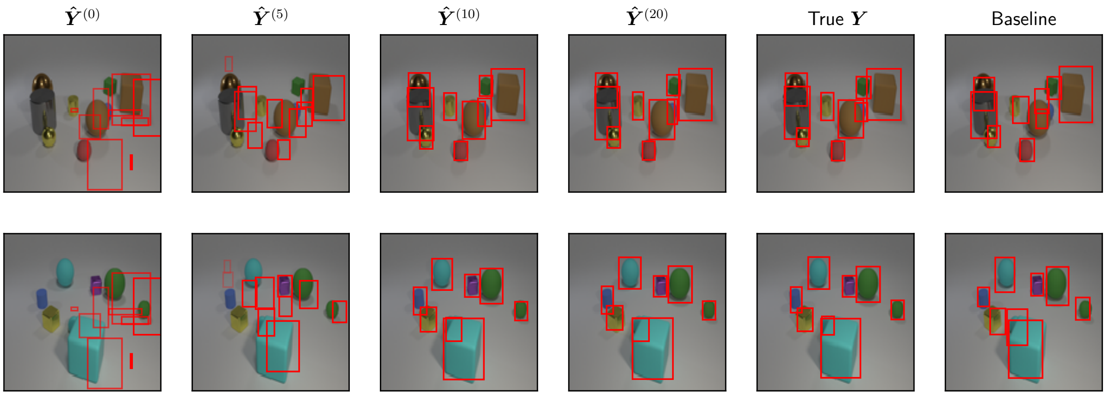


Figure 6.5: Progression of set prediction algorithm for bounding boxes in CLEVR. The shown MLP baseline sometimes struggles with heavily-overlapping objects and often fails to centre the object in the boxes.

Table 6.2: Average Precision (AP) for different intersection-over-union thresholds for a predicted bounding box to be considered correct. Higher is better. Mean and standard deviation over 6 runs.

Model	AP_{50}	AP_{90}	AP_{95}	AP_{98}	AP_{99}
MLP baseline	$99.3_{\pm 0.2}$	$94.0_{\pm 1.9}$	$57.9_{\pm 7.9}$	$0.7_{\pm 0.2}$	$0.0_{\pm 0.0}$
RNN baseline	$99.4_{\pm 0.2}$	$94.9_{\pm 2.0}$	$65.0_{\pm 10.3}$	$2.4_{\pm 0.0}$	$0.0_{\pm 0.0}$
Ours (10 iters)	$98.8_{\pm 0.3}$	$94.3_{\pm 1.5}$	$85.7_{\pm 3.0}$	$\mathbf{34.5}_{\pm 5.7}$	$\mathbf{2.9}_{\pm 1.2}$
Ours (20 iters)	$\mathbf{99.8}_{\pm 0.0}$	$\mathbf{98.7}_{\pm 1.1}$	$\mathbf{86.2}_{\pm 7.2}$	$24.3_{\pm 8.0}$	$1.4_{\pm 0.9}$
Ours (30 iters)	$\mathbf{99.8}_{\pm 0.1}$	$96.7_{\pm 2.4}$	$75.5_{\pm 12.3}$	$17.4_{\pm 7.7}$	$0.9_{\pm 0.7}$

This is visible from the decent performance for low IoU thresholds, but bad performance for high IoU thresholds.

We can also run our model with more inner optimisation steps than the 10 it was trained with. Many results improve when doubling the number of steps, which shows that further minimisation of $L_{\text{repr}}(\hat{Y}, \boldsymbol{z})$ is still beneficial, even if it is unseen during training. The model “knows” that its prediction is still suboptimal when L_{repr} is high and also how to change the set to decrease it. This confirms that the optimisation is reasonably stable and does not diverge significantly with more steps. Being able to change the number of steps allows for a dynamic trade-off between prediction quality and inference time depending on what is needed for a given task.

The less-strict AP metrics (which measure large mistakes) improve with more iterations, while the very strict AP_{98} and AP_{99} metrics consistently worsen. This is a sign that the inner optimisation learned to reach its best prediction at exactly 10 steps, but slightly overshoots when run for longer. The model has learned that it does not fully converge with 10 steps, so it is compensating for that by slightly biasing the inner optimisation to get a better 10 step prediction. This is at the expense of the strictest AP metrics worsening with 20 steps, where this bias is not necessary anymore.

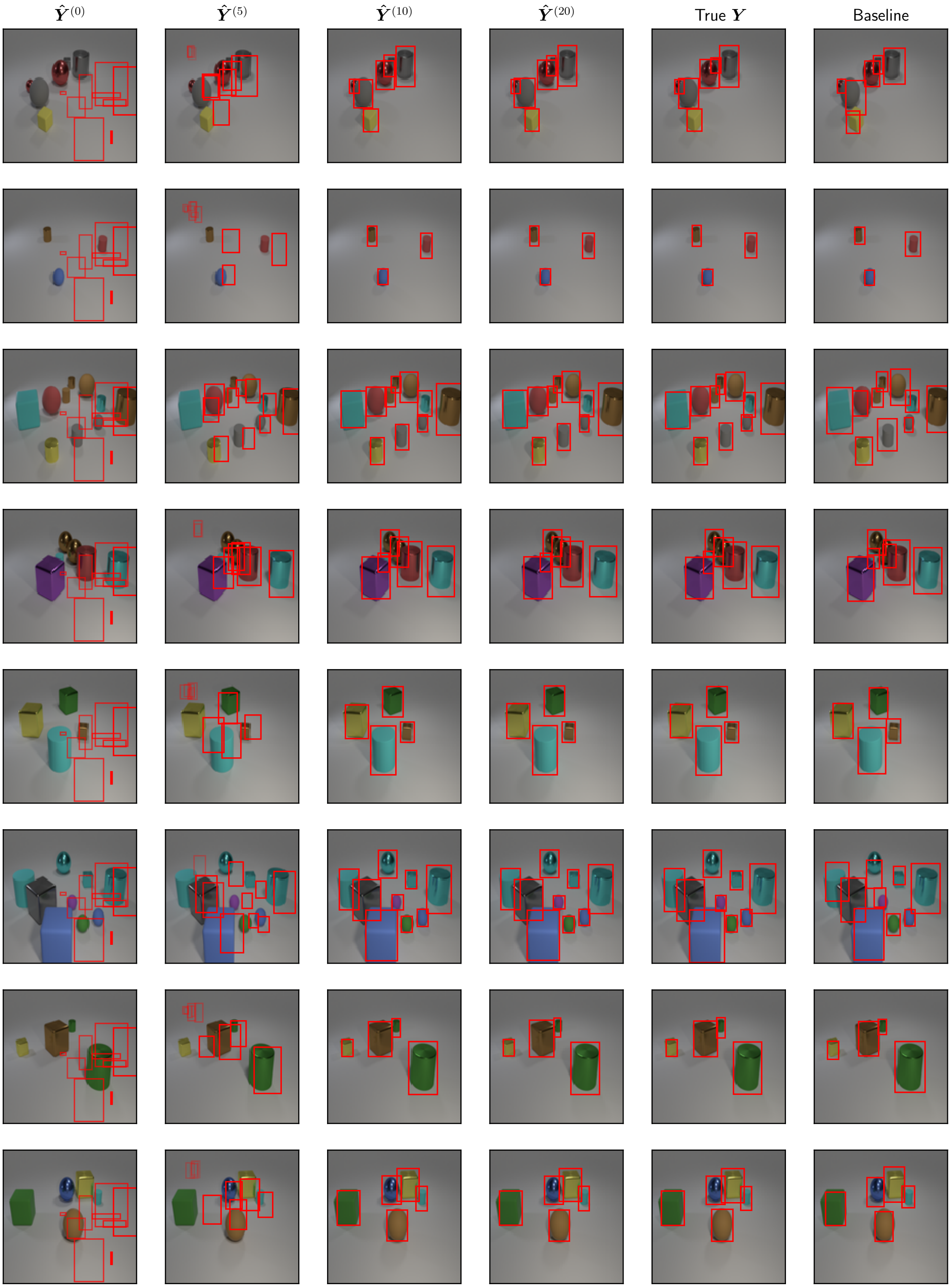


Figure 6.6: Progression of set prediction algorithm on CLEVR bounding boxes.

Table 6.3: Ablation experiments with DSPN-RN-Sum (standard RN model) and DSPN-RN-Max. The DSPN-RN-FSPool results are the same as in the previous table.

