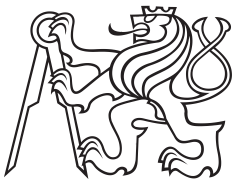
Bachelor's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Exploring Symmetries in Deep Learning

Martin Krutský

Supervisor: Ing. Gustav Šír
Field of study: Open Informatics
Subfield: Artificial Intelligence and Computer Science
May 2021

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Krutský Martin** Personal ID number: **483792**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Exploring Symmetries in Deep Learning

Bachelor's thesis title in Czech:

Základy symetrií v hlubokém učení

Guidelines:

All machine learning models are inherently trying to explore symmetries, i.e. some forms of shared regularities, in the input data distributions. However, in many model classes, such symmetries exist also in the model parameter space. This has been studied extensively in lifted graphical models, but remains largely unexplored in deep learning. A famous exception are convolutional networks (CNNs), exploiting invariance w.r.t. translation, achieving tremendous successes in many tasks. The student is expected to study beyond classic CNNs to explore a wider range of symmetries and their exhibition in the parameterization of the respective neural models. The goal will be to assess benefits of the explicit treatment of model symmetries w.r.t. symmetry-agnostic neural models. Finally, a custom solution of a non-trivial problem exhibiting symmetries is to be expected.

1. Perform an extensive review of literature related to symmetries, particularly in the context of deep learning (CNNs, deep sets, etc.).
2. Get acquainted with weight sharing in neural models and its practical implementation.
3. Provide a structured overview of various forms of symmetries and the respective neural approaches.
4. Implement selected simple problems with neural models for demonstration.
5. Solve a non-trivial problem exhibiting symmetries.

Bibliography / sources:

[1] Anselmi, Fabio, et al. "Symmetry-adapted representation learning." Pattern Recognition 86 (2019): 201-208.
[2] Hu, Shell Xu, Sergey Zagoruyko, and Nikos Komodakis. "Exploring weight symmetry in deep neural networks." Computer Vision and Image Understanding 187 (2019): 102786.
[3] Gens, Robert, and Pedro M. Domingos. "Deep symmetry networks." Advances in neural information processing systems. 2014.
[4] Zaheer, Manzil, et al. "Deep sets." Advances in neural information processing systems. 2017.
[5] Kimmig, Angelika, Lilyana Mihalkova, and Lise Getoor. "Lifted graphical models: a survey." Machine Learning 99.1 (2015): 1-45.
[6] Sourek, Gustav, and Filip Zelezny. "Lossless Compression of Structured Convolutional Models via Lifting." arXiv preprint arXiv:2007.06567 (2020).

Name and workplace of bachelor's thesis supervisor:

Ing. Gustav Šír, Department of Computer Science, FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **05.01.2021** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until: **30.09.2022**

Ing. Gustav Šír	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgements

I would like to thank my supervisor, Ing. Gustav Šír, for guiding me through the topic of the thesis and for pushing me forward when I was stuck on a problem.

Also, I would like to thank my family and friends for supporting me through the whole course of study at FEE CTU.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 21 May 2021

Abstract

Many applications of deep learning involve approximation of functions that exhibit some form of symmetries with respect to their inputs. However, this fact is often neglected by machine learning practitioners, with the exception of the widespread use of convolutional neural networks. In this thesis, I explore a wider range of neural architectures that explicitly exploit domain symmetries in their computation, ranging from simple weight-sharing schemes and convolutions to specialized models such as Deep Sets and graph neural networks. I then compare the performance of these architectures with conventional neural models across respective example problems exhibiting various symmetries. These range from learning of simple functions such as XOR and integer set sum to pattern detection within a binary array and distinguishing isomorphic from non-isomorphic graphs.

Finally, I build on the previous findings to solve a nontrivial problem of classification of the Rubik's Cube states. The findings of this thesis generally support the intuition that for problems exhibiting symmetries, the symmetry-aware neural architectures are more efficient in terms of training and generalization performance than their common, symmetry-unaware counterparts.

Keywords: neural networks, deep learning, symmetries, parameter sharing, convolutions, graph neural networks

Supervisor: Ing. Gustav Šír

Abstrakt

Mnoho aplikací hlubokého učení se zabývá aproximací funkcí, které obsahují nějakou formu symetrie vzhledem k jejich vstupu. Tento fakt se však často při tvorbě architektury zanedbává, výjimkou jsou pouze hojně rozšířené konvoluční neuronové sítě. V této práci zkoumám širší spektrum architektur, které v rámci svých výpočtů explicitně využívají symetrií v datech, od neuronových sítí se sdílenými vahami a konvolucí, po specializované modely jako jsou Deep Sets a grafové neuronové sítě. Tyto architektury poté na relevantních ukázkových problémech, které vykazují různé známky symetrií, porovnávám s konvenčními neuronovými sítěmi. Mezi zkoumané problémy patří jednoduché funkce jako je XOR, sčítání množin celých čísel, ale také detekce vzorce v binárním poli, či rozlišování izomorfních a neizomorfních grafů.

Na závěr poznatky z předchozích experimentů využívám pro řešení netriviálního problému klasifikace stavů Rubikovy kostky. Poznatky této práce obecně podporují tezi, že pro problémy vykazující známky symetrií je využití architektur explicitně využívajících těchto symetrií efektivnější než použití konvenčních neuronových sítí, a to jak pro rychlost učení, tak pro generalizaci na daných problémech.

Klíčová slova: neuronové sítě, hluboké učení, symetrie, sdílení parametrů, konvoluce, grafové neuronové sítě

Překlad názvu: Základy symetrií v hlubokém učení

Contents

Figures

Tables

Chapter 1

Introduction

The information we perceive from our surroundings is inherently geometric in its nature. From 3-dimensional objects around us, its 2-dimensional projections we call images, to abstract notions we visualize with geometry through our imagination. And since geometry can be considered as a study of symmetry and invariance, it is only logical that symmetries are inherently built into our world.

It follows that when we model parts of our world with statistical learning methods, such as neural networks, the learned functions should be able to approximate the underlying symmetries. Conventional learning approaches rely on extracting the symmetric properties from the observed relations between data points and their respective labels within large datasets. However, such an approach can often be highly sample inefficient. In this thesis, I will examine multiple methods that aim to improve the speed and learning performance of these methods by embedding the symmetries explicitly into the architecture of the learning algorithm.

Before diving into the methods, I will formally introduce the key underlying concepts of *equivariance*, *invariance*, *symmetry*, *symmetry group*, and different types of symmetries to be used throughout the thesis.

1.1 Equivariance, Invariance, and Symmetry

When we talk about symmetry in everyday life, we usually mean a visual pattern that is, in some way, repeated within an object. However, symmetry is a much broader term from the mathematical perspective. Firstly, I will define the concept of *equivariance*, followed by the definition of *invariance*.

Definition 1.1. Function f is *equivariant* with respect to mapping ϕ iff the following identity holds for $\forall x \in X$, where X is the domain of f: $f(\phi(x)) = \phi(f(x))$.

If a function is equivariant to several mappings, and these mappings form a set M, we say that a function f is equivariant to the set of mappings M.

Definition 1.2. Function f is *invariant* with respect to mapping ϕ iff the following identity holds $\forall x \in X$, where X is the domain of f: $f(\phi(x)) = f(x)$.

The term *symmetry* describes a special form of invariance connected with geometric transformations. Unlike invariance, which uses mapping between two possibly different sets, the mapping ϕ (see the definition) maps the input back to the same input space:

Definition 1.3. *Symmetry* is an invariance with respect to a mapping $\phi : A \to A$, where ϕ is a geometric transformation, such as reflection, translation, rotation, or permutation, and A is the domain of the transformation.

Given that the subject of the thesis concerns symmetries in deep networks, it is logical that the property we would like to preserve when applying the transformation is the *value predicted by the network* - the same class in case of classification, and approximately the same real number for regression.

Finally, an object can be invariant to multiple transformations, which leads to the definition of symmetry group:

Definition 1.4. *Symmetry group* (of an object) is a group of all transformations which preserve the invariance of the object.

Nevertheless, it is common to call the whole symmetry group simply *a symmetry*. Thus, for simplicity, I will e.g. commonly use the term *rotation symmetry* instead of *rotation symmetry group*.

1.1.1 Types of Symmetries

In this thesis, I will deal with the following types of symmetries: reflection, translation, rotation, input permutation, and graph isomorphism. There are many other types of symmetries, however, the aforementioned ones are commonly seen as the most prevalent in machine learning applications. In this section, I will proceed from the most specific to the most general symmetries. The same order will then be followed later in the presented experiments (with the exception of Rubik's cube state classification, which is, as the selected final nontrivial experiment, presented at the end of the thesis).

Reflection Symmetry

A reflection symmetry is defined on functions with binary input: A function f is symmetric with respect to reflection, iff $f(x_1, x_2) = f(x_2, x_1)$, for any $(x_1, x_2) \in X$, where X is the domain of the function f. This symmetry is commonly present as the mirror ("left-right") symmetry in nature, as well as in many core binary logical functions in computer science, such as AND, OR, NAND, NOR, XOR, etc.

Translation Symmetry

For the translation symmetry, we generally assume an inherent ordering over the input space elements. For simplicity, let me consider translation in 1D space. A function f is symmetric with respect to translation π of a pattern of length n iff $f(\dots, x_i, x_{i+1}, \dots, x_{i+n}, \dots) = f(\dots, x_{i+\pi}, x_{i+\pi+1}, \dots, x_{i+\pi+n}, \dots)$ where $i, n \in \mathbb{N}$ and $\pi \in \mathbb{Z}$.

This type of symmetry is common in images and generally in any object recognition/detection tasks, where the output does not depend on the position of the object but rather on its presence or absence.