Model	AP_{50}	AP_{90}	AP_{95}	AP_{98}	AP_{99}
DSPN-RN-FSPool (10 iters)	$98.8_{\pm 0.3}$	$94.3_{\pm 1.5}$	$85.7_{\pm 3.0}$	$\mathbf{34.5}_{\pm 5.7}$	$\mathbf{2.9}_{\pm 1.2}$
DSPN-RN-FSPool (20 iters)	$\mathbf{99.8}_{\pm 0.0}$	$\mathbf{98.7}_{\pm 1.1}$	$\mathbf{86.2}_{\pm 7.2}$	$24.3_{\pm 8.0}$	$1.4_{\pm 0.9}$
DSPN-RN-FSPool (30 iters)	$\mathbf{99.8}_{\pm 0.1}$	$96.7_{\pm 2.4}$	$75.5_{\pm 12.3}$	$17.4_{\pm 7.7}$	$0.9_{\pm 0.7}$
DSPN-RN-Sum (10 iters)	$88.3_{\pm 3.7}$	$43.4_{\pm 14.4}$	$10.0_{\pm 7.4}$	$0.1_{\pm 0.1}$	$0.0_{\pm 0.0}$
DSPN-RN-Sum (20 iters)	$87.2_{\pm 3.0}$	$42.9_{\pm 11.9}$	$5.7_{\pm 3.5}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$
DSPN-RN-Sum (30 iters)	$79.0_{\pm 11.9}$	$32.5_{\pm 12.4}$	$3.4_{\pm 2.2}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$
DSPN-RN-Max (10 iters)	$68.0_{\pm 4.3}$	$4.0_{\pm 2.2}$	$0.1_{\pm 0.1}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$
DSPN-RN-Max (20 iters)	$66.6_{\pm 4.5}$	$3.3_{\pm 1.8}$	$0.1_{\pm 0.0}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$
DSPN-RN-Max (30 iters)	$64.1_{\pm 5.0}$	$2.3_{\pm 1.1}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$

We also perform an ablation experiment where we replace FSPool in the encoder with sum or max pooling. The results in Table 6.3 show that using FSPool greatly benefits our algorithm, likely due to the better representation that it is able to learn. The representation of the DSPN-RN-FSPool model is good enough that iterating our algorithm for more steps than the model was trained with can benefit the prediction, while for the baselines it generally only worsens.

Bear in mind that we do not intend to directly compete against traditional object detection methods. Our goal is to demonstrate that our model can accurately predict a set from a single feature vector, which is of general use for set prediction tasks not limited to image inputs.

6.5.3 Object attribute prediction

Lastly, we want to directly predict the full state of a scene from images on CLEVR. This is the set of objects with their position in the 3d scene (xyz coordinates), shape (sphere, cylinder, cube), colour (eight colours), size (small, large), and material (metal/shiny, rubber/matte) as features. For example, an object can be a "small cyan metal cube" at position (0.95, -2.83, 0.35). We encode the categorial features as one-hot vectors and concatenate them into an 18d feature vector for each object. Note that we do not use bounding box information, so the model has to implicitly learn which object in the image corresponds to which set element with the associated properties. This makes it different from usual object detection tasks, since bounding boxes are required for traditional object detection models that rely on anchors.

Model We use exactly the same model as for the bounding box prediction in the previous experiment with all hyperparameters kept the same. The only difference is that it now outputs 18d instead of 4d set elements. For simplicity, we continue using the Hungarian loss with the

Table 6.4: Average Precision (AP) in % for different distance thresholds of a predicted set element to be considered correct. AP_∞ only requires all attributes to be correct, regardless of 3d position. Higher is better. Mean and standard deviation over 6 runs.

Model	AP_∞	AP_1	$AP_{0.5}$	$AP_{0.25}$	$AP_{0.125}$
MLP baseline	$3.6_{\pm 0.5}$	$1.5_{\pm 0.4}$	$0.8_{\pm 0.3}$	$0.2_{\pm 0.1}$	$0.0_{\pm 0.0}$
RNN baseline	$4.0_{\pm 1.9}$	$1.8_{\pm 1.2}$	$0.9_{\pm 0.5}$	$0.2_{\pm 0.1}$	$0.0_{\pm 0.0}$
Ours (10 iters)	$72.8_{\pm 2.3}$	$59.2_{\pm 2.8}$	$39.0_{\pm 4.4}$	$12.4_{\pm 2.5}$	$1.3_{\pm 0.4}$
Ours (20 iters)	$84.0_{\pm 4.5}$	$80.0_{\pm 4.9}$	$\mathbf{57.0}_{\pm 12.1}$	$\mathbf{16.6}_{\pm 9.0}$	$\mathbf{1.6}_{\pm 0.9}$
Ours (30 iters)	$\mathbf{85.2}_{\pm 4.8}$	$\mathbf{81.1}_{\pm 5.2}$	$47.4_{\pm 17.6}$	$10.8_{\pm 9.0}$	$0.6_{\pm 0.7}$

Huber loss as pairwise cost, as opposed to switching to cross-entropy for the categorical features.

Results We show our results in Table 6.4 and give sample outputs in Table 6.6. The evaluation metric is the standard average precision as used in object detection, with the modification that a prediction is considered correct if there is a matching groundtruth object with exactly the same properties and within a given Euclidean distance of the 3d coordinates. Our model clearly outperforms the baselines. This shows that our model is also suitable for modeling high-dimensional set elements.

When evaluating with more steps than our model was trained with, the difference in the more lenient metrics improves even up to 30 iterations. This time, the results for 20 iterations are all better than for 10 iterations. This suggests that 10 steps is too few to reach a good solution in training, likely due to the higher difficulty of this task compared to the bounding box prediction. Still, the representation $\boldsymbol{z}$ that the input encoder produces is good enough such that minimising L_{repr} more at evaluation time leads to better results. When going up to 30 iterations, the result for predicting the state only (excluding 3d position) improves further, but the accuracy of the 3d position worsens. We believe that this is again caused by overshooting the target due to the bias of training the model with only 10 iterations.

The results of the ablation experiment where we replace FSPool in the encoder with sum or max pooling are shown in Table 6.5. This time, the difference between FSPool and the other encoders is even larger than for the bounding box experiments. These results confirm that a better set encoder leads to better prediction results. They also show that a way to improve RNs is to simply replace the sum pooling with FSPool.

Table 6.5: Average Precision (AP, mean ± stdev) for different distance thresholds of the predicted state descriptions over 6 runs. All results are worse than the DSPN-RN-FSPool in the previous table. The DSPN-RN-FSPool results are the same as in the previous table.

Model	AP_∞	AP_1	$AP_{0.5}$	$AP_{0.25}$	$AP_{0.125}$
DSPN-RN-FSPool (10 iters)	$72.8_{\pm 2.3}$	$59.2_{\pm 2.8}$	$39.0_{\pm 4.4}$	$12.4_{\pm 2.5}$	$1.3_{\pm 0.4}$
DSPN-RN-FSPool (20 iters)	$84.0_{\pm 4.5}$	$80.0_{\pm 4.9}$	$\mathbf{57.0}_{\pm 12.1}$	$\mathbf{16.6}_{\pm 9.0}$	$\mathbf{1.6}_{\pm 0.9}$
DSPN-RN-FSPool (30 iters)	$\mathbf{85.2}_{\pm 4.8}$	$\mathbf{81.1}_{\pm 5.2}$	$47.4_{\pm 17.6}$	$10.8_{\pm 9.0}$	$0.6_{\pm 0.7}$
DSPN-RN-Sum (10 iters)	$44.6_{\pm 3.8}$	$21.9_{\pm 4.8}$	$7.1_{\pm 2.7}$	$1.0_{\pm 0.5}$	$0.0_{\pm 0.0}$
DSPN-RN-Sum (20 iters)	$39.6_{\pm 5.4}$	$15.2_{\pm 6.4}$	$3.0_{\pm 2.2}$	$0.3_{\pm 0.3}$	$0.0_{\pm 0.0}$
DSPN-RN-Sum (30 iters)	$30.2_{\pm 9.2}$	$7.1_{\pm 3.8}$	$0.9_{\pm 0.8}$	$0.1_{\pm 0.1}$	$0.0_{\pm 0.0}$
DSPN-RN-Max (10 iters)	$3.0_{\pm 0.2}$	$0.9_{\pm 0.1}$	$0.5_{\pm 0.2}$	$0.1_{\pm 0.1}$	$0.0_{\pm 0.0}$
DSPN-RN-Max (20 iters)	$3.1_{\pm 0.1}$	$1.2_{\pm 0.1}$	$0.8_{\pm 0.2}$	$0.3_{\pm 0.2}$	$0.0_{\pm 0.0}$
DSPN-RN-Max (30 iters)	$3.1_{\pm 0.1}$	$1.2_{\pm 0.1}$	$0.9_{\pm 0.2}$	$0.3_{\pm 0.2}$	$0.0_{\pm 0.0}$

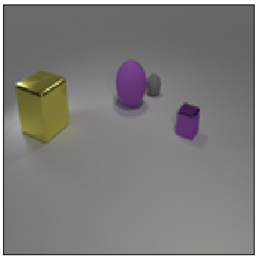
6.6 Discussion

In this chapter we showed how to predict sets with a deep neural network in a way that respects the set structure of the problem. We demonstrated in our experiments that this works for small (size 10) and large sets (up to size 342), as well as low-dimensional (2d) and higher-dimensional (18d) set elements. Our model is consistently better than the baselines across all experiments by predicting sets properly, rather than predicting a list and pretending that it is a set.

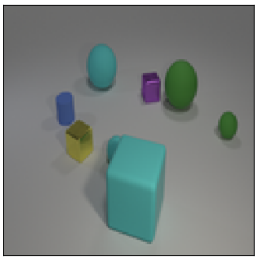
The improved results of our approach come at a higher computational cost. Each evaluation of the network requires time for $O(T)$ passes through the set encoder, which makes training take about 75% longer on CLEVR with $T = 10$. Keep in mind that this only involves the set encoder (which can be fairly small), not the input encoder (such as a CNN or RNN) that produces the target $\boldsymbol{z}$. Further study into representationally-powerful and efficient set encoders such as RN [89] and FSPool (Chapter 5) – which we found to be critical for good results in our experiments – would be of considerable interest, as it could speed up the convergence and thus inference time of our method. Another promising approach is to better initialise $Y^{(0)}$ – perhaps with an MLP – so that the set needs to be changed less to minimise L_{repr}. Our model would act as a set-aware refinement method of the MLP prediction. Lastly, stopping criteria other than iterating for a fixed 10 steps can be used, such as stopping when $L_{\text{repr}}(g_{\text{enc}}(\hat{Y}), \boldsymbol{z})$ is below a fixed threshold: this would stop when the encoder thinks $\hat{Y}$ is of a certain quality corresponding to that threshold.

Our algorithm may be suitable for generating samples under other invariance properties. For example, we may want to generate images of objects where the rotation of the object does not matter (such as aerial images). Using our decoding algorithm with a rotation-invariant image

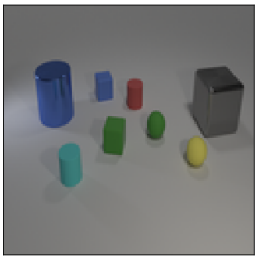
Table 6.6: Progression of set prediction algorithm on CLEVR state prediction. Red text denotes a wrong attribute. Objects are sorted by x coordinate, so they are sometimes misaligned with wrongly-coloured red text (see third example: red entries in $\hat{Y}^{(20)}$).



$\hat{Y}^{(5)}$	$\hat{Y}^{(10)}$	$\hat{Y}^{(20)}$	True Y	Baseline
(-0.14, 1.16, 3.57)	(-2.33, -2.41, 0.73)	(-2.33, -2.42, 0.78)	(-2.42, -2.40, 0.70)	(-1.65, -2.85, 0.69)
large purple rubber sphere	large yellow metal cube	large yellow metal cube	large yellow metal cube	large yellow metal cube
(0.01, 0.12, 3.42)	(-1.20, 1.27, 0.67)	(-1.21, 1.20, 0.65)	(-1.18, 1.25, 0.70)	(-0.95, 1.08, 0.68)
large gray metal cube	large purple rubber sphere	large purple rubber sphere	large purple rubber sphere	large green rubber sphere
(0.67, 0.65, 3.38)	(-0.96, 2.54, 0.36)	(-0.96, 2.59, 0.36)	(-1.02, 2.61, 0.35)	(-0.40, 2.14, 0.35)
small purple metal cube	small gray rubber sphere	small gray rubber sphere	small gray rubber sphere	small red rubber sphere
(0.67, 1.14, 2.96)	(1.61, 1.57, 0.36)	(1.58, 1.62, 0.38)	(1.74, 1.53, 0.35)	(1.68, 1.77, 0.35)
small purple rubber sphere	small yellow metal cube	small purple metal cube	small purple metal cube	small brown metal cube



$\hat{Y}^{(5)}$	$\hat{Y}^{(10)}$	$\hat{Y}^{(20)}$	True Y	Baseline
(-0.29, 1.14, 3.73)	(-2.78, 0.86, 0.72)	(-2.62, 0.83, 0.68)	(-2.88, 0.78, 0.70)	(-2.42, 0.63, 0.71)
small purple metal cube	large cyan rubber sphere	large cyan rubber sphere	large cyan rubber sphere	large purple rubber sphere
(-0.11, -0.37, 3.65)	(-2.17, -1.59, 0.38)	(-2.12, -1.58, 0.49)	(-2.14, -1.63, 0.35)	(-2.40, -2.07, 0.35)
small brown metal cube	small blue rubber cylinder	small blue rubber cylinder	small blue rubber cylinder	small green rubber cylinder
(0.08, 0.56, 3.84)	(-0.45, 2.19, 0.40)	(-0.60, 2.23, 0.29)	(-0.78, 1.97, 0.35)	(-0.74, 2.46, 0.33)
large cyan rubber cube	small purple metal cube	small purple metal cube	small purple metal cube	small cyan metal cube
(0.69, -0.43, 3.55)	(-0.14, -2.15, 0.38)	(-0.30, -1.99, 0.32)	(-0.38, -2.06, 0.35)	(0.30, -1.86, 0.34)
small brown rubber sphere	small yellow metal cube	small yellow metal cube	small yellow metal cube	small gray rubber sphere
(1.12, 0.21, 3.83)	(0.53, 2.56, 0.70)	(0.27, 2.46, 0.72)	(0.42, 2.56, 0.70)	(0.69, -2.10, 0.36)
large cyan rubber cube	large green rubber sphere	large green rubber sphere	large green rubber sphere	small red metal cube
(1.23, -0.25, 3.58)	(0.93, -1.41, 0.35)	(0.86, -1.31, 0.27)	(0.81, -1.30, 0.35)	(1.12, 2.28, 0.70)
small cyan rubber sphere	small cyan rubber sphere	small cyan rubber sphere	small cyan rubber sphere	large cyan rubber sphere
(1.73, 1.04, 3.57)	(2.50, -2.08, 0.76)	(2.64, -2.05, 0.76)	(2.56, -1.94, 0.70)	(2.55, -2.26, 0.73)
small cyan rubber sphere	large cyan rubber cube	large cyan rubber cube	large cyan rubber cube	large yellow rubber cube
(2.06, 1.94, 3.81)	(2.61, 2.59, 0.33)	(2.75, 2.73, 0.35)	(2.74, 2.64, 0.35)	(2.99, 2.59, 0.35)
large brown rubber sphere	small green rubber sphere	small green rubber sphere	small green rubber sphere	small purple rubber sphere



$\hat{Y}^{(5)}$	$\hat{Y}^{(10)}$	$\hat{Y}^{(20)}$	True Y	Baseline
(0.22, 0.12, 3.47)	(-2.76, -1.42, 0.68)	(-2.68, -1.64, 0.77)	(-2.62, -1.76, 0.70)	(-2.47, -1.73, 0.70)
small brown rubber cube	large blue metal cylinder	large blue metal cylinder	large blue metal cylinder	large cyan metal cylinder
(0.41, 0.11, 3.77)	(-1.56, -0.61, 0.35)	(-2.43, 0.03, 0.34)	(-2.29, 0.49, 0.35)	(-2.42, 0.09, 0.36)
large gray metal cube	small blue rubber cylinder	small blue rubber cube	small blue rubber cube	small blue rubber cylinder
(0.50, 0.44, 3.61)	(-1.08, 0.23, 0.33)	(-1.00, 1.18, 0.33)	(-0.93, 1.15, 0.35)	(-1.24, 1.16, 0.36)
small gray rubber cube	small green rubber cube	small red rubber cylinder	small red rubber cylinder	small red rubber cube
(0.83, 0.53, 3.45)	(-0.07, 0.97, 0.36)	(-0.01, -1.00, 0.46)	(0.28, -2.84, 0.35)	(0.39, 0.20, 0.33)
small cyan rubber sphere	small green rubber cylinder	small green rubber cube	small cyan rubber cylinder	small red rubber sphere
(0.86, 0.85, 3.50)	(0.28, -2.44, 0.49)	(0.21, -2.88, 0.40)	(0.29, -0.98, 0.35)	(0.56, -3.11, 0.35)
small gray rubber sphere	small cyan rubber cylinder	small cyan rubber cylinder	small green rubber cube	small yellow rubber cylinder
(1.86, 2.34, 3.80)	(1.36, -0.63, 0.38)	(0.99, 0.17, 0.37)	(0.92, 0.54, 0.35)	(0.90, 0.64, 0.35)
large gray metal cube	small green rubber sphere	small green rubber sphere	small green rubber sphere	small green rubber sphere
(1.97, 0.55, 3.61)	(2.01, 3.07, 0.65)	(1.97, 2.89, 0.39)	(2.04, 2.78, 0.70)	(2.39, 0.27, 0.36)
small green rubber sphere	large gray metal cube	large gray metal cube	large gray metal cube	small yellow rubber sphere
	(2.69, 0.63, 0.34)	(2.87, 0.51, 0.25)	(2.70, 0.67, 0.35)	(2.44, 2.55, 0.68)
	small yellow rubber sphere	small yellow rubber sphere	small yellow rubber sphere	large gray metal cube

encoder could predict images without forcing the model to choose a fixed orientation of the image, which could be a useful inductive bias.

In conclusion, we are excited about enabling a wider variety of set prediction problems to be tackled with deep neural networks. Our main idea should be readily extensible to similar domains such as graphs to allow for better graph prediction, for example molecular graph generation or end-to-end scene graph prediction from images. We hope that our model inspires further research into graph generation, stronger object detection models, and – more generally – a more principled approach to set prediction.

Chapter 7

Future work

In this thesis, we have developed a variety of techniques for modeling sets with deep neural networks. All of our proposed methods are fully differentiable, which makes them readily usable in existing and new neural networks for sets.

Starting from our initial work on the specialised task of counting object proposals in visual question answering (Chapter 3), we developed two building blocks for set encoders and two building blocks for set decoders. Our set encoders tackle the bottleneck problem (Subsection 2.3.3) of traditional approaches, which only use simple pooling functions like sum and max pooling. By learning how to permute the set elements with the Permutation-optimisation module (Chapter 4) or sorting the features within a set independently with FSPool (Chapter 5), the set is turned into an ordered representation, which is much easier to work with.