Rotation Symmetry

To define rotation symmetry, let me consider 2D input space with polar coordinates, since in this coordinate system, this symmetry closely resembles the translation symmetry. For a function f, its domain X, a 2-dimensional input $x \in X$, and an angle θ', let me construct another input $x' \in X$ by moving[1] a pattern of n points from x with polar coordinates $(\rho_1, \theta_1), (\rho_2, \theta_2), \dots (\rho_n, \theta_n)$ to polar coordinates $(\rho_1, \theta_1 + \theta'), (\rho_2, \theta_2 + \theta'), \dots (\rho_n, \theta_n + \theta')$. A function f is then symmetric with respect to rotation iff $f(x) = f(x')$. Similarly, the rotation could be defined for spherical coordinates in 3 dimensions or any N-dimensional generalization of these coordinates.

Akin to the translation symmetry, this type of symmetry is commonly present in visual recognition tasks but can also be presented more abstractly, e.g., in problems represented by graph structure where the angle of view is irrelevant, such as in the Rubik's cube problem.

Input Permutation Symmetry

Let me now introduce a generalization of all the aforementioned symmetries – the input permutation symmetry. When we reflect, translate, or rotate data, it is an instance of input space permutation. There are, however, other permutations of the input data, which cannot be covered by the previous concepts.

A function f is symmetric with respect to input permutation π iff $f(x_1, x_2, \dots x_n) = f(x_{\pi^{-1}(1)}, x_{\pi^{-1}(2)}, \dots x_{\pi^{-1}(n)})$, where π is *any* active permutation. Examples of functions that are inherently symmetric to input permutations include addition and multiplication (if we consider them as functions of n variables, based on their associativity), logical functions of N dimensions, statistical functions (such as arithmetic mean or median), etc.

Graph Isomorphism

A direct generalization of the permutation symmetry is the graph isomorphism symmetry. Although this symmetry is restricted to graph structure of data, all the previous symmetries can actually be modeled by choosing an appropriate graph representation. For instance, a permutation symmetry on a set can be thought of as a graph isomorphism in a graph with no edges. However, the graph isomorphism notion generally also encompasses the relations between nodes.

Definition 1.5. Two graphs, G and H, are isomorphic iff there exists a bijection $f : V(G) \to V(H)$ that preserves their adjacency, meaning that $\forall u, v \in V(G)$, $f(u)$ and $f(v)$ are adjacent iff u and v are adjacent.

Then, having a set of graphs, isomorphism divides the set into pairwise disjoint subsets (isomorphism classes). Each class then includes all the graphs from the set which are isomorphic to each other. Talking about graph isomorphism of a function, we would like to get the same output for all the isomorphic graphs. With respect to a machine learning algorithm and some dataset consisting of graphs, we would

[1] Note that when translating from polar coordinates back to Cartesian, such as in an image matrix, we need to round the result, as the input indices are integers.

like to either: (i) assign distinct labels to each isomorphism class and then learn to classify the graphs that belong to it with the respective label, or (ii) choose a single isomorphism class, label it with 1 (a positive class), and label the rest of the classes with 0 (a negative class).

The graph isomorphism is a graph theoretical problem for which we do not yet know for certain the computation complexity class. We know, however, that it is not solvable in polynomial time, so there is a need for approximate algorithms, such as the Weisfeiler-Leman test (also called Color Refinement algorithm; introduced in 1968 [25]), that serves as a refutation check for the problem, i.e. it forms a necessary, but generally insufficient, condition.

Weisfeiler-Leman Test

For each of the two graphs, the algorithm does the following:

1. start with one color for all nodes of a graph
2. aggregate the colors of the neighbors of each node
3. compare the aggregated sets of colors at each node
4. if there are any different sets with the same color, add new color(s) and recolor the nodes with them to resolve the issue
5. if there was any color change in the previous step, continue to step 2, otherwise finish

If, in the end, both graphs have the same colors of all the nodes (or this state can be reached by a simple renaming of colors in one of the graphs), the Weisfeiler-Leman test concludes that the two graphs are isomorphic. Unfortunately, this algorithm can be a source of false positives - we can only be sure that graphs are *not* isomorphic if the test says so, but we cannot be sure about the isomorphism in the aforementioned case.

Remark 1.6. Before finishing the section on graph isomorphism, an interesting question is what are the theoretical bounds on the number of iterations (an iteration being a single run of steps 2 to 6) that Weisfeiler-Leman test (WL test) needs for convergence. Kiefer and McKay proved in 2020 that "the upper bound on the iteration number of Colour Refinement on graphs of order n is $n - 1$" [14], where n is the number of nodes in the graph. Moreover, they showed that the upper bound for $n > 10$ is always either $n - 1$ or $n - 2$ and for $n \leq 10$ the upper bound is $n - 2$ at most. Interestingly, this question is closely connected to how many graph convolutional layers we should stack in a graph neural network when learning a problem with isomorphism symmetry (explained later in Section 5.1).

1.2 Using Symmetries in Deep Learning

When a conventional fully connected neural network is trained (in a supervised manner) to approximate a function involving symmetry, there exist multiple sets (or

classes of equivalence) of training examples with the same labels that are mapped onto each other by the respective mapping ϕ (see Definition 1.3). Since the network does not have any prior knowledge about this equivalence, each example from the equivalence class may actually point the optimizer in a different direction (the gradient is different). This means that parameters are adjusted differently for each sample, causing slower convergence in terms of the number of iterations of the gradient descent. On the other hand, symmetry-aware architectures represent the equivalent samples by the same embedding. Thus, the computations involving this embedding produce the same gradient for all equivalent samples, leading to faster convergence.

A technique often utilized with the conventional neural networks is data augmentation. Using symmetry-unaware approaches, the machine learning system designers are facing the fact that a neural network cannot grasp the symmetry without being explicitly shown each and every example from the symmetric class. Thus, machine learning engineers need to imitate the symmetric mappings by including more artificial data - e.g. by enlarging a dataset of images with rotated and reflected clones of the already existing examples, or by collecting schemes of a molecule from all angles instead of including the molecule only once, etc. In some cases, this means collecting more data, which is naturally not always physically possible, while in other cases, we can digitally augment the dataset, which does not always work perfectly and creates other challenges, such as increased training time and power consumption.

The question we seek to answer is: What are the symmetry-aware architectures that would allow for shorter training times and less data being collected? The following paragraphs describe some of such symmetry-aware approaches, which will be used in the practical part of the thesis. They are all based on the concept of parameter sharing, which is the idea of reusing parameters at multiple places in the (unfolded) computational graph of the model. Then, during backpropagation, we do not have to compute the gradient updates multiple times for the shared parameters, saving some training time.[2]

However, each of the following architectures achieves this parameter sharing differently. I will start with architectures using only parameter sharing (Section 1.2.1), before continuing with ones that use pooling as well (Section 1.2.3, Section 1.2.2, and Section 1.2.4).

1.2.1 Fully Connected Networks with Weight Sharing

Starting with fully connected networks with weight sharing schemes, these architectures closely resemble conventional symmetry-unaware fully connected networks.

The difference is that multiple weights are functionally tied to the same weight object that is adjusted during learning, and the changes to the value of the object are automatically propagated to the weights tied with it. This allows us to have faster learning without sacrificing the performance of the architecture, as was shown by Hu et al. [13]. Compared to the well-known convolutional neural networks (see

[2]The number of learnable parameters directly influences the training time because in each iteration of the backpropagation algorithm, [23] a partial derivative needs to be computed for each learnable parameter.

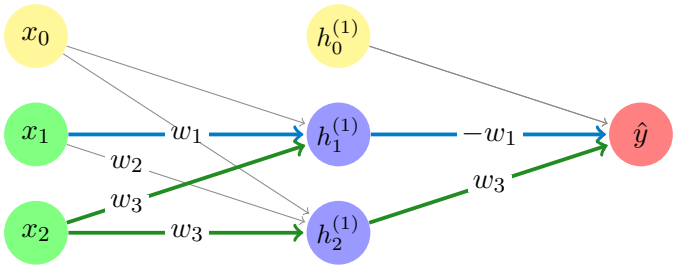


Figure 1.1: An example of weight sharing scheme in a neural network - blue and green highlighted edges represent shared weights

Section 1.2.2), there are two differences: (i) the computational graph still adheres to the structure of fully connected network, and (ii) any two weights can be tied together, even across the layers.

Remark 1.7. To give a better idea about the technique, there is an example of a neural network with a weight sharing scheme in Figure 1.1[3] which occurs later on in the experimental part of the thesis: weight objects shared with more than a single weight of the networks are in green and blue. In this particular scheme, there are 6 weight objects (3 of them are biases), even though the whole network consists of 9 weights in total.

Let me continue with the definition of a weight sharing scheme. In order to specify a dependency between weights, you have to:

1. specify the weights which reference the same weight object,
2. specify the mappings defining how to calculate each particular weight's value from the value of the shared weight object.

Specifying the dependencies (or lack of them) for all the weights in the network will then be called a *weight sharing scheme* in this thesis.

Note that the second step of the scheme definition is often neglected (not just in the case of weight sharing schemes, but also generally in symmetry-aware architectures), and a simple equivalence is used instead, meaning all the shared weights have exactly the same value as the single referenced object. However, the dependency mappings between two shared weights can generally be very diverse. In general, it can be any function $f : \mathbb{R} \to \mathbb{R}$, though some functions are more logical than some others, e.g. having a weight w_i being dependent on $w_j, i \neq j$ with function $f : \mathbb{R} \to \{0\}$ is not useful at all.

Considering the whole class of single-variable polynomial functions as candidates for the dependencies would be a possible interesting research path for the future. But for simplicity, in the experimental part of the thesis, I will only deal with two simple types of functional dependencies:

1. $f(x) = x$ (the dependent weights have the same value),
2. $f(x) = -x$ (the weights have the opposite value).

[3] For visualizations of neural architectures with fully connected layers, I used the Neural Network LaTeX package by M. Cowan, see `https://github.com/battlesnake/neural`

Definition 1.8. Two weights w_1, w_2 from neural networks are *symmetric* iff their dependency is given by a function $f(x) = x$. Thus $w_1 = w_2$.

Definition 1.9. Two weights w_1, w_2 from neural networks are *antisymmetric* iff their dependency is given by a function $f(x) = -x$. Thus $w_1 = -w_2$.

The idea behind weight sharing is to build in the symmetries of the problem being learned (as we described in the intro of this Section 1.2). On the other hand, if our estimate of dependencies is wrong, there might actually be no good solution in the chosen weight sharing scheme because a good solution for the problem might require the chosen weights to be independent of each other (or, technically speaking, in a different functional relationship than specified). This leads to a definition of an ill-defined sharing scheme.

Definition 1.10. A sharing scheme is *ill-defined* iff there exists no configuration of weights that would construct a classifier with perfect accuracy (given that the weights are used in that sharing scheme).

1.2.2 Convolutional Neural Networks

Neural architectures designated for learning functions involving translation symmetry are called Convolutional Neural Networks (or CNNs). CNNs are based on the work of Fukushima [7] and became popular after their fusion with the backpropagation algorithm by LeCun et al. [16]. Similarly to fully connected networks, CNNs can be formed by multiple layers. But those layers differ from conventional fully connected layers. Instead of a matrix multiplication involving all the input data and all the network's parameters which happens during the forward propagation in a fully connected layer, an operation called *convolution* is performed iteratively in each convolutional layer.

In mathematics, a convolution describes a certain process of combination of two functions. Instead, I will use an informal definition from Dumoulin's guide to convolution arithmetic for deep learning [5]:

Definition 1.11. "A discrete convolution is a linear transformation that preserves [the] notion of ordering [that is intrinsically present in data]."

This restricted version is sufficient as a basis for an explanation of convolution in deep learning.

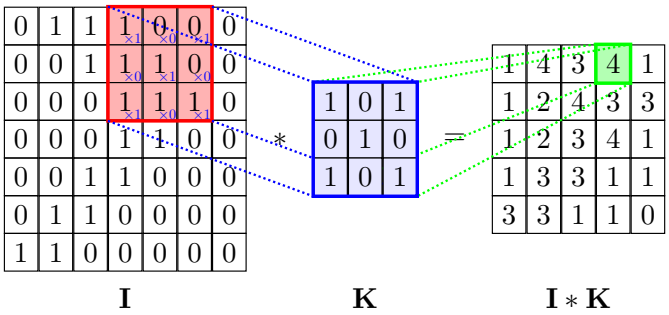


Figure 1.2: An example of convolution on 2-dimensional data

What linear transformation is commonly used in CNNs? Usually, it is a point-wise multiplication of a local part of the input with a *filter* (a parameter matrix that is smaller than the input data), and the results are then summed up. After these steps, the filter is translated to the right (or to the beginning and then down if it is located at the end of a subarray), and the whole procedure is repeated. This process is illustrated by Figure 1.2[4].

The result is a smaller array, where each item represents an aggregated information about some local patterns in the original data. This is the idea behind the convolutional layer: if we can get the right filters for patterns we are searching for, the aggregated data will represent activations, telling us how closely the local data resemble the wanted pattern. Stacking multiple convolutional layers then helps us to search for more sophisticated patterns that are dependent on simpler ones.

Since it would be impossible to hand-pick the right filters in any reasonably high-dimensional data, the filter array is an array of learnable parameters that are, similarly to weights in a fully connected layer, usually randomly initialized and learned during backpropagation. This is another good example of weight sharing - the filter (the learnable parameters) is the same for the whole picture. If we used fully connected networks instead, the network would have to manipulate many more weights during training since the number of weights in a fully connected layer is proportional to the size of its input, not to the size of a smaller filter, as is the case with the convolutional layer.

So far, with one or multiple convolutional layers, we have only achieved *equivariance* (see Definition 1.1) of the neural network. Instead, in typical applications, we would like to perform e.g. a classification *invariant* to the pattern position deciding whether a complex pattern is or is not present in the input.

This is where pooling layers come into play. A pooling layer is an operation that aggregates smaller regions in the output of a convolutional layer by applying functions, such as maximum or average. Depending on the size of the aggregated regions, the output (for a single example) could be of any size from 1 to the size of the input.

With binary classification or 1-dimensional regression, the extreme case of a global pooling with an output of size 1 enforces invariance (see Definition 1.2) with respect to the pattern/filter of the last convolutional layer. This might work well in simpler cases, but in more sophisticated ones, we usually leave more degrees of freedom, output data of higher dimensions, and leave the task-specific inference on the final, fully connected layer.

1.2.3 Deep Sets

While building a concrete weight sharing scheme into a neural network is a general technique that can be adapted to any type of symmetry, it is sometimes impractical since we may not have the intuition about the particular relationships between the network's weights.

Zaheer et al. in [28] try to offer a solution by introducing a neural architecture

[4]The illustration was acquired from GitHub TikZ repository of Petar Veličković `https://github.com/PetarV-/TikZ`

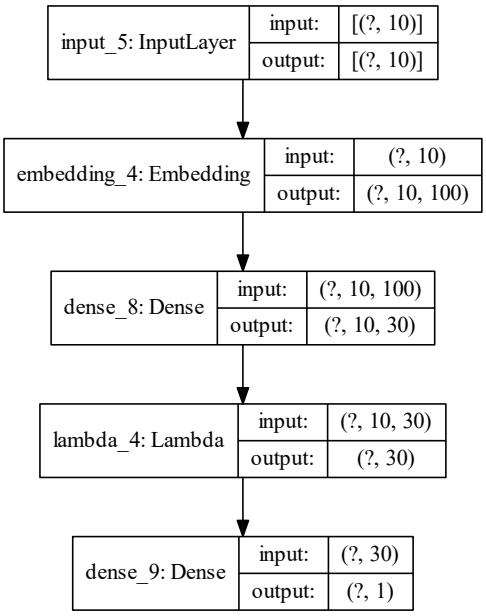


Figure 1.3: An example of the Deep Sets architecture from the integer set sum experiments

called Deep Sets. As the name suggests, the input space is a set of sets, and since sets are unordered collections of items, it is logical that this architecture is designed for learning functions invariant to input permutations. This disallows its usage on problems that require a more specific symmetry, such as translation, since Deep Sets would produce the same output for all inputs from the same input permutation class, regardless of their invariance w.r.t. the translation symmetry.

Nevertheless, specifically for problems involving input permutation invariance, it promises to be a general building block for neural architectures. Another added value of Zaheer's architecture is that it is able to generalize over inputs/sets of different lengths.

The idea behind Deep Sets is to learn high dimensional embeddings (in the experiments mentioned in Zaheer's paper, this dimension is usually in the order of hundreds) for the vocabulary of all possible elements which could occur in an input set. For simplicity, let's restrict those elements to real numbers. If the dimension of embeddings is D and the length of an input set is M, then the output for a single data sample has the shape $M \times D$ and each of the M subarrays is obtained by indexing the learned embeddings with the element of the original input array.

After that, the embeddings are propagated through one or more fully connected layers. This transforms an input of shape $M \times D$ into an output of shape $M \times D'$.

Then, the trick for generalization over different lengths of input is to sum the mentioned output along the axis of size M, obtaining an array of shape $D' \times 1$ for each data point. *This step also explicitly enforces input permutation symmetry of the*

whole network. Finally, there is a last fully connected layer which either performs a regression (typically with a single output without nonlinearities) or classification (single output and e.g. sigmoid nonlinearity for binary classification; multiple outputs and softmax for more classes).

An example illustration of the described architecture generated by the accompanying code published by Zaheer et al. [28][27] is in the Figure 1.3. It depicts the exact architecture used for learning to sum a set of digits. Note that the Lambda layer represents the sum along the axis of length D.

1.2.4 Graph Neural Networks

The last approach I will use to exploit symmetries inherently present in data are graph neural networks. Many objects in our world share relationships and these relationships can be encoded in terms of edges between nodes (either directed or undirected). Definitions of directed and undirected graphs follow:

Definition 1.12. An *undirected graph* is a pair $G = (V, E)$, where V is the set of *vertices* and E is the set of *edges*. A vertex is a named object, possibly accompanied by a feature vector. An edge is a set $\{x, y\}$ for some vertices x, y.

Definition 1.13. A *directed graph* is a pair $G = (V, E)$, where V is the set of *vertices* and E is the set of *edges*. A vertex is a named object, possibly accompanied by a feature vector. An edge is an ordered pair (x, y) for some vertices x, y.

With a conventional machine learning approach, we commonly take all the data present in the nodes and edges and aggregate them in some way to produce an array or tensor representation for learning. But when the processing order is fixed, this transformation loses the structural information about the data - e.g. when we rotate a graph, the resulting tensor looks very different since its sub-arrays are shuffled. Nevertheless, it is still the same graph, and our model should treat it the same.

The idea of graph neural networks is to maintain the structure of the data and perform the operations directly on their graph representation. In this thesis, I will concentrate on graph convolutional networks, more precisely on graph convolutional layers.

Graph Convolutional Neural Networks

Similarly to convolutional layers introduced in Section 1.2.2, graph convolutional layers also perform an aggregation over a local part of a graph before sliding to the next section. Instead of a local filter, we start with a node in the graph and aggregate the feature vectors of its neighbors and the node itself.

This aggregation, which can be arbitrarily complex, involves matrix multiplications between the feature vectors and learnable parameters. The last step is to update the node's feature vector with the newly computed aggregation. An example of this process is illustrated in Figure 1.4[5].