In Chapter 5, we also realised that existing approaches for predicting sets suffer from a responsibility problem (Section 5.3), which forces a set decoder to predict discontinuous outputs. Since we show that this can majorly hinder successful learning (Subsection 5.6.1), we tried to find ways to avoid this problem. We first did this through FSUnpool in the limited auto-encoder setting, then through Deep Set Prediction Networks (DSPN) in the much more general supervised prediction setting (Chapter 6). This is perhaps the most significant contribution of this thesis, since it is the first model for predicting sets that has the right properties for sets while being able to scale to complex tasks like object detection.

Of course, many open problems remain in the area of set encoders and set decoders, which we discuss in Section 7.1 and Section 7.2. Last but not least, we discuss the potential of latent sets (Section 7.3), which we would have loved to work on.

7.1 Set encoders

A better encoder can learn better representations and therefore should also result in better performance for the downstream task. This can even extend to improvements to set prediction through DSPN. Recent progress in set encoders has been closely linked to better modeling of *relationships* between set elements and we believe that this will continue to play an important role. Due to the similarity between the domains of sets and graphs, improvements in either area are easily transferable, so researchers in either field should be closely aware of the other.

The permutation-invariant step We have primarily used the idea of ordering the set in some way in Chapter 4 and Chapter 5. There may be other algorithms similar to sorting that are able to turn a set into a list in a way that is easy to learn from. An important consideration is the trade-off between the complexity of relationships within the set that can be modeled and the computational efficiency of that method. Developing new set pooling methods with different trade-offs and theoretically characterising the limitations like in Murphy et al. [76], Xu et al. [105], and Wagstaff et al. [103] are therefore both useful.

The permutation-equivariant step We have explored various equivariant approaches in this thesis: the FSPool-FSUnpool combination (Chapter 5) and backpropagating through a set encoder (Chapter 6). While we used them in the context of predicting sets, it would be interesting to see how well these work as a building block in a pure set encoder or in per-element set prediction tasks.

What about other permutation-equivariant ways to propagate information within the set? A recent major advance is the building block of self-attention in set transformers [100], which has been successfully used in contexts other than sets such as language modeling. Here, the same trade-off between complexity of relations and efficiency exists. Methods that strike a good balance between the two are needed, and gaining deeper theoretical understanding of this balance may help in this regard.

Applications As we have pointed out in Section 2.5, various models that already use a sum or max – such as pooling in CNNs – allow for set methods to be used as an alternative. In those cases, lessons from the set literature can be applied to understand the existing models and potentially improve them.

For language modeling, set transformers have the benefit over traditional sequential models of being able to efficiently model long-distance relationships (words that are far apart). To not lose information about the order of the words in a sentence when going from the sequence of words to the set of words, the words are augmented with positional information. In essence, the existing sequence problem is transformed into an equivalent set problem. Due to the versatility of thinking about problems in terms of sets, there is potential in transforming other problems into sets and using set-based approaches on them.

7.2 Set decoders

Predicting sets in new ways is an especially promising direction due to our contributions of the responsibility problem, FSPool-FSUnpool, and DSPN. One option is to make improvements to our DPSN algorithm like the ones we suggest in Section 6.6. There are also various other research directions for set decoders.

Set losses While the Hungarian loss seems ideal from an optimal transport standpoint, the computational complexity of $\Theta(n^3)$ is not good enough for large sets. Alternatives like Sinkhorn-based methods [35] or something else entirely [11] could prove to be just as good in terms of quality while being much faster and more easily parallelisable. There is also potential in the use of k-dimensional (k-d) trees to reduce computation. This would allow for prediction of much larger sets and thus greater applicability of set decoders.

Other approaches We argued throughout the thesis that it is important to avoid the responsibility problem. While we showed one solution with our DSPN model and its iterative optimisation approach, other ways to avoid it may exist. One potential direction is to find a more lightweight method than backpropagating through a set encoder to determine the update to the set, perhaps with a self-attentive decoder [100, 14]. It would also be interesting to find a non-iterative approach, since the iteration is a core component of DSPN.

7.2.1 Applications

There are a variety of existing tasks in machine learning where the output is a set, but none of the models for the task treat it as a set prediction problem. This is particularly evident in Computer Vision with tasks like object detection, instance segmentation, and multi-object tracking. All of these share the commonality of being about *multiple objects*, for which a set would be ideal. Models with good performances on these tasks exist, but they usually rely on various post-processing heuristics like thresholding and non-maximum suppression to produce a set of outputs. By using a proper model for predicting sets like our DSPN, results in these tasks can potentially be greatly improved.

Object detection Of course, this is not without its challenges. We will take the Faster R-CNN [86] as representative example of traditional object detectors. Our DSPN model does not make any assumptions about its input feature vector, while Faster R-CNN assumes that the input feature map comes from an image encoder. While this has the benefit of DSPN being applicable to non-image tasks, it also means that DSPN makes *global* predictions: the existence of an object in one corner of an image can affect the entire feature vector and therefore how a different object is predicted in a different corner. This is something that is often undesirable. The convolutional approach to predicting the anchors in Faster R-CNN means that its predictions are reasonably translation-invariant and mostly local. An object in one part of the image rarely affects an object in a completely different part.

Another potential challenge is how the model handles noise in the target sets. We have only tested DSPN on a synthetic dataset where perfect information about the scene is available, which is unrealistic for datasets with real images and human annotators. We currently do not know how robust our method is to noisy labels. A missing object in the labeling could again affect the entire input feature vector and therefore the quality of the entire prediction.

A hybrid approach that combines DSPN with Faster R-CNN and similar object detection architectures could combine the benefits of the two: proper set prediction with end-to-end training whilst maintaining reasonable translation-invariance.

Instance segmentation Instance segmentation models typically use a two-stage approach: first detect the objects, then segment each object individually. By using our DSPN model, this can be turned into a single stage of predicting the set of masks, each mask corresponding to the segmentation of one object. By applying a softmax function on every pixel of these masks across the set, we could ensure that each pixel in the image belongs to at most one object or the background. This is arguably a more elegant approach to instance segmentation, which could also lead to better results since it can be trained end-to-end.

Graph prediction A domain that is very similar to sets is the domain of graphs. With the set of nodes and the set of edges, or the set of nodes and – for each node – the set of neighbours, the connection to sets is quite clear. Graph prediction methods have also only relied on MLPs and RNNs so far, so the responsibility problem and our DSPN method to avoid it are directly applicable. In particular, replacing the set encoder in our model with a graph encoder should immediately enable the prediction of graphs. The main problem here would be the choice of loss function between graphs, which in the most difficult case would have to solve the graph isomorphism problem. Simonovsky et al. [93] have used such a graph loss before, which has a large $\Theta(n^4)$ time complexity in the number of nodes. Finding good, efficient graph losses is perhaps the bigger challenge with predicting graphs.

7.3 Latent sets in neural networks

Lastly, it would be fascinating to have sets used in more contexts than in neural networks specifically for sets. This is a more speculative idea.

An interesting aspect about sets is that they are a unique combination between a symbolic, object-based representation (set elements are discrete and each set element corresponds to a different symbol or object), while also being grounded (each set elements is associated with a continuous, learned feature vector describing it). Humans are able to think about objects in terms of its smaller parts, like a hand being made of five fingers and a palm. By letting a traditional neural network like an image classifier use sets of feature vectors as latent representation, we may start to see such object- and parts-based representations emerge. Instead of modeling an image of a hand as a feature vector, it could learn to model it as a set of five fingers and a palm, or other decompositions of the hand into smaller parts. We know from Chapter 3 that an object-based representation can be much easier to work with, so being able to discover such a representation automatically through the set structure would be quite useful. In some ways, this is an extension of the parts-based representation idea that Capsule Neural Networks [88] try to achieve, without forcing the notion of pose in Capsule Nets onto the neural network.

Because sets are in this unique position between connectionist and symbolic Artificial Intelligence, they have the potential to combine the best of both worlds. We hope that our thesis provides a stepping stone towards this goal.

Appendix A

Experimental details

A.1 Counting in visual question answering

Here, we detail our improved baseline model for visual question answering in Chapter 3. We provide the corresponding source code to reproduce our experiments at `https://github.com/Cyanogenoid/vqa-counting`.

The most significant change to the baseline model [54] that we make is the use of object proposal features by Anderson et al. [4] as previously mentioned. The following tweaks were made without considering the performance impact on the counting module; only the validation accuracy of the baseline was optimised. Details not mentioned here can be assumed to be the same as in their paper.

To fuse vision features $\boldsymbol{x}$ and question features $\boldsymbol{y}$, the baseline concatenates and linearly projects them, followed by a ReLU activation. This is equivalent to $\mathrm{ReLU}(\boldsymbol{W}_x\boldsymbol{x} + \boldsymbol{W}_y\boldsymbol{y})$. We include an additional term that measures how different the projected $\boldsymbol{x}$ is from the projected $\boldsymbol{y}$, changing the fusion mechanism to

$$\boldsymbol{x} \diamond \boldsymbol{y} = \mathrm{ReLU}(\boldsymbol{W}_x\boldsymbol{x} + \boldsymbol{W}_y\boldsymbol{y}) - (\boldsymbol{W}_x\boldsymbol{x} - \boldsymbol{W}_y\boldsymbol{y})^2 \tag{A.1}$$

The LSTM [44] for question encoding is replaced with a GRU [23] with the same hidden size with dynamic per-example unrolling instead of a fixed 14 words per question. We apply batch normalisation [49] before the last linear projection in the classifier to the 3000 classes. The learning rate is increased from 0.001 to 0.0015 and the batch size is doubled to 256. The model is trained for 100 epochs (1697 iterations per epoch to train on the training set, 2517 iterations per epoch to train on both training and validation sets) instead of 100,000 iterations, roughly in line with the doubling of dataset size when going from VQA v1 to VQA v2.