In the experiments, I will use three graph convolutional architectures: (i) graph convolutional layer introduced by Kipf and Welling [15], (ii) graph isomorphism

[5]This illustration was also acquired from Veličković's repository `https://github.com/PetarV-/TikZ`

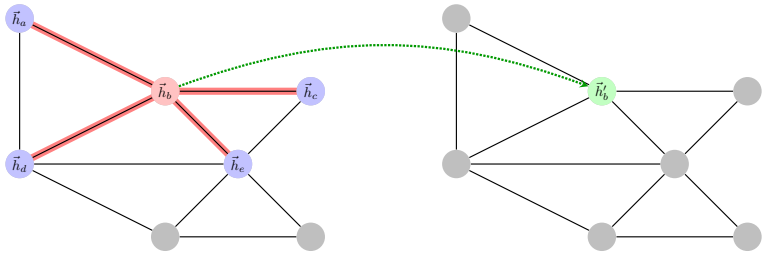


Figure 1.4: An example of graph convolution

layer introduced by Xu et al. [26], and (iii) generalized graph convolutional layer, introduced by Li et al. [17].

The most important difference between the first two architectures is the use of mean aggregation (by Kipf and Welling) and sum aggregation (by Xu) on the output of matrix multiplication between node's and its neighbors' features and the parameter matrix. The third architecture adds the possibility of using edge features.

1.3 Outline of the Experiments

In the experiments in the following sections, an example class of problems will be introduced for each type of symmetry. These will be firstly solved with the conventional neural architectures (I will refer to them as *(vanilla) fully connected neural networks*, or multilayer perceptrons, or simply as *MLPs*), which will provide me with a baseline. Then, the introduced symmetry-aware architectures (weight sharing architectures, Deep Sets, convolutional neural networks, etc.) will be trained to solve the problems, and the results will be compared with the baseline in terms of accuracy and efficiency of convergence. Thus, the experiments will generally follow the A-B testing scheme:

A Baseline with symmetry-unaware architectures

B Experiments with symmetry-aware architectures

The complexity of the neural architectures will be gradually increased (in terms of depth and width of the networks) along with the complexity of the example problems. The classes of symmetries and experiments will then follow in this order:

1. Reflection Symmetry with Weight Sharing Schemes
2. Translation Symmetry with 1-Dimensional Convolutional Neural Networks
3. Input Permutation Symmetry
   a. 3-Dimensional AND, OR, XOR with Weight Sharing Schemes
   b. Integer Set Sum with Deep Sets
4. Graph Isomorphisms with Graph Convolutional Neural Networks
5. 2x2x2 Rubik's Cube

Chapter 2

Functions Invariant to Reflection

Starting with the reflection symmetry, I conducted experiments on binary logic functions AND, OR, and XOR. They are quite simple functions, and a sufficiently wide and deep network would have no problems learning them. However, instead of this "brute-force" approach, I focused on finding the minimal working architecture and analyzed any emerging convergence problems. It was also a good opportunity to experiment with *weight sharing schemes* (see Section 1.2.1).

The first two representatives, AND and OR, were learned very easily with the most simple neural architectures. Still, I experimented with weight sharing schemes to show that even in such trivial cases these may offer an improvement in training time.

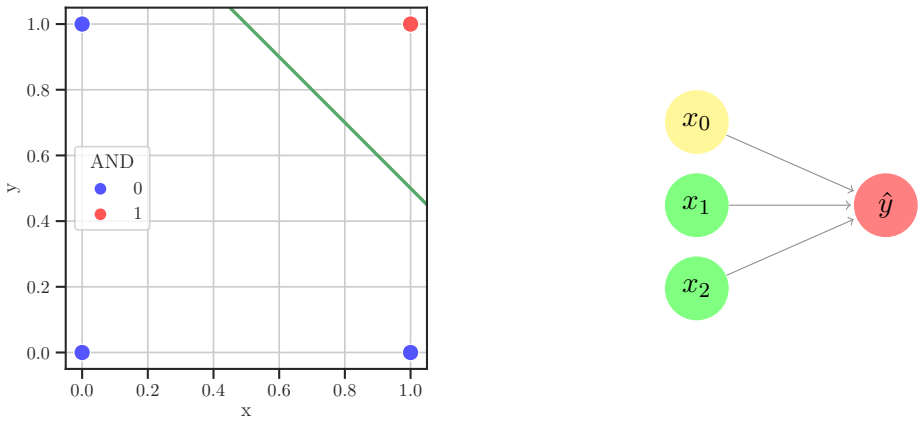
With the next representative, XOR, I could already compare not only the training times but also success in convergence. Surprisingly, if I did not largely over-shoot the network size, the conventional fully connected architecture had problems converging to the global optimum. Here, the advantages of weight sharing schemes became even more apparent.

Binary Logic Functions

Let me introduce the concept of a binary logic operator. For every binary logic operator a corresponding function $f : \{0,1\}^2 \to \{0,1\}$ can be defined (for their definition, see the corresponding baseline Sections 2.1.1, 2.1.2, and 2.1.3). This notion is used throughout the work as it unifies the view of a neural network model as a function approximation. As was mentioned in the previous paragraph, three logic functions were studied: AND, OR, and XOR.

2.1 Baselines with MLPs

The following experiments were carried out using my own neural network implementation in NumPy. The sigmoid activation function was used for all neural units. This was a straightforward choice for a perceptron learning binary logic functions, as the output of the sigmoid can be interpreted as an estimate of probability $P(Y = 1|x_1, x_2)$. As for the hidden neurons used for learning XOR, the choice of the sigmoid function was based on the intuition that each of the hidden units should learn a logic function itself (see Sections 2.1.3 and 2.1.3). This intuition



(a) : AND feature space **(b) :** A perceptron

Figure 2.1: AND - feature space and the architecture for its solution

was supported by experiments where using ReLU as the activation function for the hidden units gave noticeably worse results. Binary cross-entropy was chosen as the loss function, for it is a standard loss function for classification problems. Unless said otherwise, a learning rate of 0.1 was used for all the experiments, and the number of iterations of the stochastic gradient descent algorithm ranged from 500 to 2000, depending on the complexity of the problem (500 was used for AND and OR, 2000 for the more complex XOR problem).

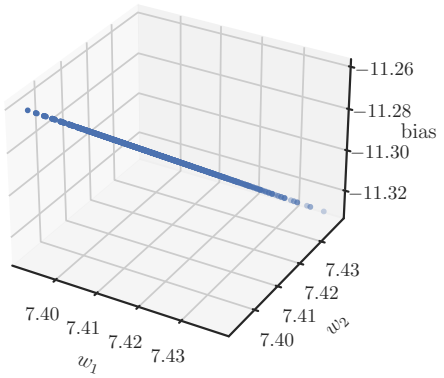
The convergence of a neural network with hidden layers might heavily depend on weight (parameter) initialization. Initially, I trained the neural network with two hidden neurons for the XOR problem with 20 different random initializations. The results were bad, with only 3 out of 20 networks classifying all data correctly after the training. Therefore I decided to examine the network's results systematically. The following initialization grid was used for the experiments: For each weight in the network, n linearly spaced values between -1 and 1 were considered. The network was then trained on all possible combinations of those weights.

2.1.1 Conjunction - Baseline

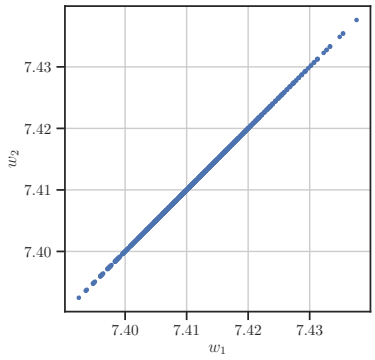
Binary conjunction, also called AND, is a function $f : \{0,1\}^2 \to \{0,1\}$, defined as follows: $AND(x,y) = 1 \iff x = y = 1$, $AND(x,y) = 0$ otherwise.

Feature Space

The two output classes of binary AND (0 and 1) are linearly separable in the feature space (see Figure 2.1a). A perceptron architecture (visualized in Figure 2.1b) with the sigmoid activation function is therefore theoretically sufficient for correctly learning (with 100% accuracy) the classification function.



(a) : 3D visualization of learned weights



(b) : 2D projection of weights discarding bias

Figure 2.2: AND - visualizations of learned weights

Initialization Grid and Results

The initialization grid was created using $n = 10$ different values between -1 and 1 for each of the 3 weights. In all 10^3 experiments on the initialization grid, the perceptron learned to classify AND correctly. This was expected as there are no local optima in the error space except the global optimum because perceptron is a linear classifier and the error space is convex.

Learned Parameters and Error Space

Figures 2.2a and 2.2b offer visualizations of weights learned by the perceptron, each point representing the learned weights for one of the experiments. We can see the learned weights w_1 and w_2 are exactly the same (they should be - AND is a function invariant to reflection). This allows to visualize the error space in 3D: x axis corresponding to $w_1 = w_2$, y axis to the bias, and z axis corresponding to the loss function (see Figure 2.3).

2.1.2 Disjunction - Baseline

Binary disjunction, also called OR, is a function $f : \{0, 1\}^2 \to \{0, 1\}$, defined as follows: $OR(x, y) = 0 \iff x = y = 0$, $OR(x, y) = 1$ otherwise.

Feature Space

The feature space of OR is linearly separable as well (see Figure 2.4a), meaning it is theoretically possible to train perceptron with the sigmoid activation function to classify all the inputs correctly.

Initialization Grid and Results

The same initialization grid was used as in the AND section, and all 10^3 experiments converged approximately to the only local and thus global minimum.

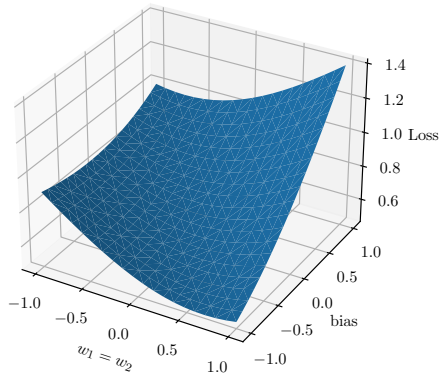


Figure 2.3: AND error space - both symmetric weights on X axis, bias on Y axis

Learned Parameters and Error Space

Figures 2.5a and 2.5b visualize the weights learned by the perceptron, each point representing the learned weights for a single experiment. Again, the learned weights w_1 and w_2 converged to approximately the same values (as they should - OR is a function invariant to reflection). The error space is visualized again in 3D with the x axis corresponding to $w_1 = w_2$, y axis to the bias, and z axis corresponding to the loss function (see Figure 2.6).