Note that this single-model baseline is regularised with dropout [95], while the other current top models skip this and rely on ensembling to reduce overfitting. This explains why our single-model baseline outperforms most single-model results of the state-of-the-art models. We found ensembling of the regularised baseline to provide a much smaller benefit in preliminary experiments compared to the results of ensembling unregularised networks reported in Teney et al. [98].

A.2 Permutation-optimisation

In this section, we describe the experimental set-up of Chapter 4 in detail. All of our experiments can be reproduced using our implementation at `https://github.com/Cyanogenoid/perm-optim` in PyTorch [82] through the `experiments/all.sh` script. For the former three experiments, we use the following hyperparameters throughout:

- Optimiser: Adam [58] (default settings in PyTorch: $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$)
- Initial step size η in inner gradient descent: 1.0

All weights are initialised with Xavier initialisation [36]. We choose the f within the ordering cost function F to be a small MLP. The input to f has 2 times the number of dimensions of each element, obtained by concatenating the pair of elements. This is done for all pairs that can be formed from the input set. This is linearly projected to some number of hidden units to which a ReLU activation is applied. Lastly, this is projected down to 1 dimension for sorting numbers and VQA, and 2 dimensions for assembling image mosaics (1 output for row-wise costs, 1 output for column-wise costs). These outputs are used for creating the ordering cost matrix $\boldsymbol{C}$.

A.2.1 Sorting numbers

- Inner gradient descent steps T: 6
- Adam learning rate: 0.1
- Batch size: 512
- Number of sets to sort in training set: 2^{18}
- Set sizes: 5, 10, 15, 80, 100, 120, 512, 1024
- Evaluation intervals: $[0, 1]$, $[0, 10]$, $[0, 1000]$, $[1, 2]$, $[10, 11]$, $[100, 101]$, $[1000, 1001]$ (same as in Mena et al. [71])
- F size of hidden dimension: 16

The ordering cost function F concatenates the two floats of each pair and applies a 2-layer MLP that takes the 2 inputs to 16 hidden units, ReLU activation, then to one output.

For evaluation, we switch to double precision floating point numbers. This is because for the interval $[1000, 1001]$, as the set size increases, there are not enough unique single precision floats in that interval for the sets to contain only unique floats with high probability (the birthday problem). Using double precision floats avoids this issue. Note that using single precision floats is enough for the other intervals and smaller set sizes, and training is always done on the interval [0, 1] at single precision.

A.2.2 Re-assembling image mosaics

- Adam learning rate: 10^{-3}
- Inner gradient descent steps T: 4
- Batch size: 32
- Training epochs: 20 (MNIST, CIFAR10) or 1 (ImageNet)
- F size of hidden dimension: 64 (MNIST, CIFAR10) or 128 (ImageNet)

For all three image datasets from which we take images (MNIST, CIFAR10, ImageNet), we first normalise the inputs to have zero mean and standard deviation one over the dataset as is common practice. For ImageNet, we crop rectangular images to be square by reducing the size of the longer side to the length of the shorter side (centre cropping). Images that are not exactly divisible by the number of tiles are first rescaled to the nearest bigger image size that is exactly divisible. Following Mena et al. [71], we process each tile with a 5×5 convolution with padding and stride 1, 2×2 max pooling, and ReLU activation. This is flattened into a vector to obtain the feature vector for each tile, which is then fed into our F. Unlike Mena et al. [71], we decide not to arbitrarily upscale MNIST images by a factor of two, even when upscaling results in slightly better performance in general.

While we were able to mostly reproduce their MNIST results, we were not able to reproduce their ImageNet results for the 3×3 case. In general, we observed that good settings for their model also improved the results of our PO-U and PO-LA models. Better hyperparameters than what we used should improve all models similarly while keeping the ordering of how well they perform the same.

This task is also known as jigsaw puzzle [79], but we decided on naming it image mosaics because the tiles are square which can lead to multiple solutions, rather than the typical unique solution in traditional jigsaw puzzles enforced by the different tile shapes.

A.2.3 Implicit permutations through classification

We use the same setting as for the image mosaics, but further process the output image with a ResNet-18. For MNIST and CIFAR10, we replace the first convolutional layer with one that has a 3×3 kernel size and no striding. This ResNet-18 is first trained on the original dataset for 20 epochs (1 for ImageNet), though images may be rescaled if the image size is not divisible by the number of tiles per side. All weights are then frozen and the permutation method is trained for 20 epochs (1 for ImageNet). As stated previously, this is necessary in order for the ResNet-18 to not use each tile individually and ignore the resulting artefacts from the permuted tiles. This is also one of the reasons why we downscale ImageNet images to 64×64 pixels. Because the resulting image tiles are so big while the receptive field of ResNet-18 is relatively small if we were to use 256×256 images, the permutation artefacts barely affect results because they are only a small fraction of the globally-pooled features. The permutation permutes each set of tiles, which are reconstructed (without use of the Hungarian algorithm) into an image, which is then processed by the ResNet-18.

We observed that the LinAssign model by Mena et al. [71] consistently results in NaN values after Sinkhorn normalisation in this set-up, despite our Sinkhorn implementation using the numerically-stable version of softmax with the exp-normalise trick. We avoided this issue by clipping the outputs of their model into the [-10, 10] interval before Sinkhorn normalisation. We did not observe these NaN issues with our PO-U model.

A.2.4 Visual question answering

We use the official implementation of BAN as baseline without changing any of the hyperparameters. We thus refer to [56] for details of their model architecture and hyperparameters. The only change to hyperparameters that we make is reducing the batch size from 256 to 112 due to the GPU memory requirements of the baseline model, even without our permutation mechanism.

The BAN model generates attention weights between all object proposals in a set and words of the question. We take the attention weight for a single object proposal to be the maximum attention weight for that proposal over all words of the question, the same as in their integration of the counting module. Each element of the set, corresponding to object proposals, is the concatenation of this attention logit, bounding box coordinates, and the feature vector projected from 2048 down to 8 dimensions. We found this projection necessary to not inhibit learning of the rest of the model, which might be due to gradient clipping or other hyperparameters that are no longer optimal in the BAN model. This set of object proposals is then permuted with $T = 3$ and a 2-layer MLP with hidden dimension 128 for f to produce the ordering costs. The elements in the permuted sequence are weighted by how relevant each proposal is (sigmoid of the corresponding attention logit) and the sequence is then fed into an LSTM with 128 units. The last cell state of the LSTM is the set representation which is projected, ReLUd, and added back into the hidden state of the BAN model. The remainder of the BAN model is now able to use information from this set representation. There are 8 attention glimpses, so we process each of these with a PO-U module and an LSTM with shared parameters across these 8 glimpses.

A.3 Featurewise sort pooling

In this section, we describe the experimental set-up of Chapter 5 in detail. We provide the code to reproduce all experiments at `https://github.com/Cyanogenoid/fspool`.

For almost 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 to $k = 5$ and $k = 40$. $\bar{\boldsymbol{W}}$ 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.

A.3.1 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 [58] 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 et al. [36]. 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). We have also tried much hidden and latent space sizes of 128 when we tried to get better results for the baselines.

A.3.2 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 Hungarian 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.

A.3.3 CLEVR

The architecture and hyperparameters come from the 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. For the Janossy Pooling baseline, we use the model configuration from Murphy et al. [76] that appeared best in their experiments, which uses π-SGD with an LSTM that has $|h|$ as neighbourhood size.

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 pooling method like 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.

A.3.4 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 [105] and we chose Linear-ReLU-BN-Linear-ReLU-BN in the end. We tried Linear-BN-ReLU-Linear-BN-ReLU

Table A.1: 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	28.8	19.2	28.8	30.4
- *batch size*	66.0	100	45.6	32.0	86.4	89.6	60.8	41.6	128
- *dropout*	0.25	0.15	0.35	0.10	0.40	0.15	0.35	0.20	0.50
GIN-Base									
- *dimensionality*	64.0	64.0	64.0	64.0	64.0	27.2	20.8	25.6	28.8
- *batch size*	86.4	93.2	72.8	100	100	70.4	60.8	60.8	128
- *dropout*	0.30	0.15	0.25	0.45	0.40	0.25	0.45	0.20	0.35

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 [31] 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 [36]. 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.

A.4 Deep set prediction networks

In this section, we describe the experimental set-up of Chapter 6 in detail. In our algorithm, η was chosen in initial experiments and we did not tune it beyond that. We did this by increasing η until the output set visibly changed between inner optimisation steps when the set encoder is randomly initialised. This makes it so that changing the set encoder weights has a noticeable effect rather than being stuck with $\hat{\boldsymbol{Y}}^{(T)} \approx \hat{\boldsymbol{Y}}^{(0)}$.

$T = 10$ was chosen because it seemed to be enough to converge to good solutions on MNIST. We simply kept this for the supervised experiments on CLEVR.

In the supervised experiments, we would often observe large spikes in training that cause the model diverge when $\lambda = 1$. By changing around various parameters, we found that reducing λ eliminated most of this issue and also made training converge to better solutions. Much smaller values than 0.1 converged to worse solutions. This is likely because the issue of not having the $L_{\text{repr}}(Y, \boldsymbol{z})$ term in the outer loss in the first place ($\lambda = 0$) is present again – see Subsection 6.3.3.

For all experiments, we used Adam with the default momentum values and batch size 32 for the outer optimisation. The only hyperparameter we tuned in the experiments is the learning rate of the outer optimisation. Every individual experiment is run on a single 1080 Ti GPU.

The MLP decoder baseline has 3 layers with 256 (MNIST) or 512 (CLEVR) neurons in the first two layers and the number of channels of the output set in the task in the third layer. The LSTM decoder linearly transforms the latent space into 256 (MNIST) or 512 (CLEVR) dimensions, which is used as initial cell state of the LSTM. The LSTM is run for the same number of steps as the maximum set size, and the outputs of these steps is each linearly transformed into the output dimensionality.

A.4.1 MNIST

For MNIST, we train our model and the baseline model for 100 epochs to make sure that they have converged. Both models have a 3-layer MLP with ReLU activations and 256, 256, and 64 neurons in the three layers. For simplicity, sets are padded to a fixed size for FSPool. FSPool has 20 pieces in its piecewise linear function. We tried learning rates in $\{1.0, 0.1, 0.03, 0.01, 0.003, 0.001, 0.0003, 0.0001, 0.00001\}$ and chose 0.01. For the baselines, none of the other learning rates performed significantly better than the one we chose.

The baselines are trained slightly differently to our model. They do not output mask values natively, so we have to train them with the mask values in the training target. In other words, they are trained to predict x coordinate, y coordinate, and the mask for each point. We found it crucial to explicitly add 1 to the mask in the baseline model for good results. Otherwise, many of the baseline outputs get stuck in the local optimum of predicting the (0, 0, 0) point and the output is too sparse.

A.4.2 CLEVR

We train our model and the baselines models for 100 epochs on the training set of CLEVR and evaluate on the validation set, since no ground-truth scene information is available for the test set. All images are resized to 128×128 resolution. The set encoder is a 2-layer Relation Network with ReLU activation between the two layers, wherein the sum pooling is replaced with FSPool. The two layers have 512 neurons each. Because we use the Hungarian loss instead of the Chamfer loss here, including the mask feature in the target set does not worsen results, so we include the mask target for both the baseline and our model for consistency. To tune the learning rate, we started with the learning rate found for MNIST and decreased it similarly-sized steps until the training accuracy after 100 epochs worsened. We settled on 0.0003 as learning rate for both the bounding box and the state prediction task. All other hyperparameters are kept the same as for MNIST. The ResNet34 that encodes the image is not pre-trained.

A.4.3 Fixed results

Due to a bug, the practical trick of computing the loss over all $\hat{Y}^{(t)}$ instead of only the last $\hat{Y}^{(T)}$ was not used for the experiments on CLEVR. Also, all evaluation numbers on CLEVR with bounding boxes are overestimates, but consistent across the models. These are minor problems that do not affect the conclusions of the experiments. The first bug was found by Sangwoo Mo [1], the second bug was found by Chao Feng [2]. We are very grateful to them for finding and letting us know of these bugs. The following tables show the results with the bugs fixed when using exactly the same hyperparameters as before:

- Table A.2 shows the results when only fixing the evaluation bug (note the different AP metrics compared to Table 6.2) for bounding box prediction,
- Table A.3 shows the results when fixing the evaluation bug and also computing the loss on all $\hat{Y}^{(t)}$ for bounding box prediction,
- Table A.4 shows the results when computing the loss on all $\hat{Y}^{(t)}$ for state prediction.

In summary, our model still handily beats the baseline models. On bounding box prediction, interestingly the results with the set loss bug fixed tend to be worse than when only using the last prediction to compute the loss. On state prediction, fixing the set loss bug improves almost all results. The largest improvement is on the $\text{AP}_{0.5}$ metric with

[1] swmo@kaist.ac.kr, Korea Advanced Institute of Science and Technology.
[2] chf2018@med.cornell.edu, Weill Cornell Graduate School of Medical Sciences.

Table A.2: Fixed bounding box evaluation, still computing loss only on the last $\hat{Y}^{(T)}$. Average Precision (AP) for different intersection-over-union thresholds for a predicted bounding box to be considered correct. Higher is better. Mean and standard deviation over 6 runs.

Model	AP_{50}	AP_{60}	AP_{70}	AP_{80}	AP_{90}
MLP baseline	$87.6_{\pm 2.4}$	$73.6_{\pm 3.9}$	$48.1_{\pm 5.0}$	$15.1_{\pm 2.8}$	$0.1_{\pm 0.1}$
RNN baseline	$85.8_{\pm 2.5}$	$70.0_{\pm 2.8}$	$43.9_{\pm 2.4}$	$13.3_{\pm 1.5}$	$0.2_{\pm 0.0}$
Ours (10 iters)	$94.0_{\pm 0.4}$	$90.6_{\pm 0.6}$	$82.2_{\pm 1.4}$	$\mathbf{58.9}_{\pm 3.4}$	$\mathbf{16.0}_{\pm 2.4}$
Ours (20 iters)	$\mathbf{98.0}_{\pm 0.5}$	$\mathbf{94.4}_{\pm 0.9}$	$\mathbf{83.0}_{\pm 1.6}$	$55.0_{\pm 3.3}$	$12.3_{\pm 2.6}$
Ours (30 iters)	$95.9_{\pm 2.1}$	$89.4_{\pm 3.3}$	$74.8_{\pm 3.9}$	$46.1_{\pm 3.4}$	$9.4_{\pm 1.9}$

Table A.3: Fixed bounding box evaluation and computing the loss on all intermediate sets. Average Precision (AP) for different intersection-over-union thresholds for a predicted bounding box to be considered correct. Higher is better. Mean and standard deviation over 6 runs. Baselines are the same as in Table A.2.

Model	AP_{50}	AP_{60}	AP_{70}	AP_{80}	AP_{90}
Ours (10 iters)	$97.1_{\pm 1.1}$	$\mathbf{91.4}_{\pm 2.8}$	$\mathbf{73.4}_{\pm 5.1}$	$\mathbf{35.4}_{\pm 4.8}$	$\mathbf{3.3}_{\pm 1.3}$
Ours (20 iters)	$\mathbf{97.5}_{\pm 1.0}$	$90.8_{\pm 2.6}$	$69.4_{\pm 4.9}$	$31.3_{\pm 4.7}$	$2.9_{\pm 1.2}$
Ours (30 iters)	$97.1_{\pm 1.0}$	$88.6_{\pm 2.9}$	$64.4_{\pm 5.3}$	$27.7_{\pm 5.2}$	$2.6_{\pm 1.2}$

Table A.4: State predicton when computing the loss on all intermediate sets. Average Precision (AP) in % for different distance thresholds of a predicted set element to be considered correct. AP_∞ only requires all attributes to be correct, regardless of 3d position. Higher is better. Mean and standard deviation over 6 runs. Baselines are the same as in Table 6.4.

Model	AP_∞	AP_1	$AP_{0.5}$	$AP_{0.25}$	$AP_{0.125}$
Ours (10 iters)	$76.4_{\pm 2.5}$	$69.6_{\pm 2.8}$	$53.7_{\pm 4.3}$	$18.2_{\pm 3.7}$	$2.2_{\pm 0.6}$
Ours (20 iters)	$80.9_{\pm 1.9}$	$77.3_{\pm 2.3}$	$63.9_{\pm 4.1}$	$22.7_{\pm 4.4}$	$2.7_{\pm 0.6}$
Ours (30 iters)	$\mathbf{82.5}_{\pm 1.7}$	$\mathbf{79.8}_{\pm 2.0}$	$\mathbf{66.8}_{\pm 4.1}$	$\mathbf{23.6}_{\pm 4.6}$	$\mathbf{2.8}_{\pm 0.8}$

an increase from 57.0 for the best result to 66.8 for the best result. Notably, running more iterations now only results in strict improvements without worsening on any metric.

Bibliography

[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. arXiv: 1707.02392.

[2] Ryan Prescott Adams and Richard S. Zemel, "Ranking via Sinkhorn Propagation", 2011. arXiv: 1106.1925.

[3] Brandon Amos and J. Zico Kolter, "OptNet: Differentiable Optimization as a Layer in Neural Networks", in *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 2017. arXiv: 1703.00443.

[4] Peter Anderson, Xiaodong He, Chris Buehler, Damien Teney, Mark Johnson, Stephen Gould, and Lei Zhang, "Bottom-Up and Top-Down Attention for Image Captioning and Visual Question Answering", in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. arXiv: 1707.07998.

[5] Cem Anil, James Lucas, and Roger Grosse, "Sorting out Lipschitz function approximation", in *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2019. arXiv: 1811.05381.

[6] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, and Devi Parikh, "VQA: Visual Question Answering", in *The IEEE International Conference on Computer Vision (ICCV)*, 2015. arXiv: 1505.00468.

[7] James Atwood and Don Towsley, "Diffusion-Convolutional Neural Networks", in *Advances in Neural Information Processing Systems 29 (NeurIPS)*, 2016. arXiv: 1511.02136.