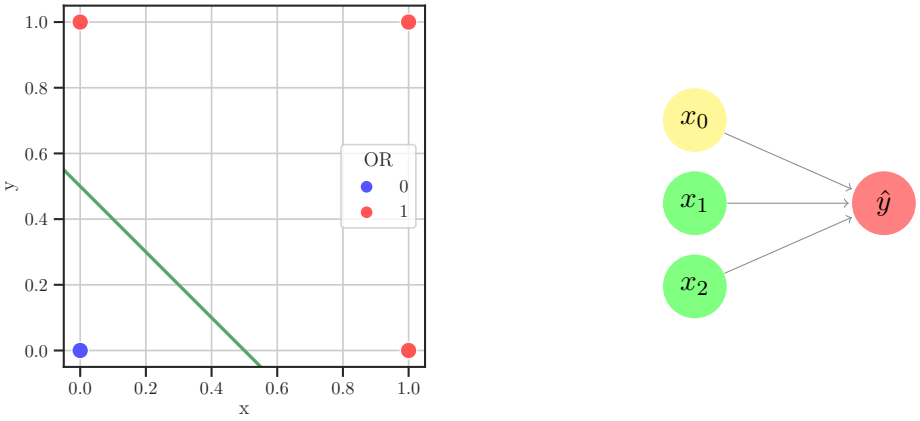
Note that Figure 2.5b has a little bit different shape from Figure 2.2b. In case of AND, the projection resembled a straight line. In case of OR, the line is thicker and the points form artifacts around it. This was most probably caused by the imperfect representation of floating point data types during learning.

2.1.3 XOR - Baseline

Binary exclusive disjunction, also called XOR, is a function $f : \{0,1\}^2 \to \{0,1\}$, defined as follows: $XOR(x,y) = 1 \iff x \neq y \wedge (x = 1 \vee y = 1)$, $OR(x,y) = 0$ otherwise.

Feature Space

So far, we have encountered classification problems with linearly separable classes. But the classes in the feature space of XOR are not linearly separable. Figure 2.7a shows the two linear separators that are needed. The lower line represents a learned threshold for logical function OR (classifying all points above the line as 1), the upper line corresponds to a threshold of logical function NAND (not AND; classifying all points below the line as 1). If we make a conjunction of those two logic functions, we get the desired XOR function. This intuitively brings us to the minimal working architecture needed to learn XOR. Since Minsky and Papert showed in [20] that it is not possible to learn a non-linearly separable function (such as XOR) with a



(a) : OR feature space **(b) :** A perceptron

Figure 2.4: OR - feature space and architecture for solving it

X	Y	f1	f2	f3	f3 on outputs of f1 and f2
0	0	1	0	1	**0**
0	1	1	1	1	**1**
1	0	0	0	0	**1**
1	1	1	0	1	**0**

Table 2.1: Antisymmetric binary logic functions leading to XOR

perceptron, the minimal neural architecture for learning XOR would be composed of two hidden and one output neurons (see Figure 2.7b)[1], all learning some logic functions. Given our intuition, the hidden units would be learning OR and NAND, respectively, while the output unit would be learning a simple AND function. This way, even individual neurons would be invariant to input permutations.

But, the possible solution described above is just an intuition based on the illustration of two linear separators in Figure 2.7b. Consequently, two questions arise.

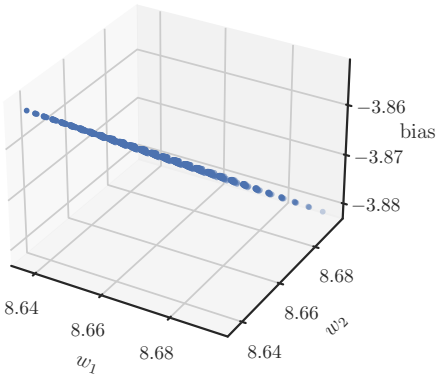
1. Is there a reason to think that the individual neurons of our architecture would learn only logical functions invariant to reflection (AND, OR, NAND, NOR, etc.)?

2. Is it reasonable to think that these neurons would learn any logical functions at all?

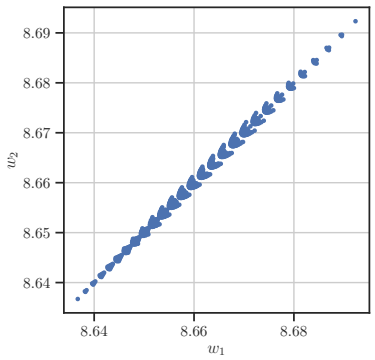
In other words, the individual neurons with their respective weights could possibly compute unintuitive functions that together, combined into the network, compute XOR.

The answer to the first question is no, there are many combinations of other logical functions (e.g. logical function given by the truth table F, F, T, F) that are not

[1] An architecture with a single hidden and a single output neuron does not add any expressiveness to the network compared to the perceptron, thus we proceed immediately to two hidden units.



(a) : 3D visualization of learned weights



(b) : 2D projection of weights discarding bias

Figure 2.5: OR - visualizations of learned weights

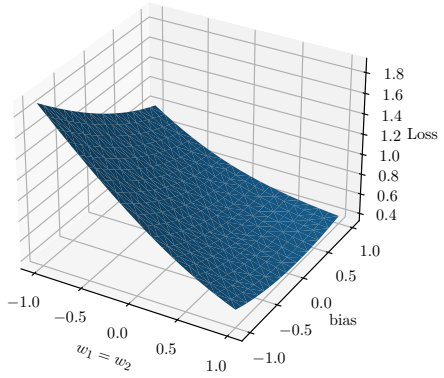
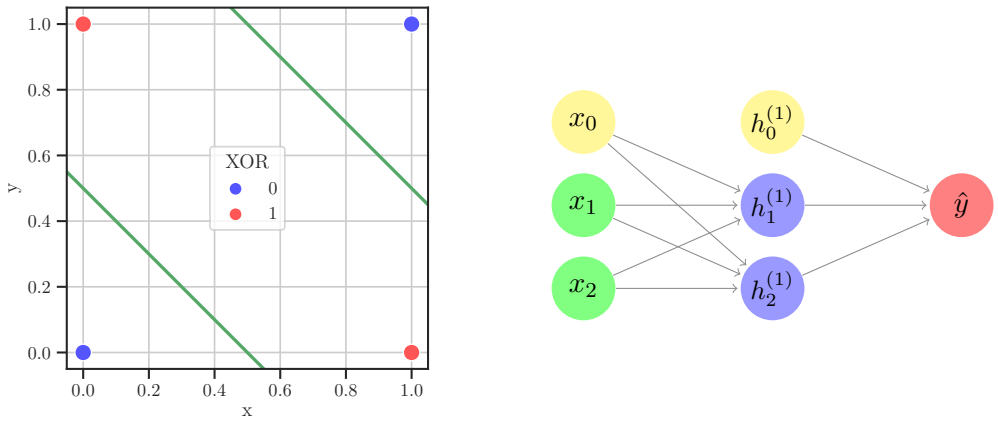


Figure 2.6: OR error space - both symmetric weights on X axis, bias on Y axis

invariant to reflection, and those combinations still succeed at XOR classification. An example of that is demonstrated in the Table 2.1. If the two hidden neurons learn functions f1 and f2 and the output neuron learns function f3, then by using the function f3 on the output of functions f1 and f2 (given all possible inputs), we get the XOR function. Note that neither of the functions f1, f2, and f3 is invariant to reflection.

The answer to the second function is 'probably yes'. Both me and Richard Bland (in [2]) came to this conclusion empirically - no matter how you initialize the weights, the two weights of each neuron tend to converge to approximately the same or opposite values. When I say approximately the same weights, I mean the following:

Definition 2.1. Two weights have *approximately the same (opposite) value* iff if the values (or their absolute values) differ in no more than units of percents compared to each other, *and* the gradients (the absolute values of the gradients) in last few



(a) : XOR feature space **(b) :** An architecture with two hidden units

Figure 2.7: XOR - feature space and architecture for solving it

epochs/iterations of the training with respect to the weights differ in no more than units of percents as well.

This is useful since gradient descent is a numerical algorithm and two differently initialized weights rarely converge to the exact same value.

Compositions of Logic Functions Leading to XOR

In [2], Richard Bland expanded the idea that the conjunction of OR and NAND is not the only way to get XOR via composition of binary logic functions. In total, there are 16 different binary logic functions. Out of those 16 functions, 14 of them have linearly separable output classes (the remaining two are XOR and NXOR - not XOR). Thus, for our minimal architecture with two hidden neurons and an output neuron, there are 14^3 potential permutations of those functions (the order is generally important, as some of the 14 functions are not symmetric in their inputs). Bland explained that only 16 of those permutations compute XOR (which I verified by an experiment). This is a first sign that learning XOR is not a trivial task.

Initialization Grid and Results

The structure of the XOR experiments was following: the initialization of weights slightly differed from the previous two sections - the biases were always initialized to 0. This decision was purely practical since it requires at least two hidden units for the neural network to learn the XOR (as will be argued later), and thus the initialization grid including all the biases' combinations would be too large for practical experiments – a network with two inputs, a hidden layer with two units, and a single output has $2 \times 2 + 2$ weights in the first layer and 1×2 weights in the second layer. This yields n^9 combinations, considering n linearly spaced values between -1 and 1 for each weight. The number of combinations thus grows quickly with n and trying out e.g. 10^9 experiments is not feasible. Instead, without the

NR of misclassified inputs	Count
0	269 215
1	420 227
2	308 940
3	1 618
4	0
Total	**1 000 000**

Table 2.2: Table of results

additional bias combinations, we have n^6 combinations (neural networks to be learned) in our grid experiment. That is a feasible task for a computer cluster with the possibility of parallelization.

Similarly to previous two logic functions, $n = 10$ different values between -1 and 1 for each of the 6 weights (except the 3 bias weights) were chosen. The results were much worse than with AND and OR. Out of the 10^6 trained networks, only $269,215$ classified all input correctly, other $420,227$ networks classified only 3 out of 4 inputs correctly, $308,940$ networks classified 2 inputs correctly, $1,618$ networks classified 1 input correctly, but at least there was no network that would classify all the inputs incorrectly (see also Table 2.2). This means that only about 27% of networks converged to a perfect classifier.

Following weight combination is an example of one of the perfect classifiers learned within the grid. Each matrix represents a single layer (excluding the input layer), each row in a matrix represents weights for a single neuron, the last columns of the matrices correspond to bias weights (intercepts):

$$W_0 = \begin{pmatrix} -3.0359 & 3.6427 & 1.2290 \\ -5.0289 & 5.0611 & -3.1574 \end{pmatrix}$$

$$W_1 = \begin{pmatrix} -4.5676 & 5.9261 & 1.6877 \end{pmatrix}$$

The neurons itself do not compute logic functions that are invariant to input permutations (an input $(0, 1)$ gives a different output than a permutation of the same input - $(1, 0)$). Those three learned functions are approximately (Definition 2.1) antisymmetric, see Definition 1.9, and they were already mentioned in Table 2.1.

Classes of Perfect Classifiers

If we look at the weights of the 269215 perfect classifiers (with 100% accuracy), they all learned XOR in one way or another. Nevertheless, it is interesting to divide them into classes by studying which functions were learned at the level of individual neurons. This might give us intuition about which weight sharing schemes are probably going to work and which not. For our purposes of XOR experiments, let me define a *class of classifier*:

Definition 2.2. A *class of a classifier* is a set of perfect classifiers with the same combination of learned logic functions at the level of individual neurons. The order of those three functions is generally important.

Order	1st Neuron	2nd Neuron	3rd Neuron	Total	%
1	F, T, F, F	T, T, F, T	T, F, T, T	30 997	11.51
2	F, T, F, F	T, T, F, T	T, F, T, T	29 865	11.09
3	F, T, T, T	F, F, F, T	F, F, T, F	28 426	10.56
4	F, F, T, F	T, F, T, T	T, F, T, T	27 453	10.20
5	T, F, T, T	F, F, T, F	T, T, F, T	27 453	10.20

Table 2.3: Most commonly learned boolean functions at each neuron for perfect classifiers

Note that we treat individual neurons as units that learn logical functions themselves based on the empirical observation that the learned weights approximately (see Definition 2.1) correspond to them in most of the successfully converged cases. Thus, we have defined the classes the same way as Bland did in [2].

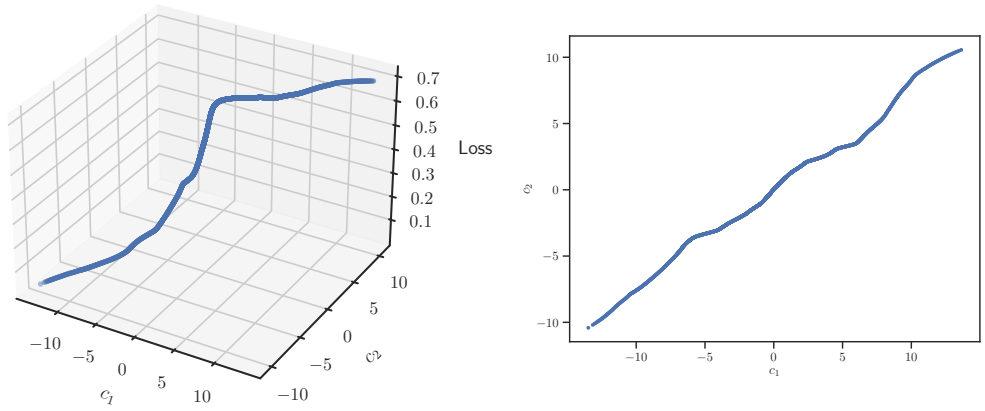
By this definition, there are 81 classes of perfect classifiers obtained by the experiment on the grid, e.g.: the first neuron learned OR, the second one learned NAND and the third one learned a logical function called converse nonimplication (with truth table False, False, True, False). Keep in mind that the order of inputs for the neurons is generally important (we treat converse nonimplication with truth table F, F, T, F and material nonimplication with truth table F, T, F, F as two different functions).

It might seem suspicious that we have 81 classes when Bland (and I experimentally, as well) came to the conclusion that only 16 combinations of linearly separable functions compute XOR. But there is no contradiction, the difference is that we do not use a step activation function, instead we use sigmoid as a differentiable alternative needed for backpropagation algorithm. Thus the activation values are never strictly equal to 0 or 1, and so individual neurons only approximate (see Definition 2.1) logic functions. This allows for classes that do not strictly compute XOR from the perspective of logic function composition. Moreover, if we order those 81 classes by occurrence, we can see that the 16 desirable classes occupy the first 16 places of the ordering and that together they make up 98.91% from the 269215 perfect classifiers.

The 5 most common classifier classes are shown in the Table 2.3

Generalized Classes of Perfect Classifiers

As preparation for weight sharing architecture, it is useful to look at more generalized classes of classifiers: namely to differentiate only whether each of the three logic functions has approximately the same or opposite weights (see Definition 2.1) corresponding to its two inputs. Generally, from the 16 combinations of linearly separable logic functions computing XOR, 4 functions have the same two input weights (except the biases) in each of the three neurons (we can say that all the neurons are input invariant themselves), another 4 functions have the same weights in the first neuron and the same weights in the second neuron, and another 4 functions have the same input weights only in the third neuron. The last 4 functions have opposite weights in all three neurons.



(a) : XOR - error w.r.t learned weights reduced to 2 dimensions

(b) : XOR - learned weights reduced to 2 dimensions

Figure 2.8: XOR - learned weights in reduced space with and without loss

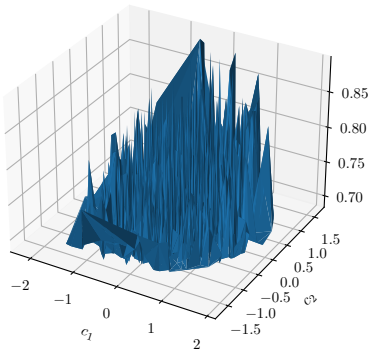
How are these options distributed in the weights obtained by the grid experiment? All neurons are invariant to input permutations in 31590 cases (11.73%), first two neurons only are input invariant in 83040 cases (30.85%), third neuron only is input invariant in 47080 cases (17.49%), and lastly, there are 104574 cases (38.84%) with no input invariant neurons.[2]

Learned Parameters and Error Space

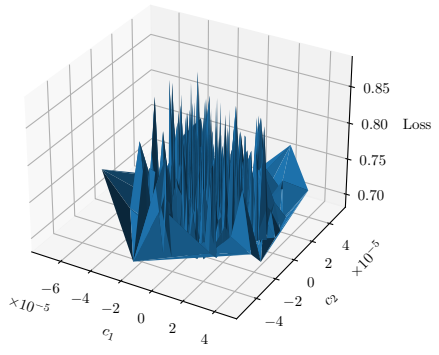
Since the neural architecture in use has 9 weights, they have to be visualized using dimensionality reduction techniques. Particularly, we selected PCA - principal component analysis, t-SNE - T-distributed Stochastic Neighbor Embedding, and LLE - locally linear embedding. Figure 2.8b visualizes the learned weights while reducing the 9-dimensional space to a 2-dimensional one. Each point represents the learned weights for a single experiment. Adding the final values of cross-entropy to a third axis, we get an approximation of the error space on converged solutions (see Figure 2.8a; x and y axes correspond to the reduced weight space and z axis corresponds to the loss function).

It is interesting to compare this error space consisting of only converged weights to an approximation of the complete error space on the $[-1, 1]^3$ interval. I thus also tried to visualize the latter with the three different techniques, see Figures 2.9a, 2.9b and 2.9c (again, x and y axes correspond to the reduced weight space and z axis corresponds to the loss function). All three methods gave results with a lot of spikes, suggesting a very complex error space. Yet, the converged weights suggest something a little bit different. As seen in Figure 2.8b, the converged weights (after reducing the dimension) lie on an approximation of a hyperplane. The fact that PCA reduction, which tries to preserve as much variance as possible, returns a

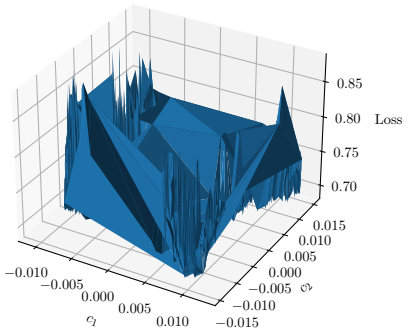
[2] These results will be compared with the convergence of weight sharing architectures, depending on whether we select the same or opposite initial weights for each neuron.



(a) : Reduction by PCA



(b) : Reduction by t-SNE



(c) : Reduction by LLE

Figure 2.9: XOR - loss landscape reduced to 2 dimensions by different methods

line in a 2-dimensional space suggests that the original converged weights in the 9-dimensional space might have formed a hyperplane as well. That would refute the usual intuition about minima/saddle points scattered around the feature space - instead, these minima/saddle points (including global minima) seem to lay approx. on a hyperplane.

Even though the results are not impressive, the learned weights (reduced to two dimensions with 3^{rd} dimension being the loss) resemble the picture of learned weights from AND and OR problems. After adding the loss function to the 3^{rd} dimension, they still approximate a hyperplane, see Figure 2.8a.