[8] Samaneh Azadi, Jiashi Feng, and Trevor Darrell, "Learning Detection With Diverse Proposals", in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. arXiv: 1704.03533.

[9] Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu, "Multiple Object Recognition with Visual Attention", in *International Conference on Learning Representations (ICLR)*, 2015. arXiv: 1412.7755.

[10] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate", in *International Conference on Learning Representations (ICLR)*, 2015. arXiv: 1409.0473.

[11] Lukas Balles and Thomas Fischbacher, "Holographic and other Point Set Distances for Machine Learning", 2019. Available at: https://openreview.net/forum?id=rJlpUiAcYX.

[12] David Belanger and Andrew McCallum, "Structured Prediction Energy Networks", in *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, 2016. arXiv: 1511.06350.

[13] David Belanger, Bishan Yang, and Andrew. McCallum, "End-to-End Learning for Structured Prediction Energy Networks", in *Proceedings of the 34th International Conference on Machine Learning (ICML)*, 2017. arXiv: 1703.05667.

[14] David Belli and Tomas Kipf, "Image-Conditioned Graph Generationfor Road Network Extraction", in *NeurIPS workshop on Graph Representation Learning*, 2019. arXiv: 1910.14388.

[15] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston, "Curriculum Learning", in *Proceedings of the 26th International Conference on Machine Learning (ICML)*, 2009.

[16] Piotr Bojanowski, Armand Joulin, David Lopez Paz, and Arthur Szlam, "Optimizing the Latent Space of Generative Networks", in *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 2018. arXiv: 1707.05776.

[17] Wieland Brendel and Matthias Bethge, "Approximating CNNs with Bag-of-local-Features models works surprisingly well on ImageNet", in *International Conference on Learning Representations (ICLR)*, 2019. arXiv: 1904.00760.

[18] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender, "Learning to rank using gradient-descent", in *Proceedings of the 22nd International Conference on Machine Learning (ICML)*, 2005.

[19] Cătălina Cangea, Petar Veličković, Nikola Jovanović, Thomas Kipf, and Pietro Liò, "Towards Sparse Hierarchical Graph Classifiers", in *NeurIPS workshop on Relational Representation Learning*, 2018. arXiv: 1811.01287.

[20] Nicola De Cao and Thomas Kipf, "MolGAN: An implicit generative model for small molecular graphs", in *ICML workshop on Deep Generative Models*, 2018. arXiv: 1805.11973.

[21] Irène Charon and Olivier Hudry, "A survey on the linear ordering problem for weighted or unweighted tournaments", *A Quarterly Journal of Operations Research (4OR)*, volume 5, number 1, pages 5–60, 2007. DOI: 10.1007/s10288-007-0036-6.

[22] Prithvijit Chattopadhyay, Ramakrishna Vedantam, Ramprasaath R. Selvaraju, Dhruv Batra, and Devi Parikh, "Counting Everyday Objects in Everyday Scenes", in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. arXiv: 1604.03505.

[23] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio, "On the properties of neural machine translation: Encoder-decoder approaches", in *EMNLP workshop on Syntax, Semantics and Structure in Statistical Translation (SSST)*, 2014. arXiv: 1409.1259.

[24] 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. arXiv: 1704.08847.

[25] Joseph Paul Cohen, Henry Z. Lo, and Yoshua Bengio, "Count-ception: Counting by Fully Convolutional Redundant Counting", in *ICCV workshop*, 2017. arXiv: 1703.08710.

[26] Rodrigo Santa Cruz, Basura Fernando, Anoop Cherian, and Stephen Gould, "DeepPermNet: Visual Permutation Learning", in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. arXiv: 1704.02729.

[27] George Cybenko, "Approximation by superpositions of a sigmoidal function", *Mathematics of Control, Signals and Systems*, volume 2, number 4, pages 303–314, 1989, ISSN: 1435-568X. DOI: 10.1007/BF02551274.

[28] Mikyas T. Desta, Larry Chen, and Tomasz Kornuta, "Object-based reasoning in VQA", in *IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2018. arXiv: 1801.09718.

[29] Justin Domke, "Generic Methods for Optimization-Based Modeling", in *Proceedings of the 15th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2012.

[30] 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. arXiv: 1612.00603.

[31] 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. arXiv: 1711.08920.

[32] Thomas Finley and Thorsten Joachims, "Supervised Clustering with Support Vector Machines", in *Proceedings of the 22nd International Conference on Machine Learning (ICML)*, 2005.

[33] Fajwel Fogel, Rodolphe Jenatton, Francis Bach, and Alexandre D'Aspremont, "Convex Relaxations for Permutation Problems", in *Advances in Neural Information Processing Systems 26 (NeurIPS)*, 2013. arXiv: 1306.4805.

[34] Hongyang Gao and Shuiwang Ji, "Graph U-Net", in *Proceedings of the 36th International Conference on Machine Learning (ICML)*, 2019. arXiv: 1905.05178.

[35] Aude Genevay, Gabriel Peyré, and Marco Cuturi, "Learning Generative Models with Sinkhorn Divergences", in *Proceedings of the 21st International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010. arXiv: 1706.00292.

[36] Xavier Glorot and Yoshua Bengio, "Understanding the difficulty of training deep feedforward neural networks", in *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010.

[37] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*. MIT Press, 2016, ISBN: 9780262035613. arXiv: 1807.07987.

[38] Yash Goyal, Tejas Khot, Douglas Summers-Stay, Dhruv Batra, and Devi Parikh, "Making the V in VQA Matter: Elevating the Role of Image Understanding in Visual Question Answering", in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. arXiv: 1612.00837.

[39] 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. arXiv: 1903.00450.

[40] 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. arXiv: 1903.08850.

[41] S. Hamid Rezatofighi, Vijay Kumar B G, Anton Milan, Ehsan Abbasnejad, Anthony Dick, and Ian Reid, "DeepSetNet: Predicting Sets With Deep Neural Networks", in *The IEEE International Conference on Computer Vision (ICCV)*, 2017. arXiv: 1611.08998.

[42] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Deep Residual Learning for Image Recognition", in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. arXiv: 1512.03385.

[43] Paul Henderson and Vittorio Ferrari, "End-to-End Training of Object Class Detectors for Mean Average Precision", in *Asian Conference on Computer Vision (ACCV)*, 2017. arXiv: 1607.03476.

[44] Sepp Hochreiter and Jürgen Schmidhuber, "Long Short-Term Memory", *Neural Computation*, pages 1735–1780, 1997, ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735.

[45] Jan Hosang, Rodrigo Benenson, and Bernt Schiele, "Learning Non-Maximum Suppression", in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. arXiv: 1705.02950.

[46] Han Hu, Jiayuan Gu, Zheng Zhang, Jifeng Dai, and Yichen Wei, "Relation Networks for Object Detection", in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. arXiv: 1711.11575.

[47] 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. arXiv: 1810.12348.

[48] Drew A. Hudson and Christopher D. Manning, "Compositional Attention Networks for Machine Reasoning", in *International Conference on Learning Representations (ICLR)*, 2018. arXiv: 1803.03067.

[49] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2015. arXiv: 1502.03167.

[50] Allan Jabri, Armand Joulin, and Laurens van der Maaten, "Revisiting Visual Question Answering Baselines", in *European Conference on Computer Vision (ECCV)*, 2016. arXiv: 1606.08390.

[51] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu, "Spatial Transformer Networks", in *Advances in Neural Information Processing Systems 28 (NeurIPS)*, 2015. arXiv: 1506.02025.

[52] Daniel D. Johnson, "Learning Graphical State Transitions", in *International Conference on Learning Representations (ICLR)*, 2017.

[53] 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 *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. arXiv: 1612.06890.

[54] Vahid Kazemi and Ali Elqursh, "Show, Ask, Attend, and Answer: A Strong Baseline For Visual Question Answering", 2017. arXiv: 1704.03162.

[55] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann, *Benchmark Data Sets for Graph Kernels*, 2016. Available at: http://graphkernels.cs.tu-dortmund.de.

[56] Jin-Hwa Kim, Jaehyun Jun, and Byoung-Tak Zhang, "Bilinear Attention Networks", in *Advances in Neural Information Processing Systems 31 (NeurIPS)*, 2018. arXiv: 1805.07932.

[57] Yoon Kim, Carl Denton, Luong Hoang, and Alexander M. Rush, "Structured Attention Networks", in *International Conference on Learning Representations (ICLR)*, 2017. arXiv: 1702.00887.

[58] Diederik P. Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", in *International Conference on Learning Representations (ICLR)*, 2015. arXiv: 1412.6980.

[59] Thomas N. Kipf and Max Welling, "Semi-Supervised Classification with Graph Convolutional Networks", in *International Conference on Learning Representations (ICLR)*, 2017. arXiv: 1609.02907.

[60] Alex Krizhevsky, "Learning multiple layers of features from tiny images", Tech. Rep., 2009.

[61] Hugo Larochelle and Geoffrey E Hinton, "Learning to combine foveal glimpses with a third-order Boltzmann machine", in *Advances in Neural Information Processing Systems 23 (NeurIPS)*, 2010.

[62] Justin Lazarow, Long Jin, and Zhuowen Tu, "Introspective neural networks for generative modeling", in *The IEEE International Conference on Computer Vision (ICCV)*, 2017.

[63] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, "Gradient-based learning applied to document recognition", in *Proceedings of the IEEE*, 1998. DOI: 10.1109/5.726791.