Even though the pictures of the XOR error space (see Figures 2.9a, 2.9b and 2.9c) are not completely accurate, the fact that very different dimensionality techniques produced similar spike builds an intuition that many local optima or saddle points exist in the error space of the chosen neural architecture. The XOR problem has been studied by many statistical learning researchers and while some authors (e.g. Lisboa et al. [18]) claimed that there are local minima in the error space preventing the network from converging to a model with perfect accuracy, all Hamey (formally,

by studying the loss function, in [11]), Sprinkhuizen-Kuyper (also formally, in [24]), and Bland (less formally, but more instructively, visualizing and commenting on error space slices, in [2]) showed that there are no local minima other than the ones with zero loss (that lead to a perfect classification of XOR). The problematic regions are either saddle points or practically flat areas, where the gradient is too small (vanishes). Still, such areas are very disruptive for all conventional gradient-based iterative algorithms, and specifically for a gradient descent algorithm may lead to nonconvergence given for some weight initializations. It is thus logical to experiment with alternative algorithms to pure gradient descent.

A Case for Momentum Optimizers

In previous paragraphs, I explained that the problematic regions where gradient descent gets usually stuck are saddle points, and even before that, I made a point that these saddle points lie approx. on a hyperplane together with the global optima. This means that around the division point (between successful and unsuccessful classifiers) on the hyperplane, even small oscillations might have an impact. Therefore it is logical to try out gradient methods with momentum because they avoid some of the problems with oscillations.

2.1.4 XOR - Momentum Optimizers

I experimented with two major momentum techniques:

1. simple adjusted gradient descent algorithm with (constant) momentum
2. ADAM - adaptive moment estimation algorithm

For the adjusted gradient descent, the momentum of 0.9 worked best in my experiments. This was only reinforced by the latter experiments with ADAM optimizer, which uses a dynamically set momentum. Those experiments yielded the same results in terms of misclassification statistics over the whole grid. Other experiment parameters remained the same as in the experiments with vanilla fully connected neural networks.

Results of Gradient Descent with Momentum

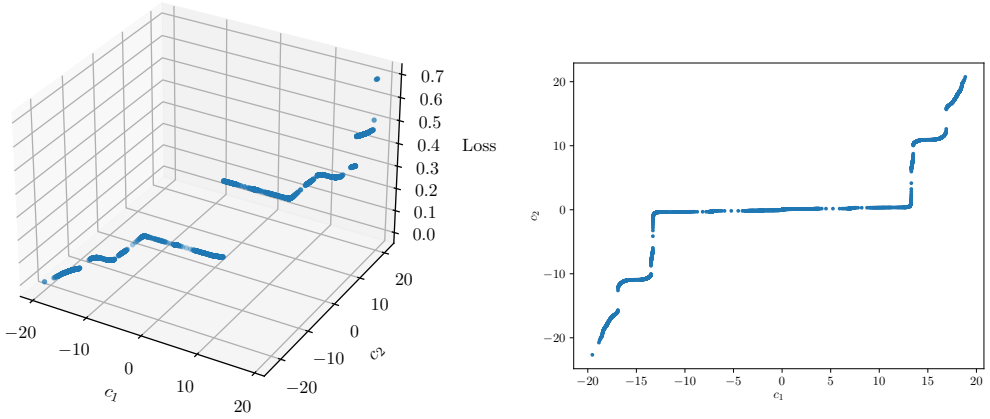
The results with gradient descent optimizer with momentum set to 0.9 were better than with conventional gradient descent (but still not perfect). Out of the 10^6 trained networks, $572,207$ classified all input correctly, other $109,699$ networks classified 3 out of 4 inputs correctly, $318,077$ networks classified 2 inputs correctly, 17 networks classified 1 input correctly, and there was not any network that would classify all inputs incorrectly (see also Table 2.4). This means that about 57% of networks converged to a perfect classifier.

Learned Parameters and Error Space

Similarly to the baseline, I will visualize the learned parameters and compare them to the previous visualizations.

NR of misclassified inputs	Count
0	572 207
1	109 699
2	318 077
3	17
4	0
Total	**1 000 000**

Table 2.4: Table of results



(a) : Loss w.r.t. learned weights reduced to 2 dimensions

(b) : Learned weights reduced to 2 dimensions

Figure 2.10: XOR - learned weights in reduced space with and without loss (ADAM)

In Figure 2.10b, you can see the learned weights reduced to two dimensions. Moreover, in Figure 2.10a, the third dimension entails the values of the loss function. It shows a very similar picture to the previous visualizations of learned weights, but now the points that belong to perfect classifiers are clearly separated. This coincides with our intuition about momentum methods:

1. thanks to momentum, oscillations leading to saddle points were avoided in many more cases,

2. the inertia of momentum methods caused successful networks to converge closer to the optimum, thus widening the gap in error space between weights of successfully converged networks and unsuccessful ones.

Classes of Perfect Classifiers

Similarly to the XOR subsection, let's take a look at the converged weights of perfect classifiers in a bigger depth. Using our previous definition of classifier classes, the 572207 perfect classifiers fall into 16 such classes, the exact same ones Bland talked about in [2]. This is progress compared to the previous results, as previously there

Order	1st Neuron	2nd Neuron	3rd Neuron	Total	%
1.	F, F, T, F	T, F, T, T	T, F, T, T	60 441	10.56
2.	F, T, F, F	T, T, F, T	T, F, T, T	60 441	10.56
3.	T, F, T, T	F, F, T, F	T, T, F, T	60 441	10.56
4.	T, T, F, T	F, T, F, F	T, T, F, T	60 441	10.56
5.	F, T, F, F	F, F, T, F	F, T, T, T	51 957	9.08

Table 2.5: 5 most common combinations of learned logic functions

were some residual classes (though only approx. 1.09% of perfect classifiers belonged to them) which worked only thanks to the fact that sigmoid activation function was used, and strictly speaking, did not compute XOR if we binarized the activations and assigned the corresponding logic functions to them. These improvements are not surprising since they correspond with the intuition mentioned in Section 2.1.3.

The 5 most common combinations of learned functions are in the Table 2.5.

Generalized Classes of Perfect Classifiers

Looking at the distribution of perfect classifiers within the 4 generalized classes defined in the "Generalized Classes" in the first XOR subsection, all neurons are input invariant in 89582 cases (15.66%), first two neurons only are input invariant in 134202 cases (23.45%), third neuron only is input invariant in 106658 cases (18.64%), and lastly, there are 241764 cases (42.25%) with no input invariant neurons.

Again, those results will be compared later on, but we can again see a slight trend towards neurons with opposite weights.

Results of ADAM Optimizer

The adaptive moment estimation optimizer had exactly the same results as SGD with momentum in terms of the number of misclassifications for each neural network with different weight initialization. Thus the results are already summarized in Table 2.4 and 572207 (57%) of networks converged to a perfect classifier.

The distribution of perfect classifiers between the classes and generalized classes is exactly the same as in the momentum section so that the reader can review the corresponding Table 2.5, and Sections 2.1.4 and 2.1.4.

Summary - Momentum Optimizers

While we have seen much better results with ADAM and gradient descent with momentum than with the basic (SGD) optimizer, almost half of the networks did not converge to the optimal classifier and there is a big room for improvement.

2.2 Weight Sharing Schemes

The next thing I will experiment with is learning binary logic functions with the symmetry-aware model. I will be using neural networks with weight sharing schemes

as they were introduced in the introduction chapter. As I mentioned (see Section 1.2.1), the sharing schemes will contain only two basic dependency types between weights:

1. the weights have the same value,
2. the weights have the opposite value (e.g. $w_1 = -1 * w_2$).

In this section, I analyze how different weight sharing schemes affect convergence and speed of convergence of previously introduced problems. In my experiments, I listed all the weight sharing combinations. For a neural network with n weights, this means generating permutations of 1 to $n-1$ weight objects shared among the n weights (n objects among n weights is not an interesting case since there is no weight sharing present). The second level of combinations arises from the fact that we explore two types of dependencies: symmetric dependency (Definition 1.8) and antisymmetric dependency (Definition 1.9).

At first sight, it looks like too many combinations, but many of these combinations are equivalent from the perspective of gradient descent learning.

Example 2.3. Suppose that we would have a network with 4 weights and decide to share 2 weight objects among them. Then, following cases are equivalent: (i) Sharing first weight object between w_1 and w_2, and second object between w_3 and w_4, (ii) Sharing first weight object between w_3 and w_4, and second object between w_1 and w_2. Similarly, the change of sign before all the weights that share the same weight object yields an equivalent scheme as well (thanks to the fact that we use a symmetric initialization grid).

More generally for our purposes:

Definition 2.4. *Weight sharing schemes are equivalent* iff there exists a renaming between the combinations that would map one onto another.

Remark 2.5. For simplicity (and in case of XOR network also out of necessity resulting from high time complexity), I did not include sharing combinations for bias weights. That means bias weights are never shared in the following experiments and are always learned independently during backpropagation. Even though adding bias to weight sharing might offer additional convergence improvements (and it would be an interesting subject for future work), it is not necessary for showing the advantages of weight sharing, which is the main idea behind this section.

To cover the mentioned weight sharing combinations systematically, I ran the same initialization grid as in the baseline Section 2.1 on each of the weight sharing combination/scheme.

2.2.1 Conjunction (AND)

Following up on the perceptron architecture from the baseline Section (2.1.1), there are two weights that could be shared for the AND problem (and one bias which is not included in the weight sharing schemes, see Remark 2.5). Considering also the two types of dependencies between weights, there are two weight sharing schemes we should consider:

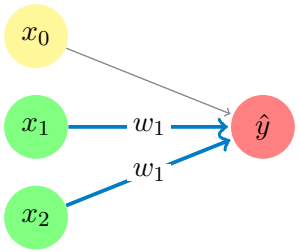


Figure 2.11: A perceptron with shared weights

	Init 1	Init 2	Init 3	Init 4
Perfect classifier after	26 iters	25 iters	25 iters	24 iters

Table 2.6: Number of iterations to achieve perfect AND classification with sharing scheme, 4 example initializations

1. $w_1 = w_2$
2. $w_1 = -w_2$

For both of the schemes, I ran the experiments on the initialization grid (see Section 2.2).