[64] Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. 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 (ICML)*, 2019. arXiv: 1810.00825.

[65] Kwonjoon Lee, Weijian Xu, Fan Fan, and Zhuowen Tu, "Wasserstein Introspective Neural Networks", in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. arXiv: 1711.08875.

[66] Victor Lempitsky and Andrew Zisserman, "Learning To Count Objects in Images", in *Advances in Neural Information Processing Systems 23 (NeurIPS)*, 2010.

[67] Chun-Liang Li, Manzil Zaheer, Yang Zhang, Barnabas Poczos, and Ruslan Salakhutdinov, "Point Cloud GAN", in *ICLR workshop on Deep Generative Models for Highly Structured Data*, 2019. arXiv: 1810.05795.

[68] 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. arXiv: 1803.03324.

[69] 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. arXiv: 1411.4038.

[70] Jiasen Lu, Jianwei Yang, Dhruv Batra, and Devi Parikh, "Hierarchical Question-Image Co-Attention for Visual Question Answering", in *Advances in Neural Information Processing Systems (NeurIPS)*, 2016. arXiv: 1606.00061.

[71] Gonzalo Mena, David Belanger, Scott Linderman, and Jasper Snoek, "Learning Latent Permutations with Gumbel-Sinkhorn Networks", in *International Conference on Learning Representations (ICLR)*, 2018. arXiv: 1802.08665.

[72] Nicola Messina, Giuseppe Amato, Fabio Carrara, Fabrizio Falchi, and Claudio Gennaro, "Learning Relationship-aware Visual Features", in *ECCV workshop on Compact and Efficient Feature Representation and Learning in Computer Vision (CEFRL)*, 2018.

[73] Dmytro Mishkin and Jiri Matas, "All you need is a good init", in *International Conference on Learning Representations (ICLR)*, 2016. arXiv: 1511.06422.

[74] Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu, “Recurrent Models of Visual Attention”, in *Advances in Neural Information Processing Systems 27 (NeurIPS)*, 2014. arXiv: 1406.6247.

[75] Igor Mordatch, “Concept Learning with Energy-Based Models”, in *ICLR workshop*, 2018. arXiv: 1811.02486.

[76] 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. arXiv: 1811.01900.

[77] Marion Neumann, Roman Garnett, Christian Bauckhage, and Kristian Kersting, “Propagation Kernels: Efficient Graph Kernels from Propagated Information”, *Machine Learning*, volume 102, number 2, pages 209–245, 2016, ISSN: 0885-6125.

[78] 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. arXiv: 1605.05273.

[79] Mehdi Noroozi and Paolo Favaro, “Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles”, in *European Conference on Computer Vision (ECCV)*, 2016. arXiv: 1603.09246.

[80] Rasmus Palm, Ulrich Paquet, and Ole Winther, “Recurrent Relational Networks”, in *Advances in Neural Information Processing Systems 31 (NeurIPS)*, 2018. arXiv: 1711.08028.

[81] Panos M. Pardalos and Stephen A. Vavasis, “Quadratic programming with one negative eigenvalue is NP-hard”, *Journal of Global Optimization*, volume 1, number 1, pages 15–22, 1991. DOI: 10.1007/BF00120662.

[82] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer, “Automatic differentiation in PyTorch”, in *NeurIPS workshop on Autodiff*, 2017.

[83] Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron C. Courville, “FiLM: Visual Reasoning with a General Conditioning Layer”, in *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, 2018. arXiv: 1709.07871.

[84] 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. arXiv: 1612.00593.

[85] Mengye Ren and Richard S. Zemel, "End-To-End Instance Segmentation With Recurrent Attention", in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. arXiv: 1605.09410.

[86] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", in *Advances in Neural Information Processing Systems 28 (NeurIPS)*, 2015. arXiv: 1506.01497.

[87] 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. arXiv: 1805.00613.

[88] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton, "Dynamic Routing Between Capsules", in *Advances in Neural Information Processing Systems 30 (NeurIPS)*, 2017.

[89] Adam Santoro, David Raposo, David G Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Tim Lillicrap, "A simple neural network module for relational reasoning", in *Advances in Neural Information Processing Systems 30 (NeurIPS)*, 2017. arXiv: 1706.01427.

[90] Aliaksei Severyn and Alessandro Moschitti, "Learning to Rank Short Text Pairs with Convolutional Deep Neural Networks", in *Special Interest Group on Information Retrieval (SIGIR)*, 2015.

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

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

[93] Martin Simonovsky and Nikos Komodakis, "GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders", in *International Conference on Artificial Neural Networks (ICANN)*, 2018. arXiv: 1802.03480.

[94] Richard Sinkhorn, "A Relationship Between Arbitrary Positive Matrices and Doubly Stochastic Matrices", *The Annals of Mathematical Statistics*, volume 35, number 2, pages 876–879, 1964. DOI: 10.1214/aoms/1177703591.

[95] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", *Journal of Machine Learning Research*, pages 1929–1958, 2014, ISSN: 1532-4435.

[96] Russell Stewart and Mykhaylo Andriluka, "End-to-end people detection in crowded scenes", in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. arXiv: 1506.04878.

[97] Veselin Stoyanov, Alexander Ropson, and Jason Eisner, "Empirical Risk Minimization of Graphical Model Parameters Given Approximate Inference, Decoding, and Model Structure", in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011.

[98] Damien Teney, Peter Anderson, Xiaodong He, and Anton van den Hengel, "Tips and Tricks for Visual Question Answering: Learnings from the 2017 Challenge", in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. arXiv: 1708.02711.

[99] Alexander Trott, Caiming Xiong, and Richard Socher, "Interpretable Counting for Visual Question Answering", in *International Conference on Learning Representations (ICLR)*, 2018. arXiv: 1712.08697.

[100] 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 30 (NeurIPS)*, 2017. arXiv: 1706.03762.

[101] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur, "Order Matters: Sequence to sequence for sets", in *International Conference on Learning Representations (ICLR)*, 2015. arXiv: 1511.06391.

[102] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly, "Pointer Networks", in *Advances in Neural Information Processing Systems 28 (NeurIPS)*, 2015. arXiv: 1506.03134.

[103] 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. arXiv: 1901.09006.

[104] 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. arXiv: 1711.05246.

[105] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka, "How Powerful are Graph Neural Networks?", in *International Conference on Learning Representations (ICLR)*, 2019. arXiv: `1810.00826`.

[106] Carl Yang, Peiye Zhuang, Wenhan Shi, Alan Luu, and Pan Li, "Conditional Structure Generation through Graph Variational Generative Adversarial Nets", in *Advances in Neural Information Processing Systems 32 (NeurIPS)*, 2019.

[107] 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)*, 2018. arXiv: `1712.07262`.

[108] Zichao Yang, Xiaodong He, Jianfeng Gao, Li Deng, and Alexander J. Smola, "Stacked Attention Networks for Image Question Answering", in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. arXiv: `1511.02274`.

[109] 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. arXiv: `1806.08804`.

[110] 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. arXiv: `1802.08773`.

[111] Seungil You, David Ding, Kevin Canini, Jan Pfeifer, and Maya Gupta, "Deep Lattice Networks and Partial Monotonic Functions", in *Advances in Neural Information Processing Systems 30 (NeurIPS)*, 2017. arXiv: `1709.06680 [stat.ML]`.

[112] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan R Salakhutdinov, and Alexander J Smola, "Deep Sets", in *Advances in Neural Information Processing Systems 30 (NeurIPS)*, 2017. arXiv: `1703.06114`.

[113] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, Murray Shanahan, Victoria Langston, Razvan Pascanu, Matthew Botvinick, Oriol Vinyals, and Peter Battaglia, "Deep reinforcement learning with relational inductive biases", in *International Conference on Learning Representations (ICLR)*, 2019.

[114] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen, "An End-to-End Deep Learning Architecture for Graph Classification", in *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

[115] Yan Zhang, Jonathon Hare, and Adam Prügel-Bennett, "Deep Set Prediction Networks", in *Advances in Neural Information Processing Systems 32 (NeurIPS)*, 2019. arXiv: 1906.06565.

[116] Yan Zhang, Jonathon Hare, and Adam Prügel-Bennett, "FSPool: Learning Set Representations with Featurewise Sort Pooling", in *NeurIPS workshop on Sets & Partitions*, 2019. arXiv: 1906.02795.

[117] Yan Zhang, Jonathon Hare, and Adam Prügel-Bennett, "Learning Representations of Sets through Optimized Permutations", in *International Conference on Learning Representations (ICLR)*, 2019. arXiv: 1812.03928.

[118] Yan Zhang, Jonathon Hare, and Adam Prügel-Bennett, "Learning to Count Objects in Natural Images for Visual Question Answering", in *International Conference on Learning Representations (ICLR)*, 2018. arXiv: 1802.05766.

[119] Yu Zhou, Yu Jun, Xiang Chenchao, Fan Jianping, and Tao Dacheng, "Beyond Bilinear: Generalized Multi-modal Factorized High-order Pooling for Visual Question Answering", 2017. arXiv: 1708.03619.

[120] Chen Zhu, Yanpeng Zhao, Shuaiyi Huang, Kewei Tu, and Yi Ma, "Structured Attentions for Visual Question Answering", in *The IEEE International Conference on Computer Vision (ICCV)*, 2017. arXiv: 1708.02071.