The first scheme (see Figure 2.11) successfully and quite quickly converged in all initialization cases with example results in Table 2.6. This is an improvement over vanilla perceptron that needed approx. 100 to 200 iterations (with the same learning rate and architecture) to converge to perfect classification.

Remark 2.6. Logically, the second scheme (see Figure 2.12) did not converge to a perfect classifier (the final result is 1 misclassified input in all 4 initialization cases) because there is no line that would separate the point $\vec{x} = (1, 1)$ (labeled True) from the other points (labeled False) with a prescription: $ax_1 - ax_2 + b = 0$. This is a case of an ill-defined weight sharing scheme (see Definition 1.10).

2.2.2 Disjunction (OR)

Both the experiments and the results for OR are very similar to ones performed with described in the AND Section 2.2.1. Again, there are two weights that could be shared in the perceptron and thus two weight sharing schemes:

1. $w_1 = w_2$
2. $w_1 = -w_2$

And again, the first scheme (see Figure 2.11) successfully converged in all initialization cases with the example results in Table 2.7, marking an improvement over vanilla perceptron that needed approx. 100 iterations to converge.

2.2.3 Exclusive Disjunction (XOR)

Moving forward to XOR, we used the network with two hidden units, as in the Baseline (see Section 2.1.3) and Momentum (see Section 2.1.4) experiments. With 6

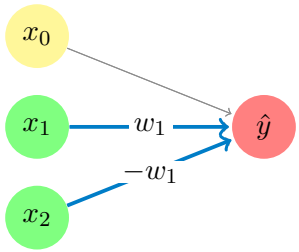


Figure 2.12: A Perceptron with Shared but Opposite Weights

	Init Nr. 1	Init Nr. 2	Init Nr. 3	Init Nr. 4
Perfect classifier after	31 iters	30 iters	30 iters	1 iter

Table 2.7: Number of iterations to achieve perfect OR classification with sharing scheme

weights excluding the biases, there are 1538 different weight sharing schemes. For each of the schemes I ran the experiments on the initialization grid (see Section 2.2). The analysis of the results follows below.

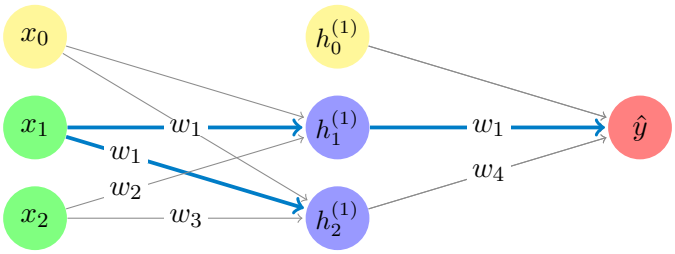


Figure 2.15: Successful architecture with 4 learned parameters (+ 2 biases), grid of 256 initialization, min iters to convergence: 855, mean iters: 956.363, median iters: 957

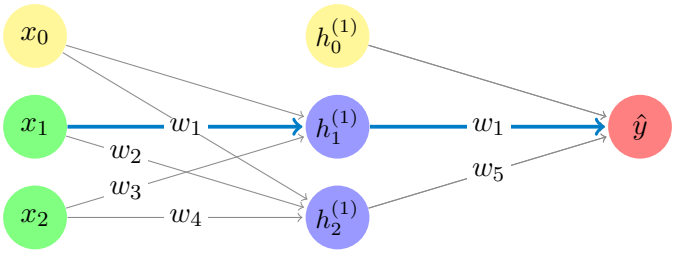


Figure 2.16: Successful architecture with 5 learned parameters (+ 2 biases), grid of 1024 initialization, min iters to convergence: 1288, mean iters: 1487.844, median iters: 1487

Results

There were 141 weight sharing schemes that overperformed the Baseline and Momentum experiments in terms of convergence percentage (how many runs from the grid experiments converged to a perfect classifier). In case of Baseline it was 27%, in case of Momentum it was 54% of successful classifiers.

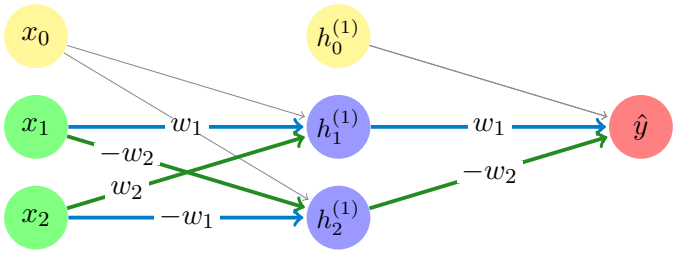


Figure 2.13: Successful architecture with 2 learned parameters (+ 2 biases), grid of 16 initialization, min iters to convergence: 630, mean iters: 721.875, median iters: 720.5

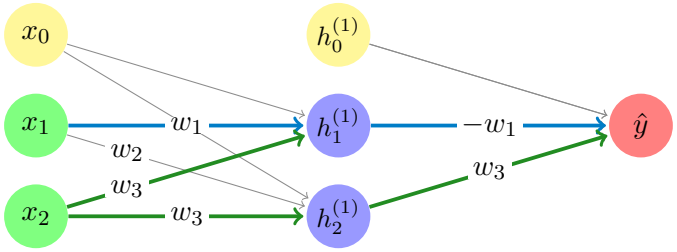


Figure 2.14: Successful architecture with 3 learned parameters (+ 2 biases), grid of 64 initialization, min iters to convergence: 775, mean iters: 854.016, median iters: 856.5

But the most interesting schemes are the ones which led to perfect classifiers with all the initialization combinations (100% convergence). There were 86 such schemes. Some of these successful schemes will be used in the next section on N-dimensional XOR. Examples of such schemes are showed below together with the minimum, mean and median number of iterations needed for convergence, see Figures 2.13, 2.14, 2.15, 2.16 - the shared weights/edges are colored with the same color, the ones in black are always independent since they are not shared anywhere. If a vertex shares the same weight but with opposite values, there is a minus next to the vertex in the illustrations.

We can see that these schemes are more successful than vanilla fully connected networks by both the stability with respect to weight initialization (convergence rate along the grid) and with convergence time (in terms of iterations needed).

2.3 Functions Invariant to Reflection - Conclusion

I have conducted learning experiments for 3 logic functions (AND, OR and XOR) using 2 symmetry-unaware baselines, and 1 symmetry-aware approach. The first, naive approach was to use vanilla MLPs. For AND and OR it worked well and a simple perceptron was able to classify the inputs flawlessly after couple hundreds of iterations of learning. Learning XOR was more sophisticated. Not only that a XOR needs a larger network to be learned perfectly, it proved that a 2-layer architecture that is theoretically capable of learning XOR is no guarantee of success. The network ended up converging only in 27% of cases. The next step was a second baseline approach - using gradient optimizers with momentum (or adaptive momentum). This technique improved the convergence rate to 57%.

Then, I followed up with the symmetry-aware approach. I analyzed different possible weight sharing schemes with the idea to further improve the convergence rate and reduce the number of iterations needed for convergence. For AND and OR, there was one ill-defined sharing scheme (making convergence impossible), but also one well-defined one which reduced the number of iterations needed for convergence from hundreds to tens. For XOR, the choice of weight sharing scheme is not so straightforward. Yet, we showed that good (and intuitively justified) schemes exist and offer convergence and time improvements over both Baseline and Momentum experiments.

While choosing the right weight dependencies was not an easiest task, the symmetry-aware approach with weight sharing schemes turned out to be successful for learning functions invariant to Reflection in the end.

Chapter 3

Functions Invariant to Translations

Continuing with the next type of symmetry, I trained neural networks to approximate functions that are invariant to translation of a pattern in the input. The neural networks suitable for this kind of problem are convolutional neural networks (CNNs; see Section 1.2.2 in the introduction).

As an example problem, I trained a convolutional neural network to classify a 1-dimensional binary array based on whether a certain (predefined) pattern was present in it, or not. For this problem I used 1-dimensional convolutional filter (as I talked about it in Section 1.2.2), a 1-dimensional array smaller than the input of the network which is iteratively shifted starting from the left borders of the input data and ending at the right border of the input data. Today, the most common usage of CNNs are computer vision applications, where 2-dimensional convolution filters are used for a 2-dimensional inputs, images. Nevertheless, the idea is still the same.

3.1 Pattern Search in 1-Dimensional Array

The concrete definition of the problem was the following: training a binary classifier for binary arrays of length 8, classifying an array with the label 1 *iff* it contains the pattern "1010" anywhere in it, and with the label 0 otherwise. As previously, I compared the performance of a vanilla fully connected network with the convolutional network.

For the following experiments, I used the ADAM optimizer with learning rate of 10^{-3}. In each epoch of the training phase, I used the full batch of the data, since the whole dataset is relatively small.

Random Init	1	2	3	4	5
Accuracy	97.54%	93.60%	83.25%	81.77%	89.66%

Table 3.1: Accuracies of first baseline with 5 random initializations

Random Init	1	2	3	4	5
Epochs before Convergence	1569	1100	1689	1930	1793

Table 3.2: Epochs needed for perfect classification in the second baseline

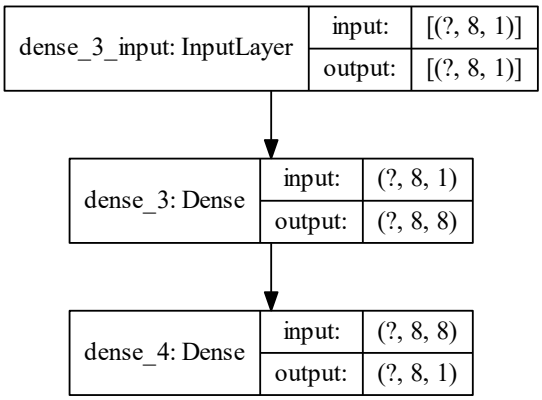


Figure 3.1: First fully connected architecture for pattern search

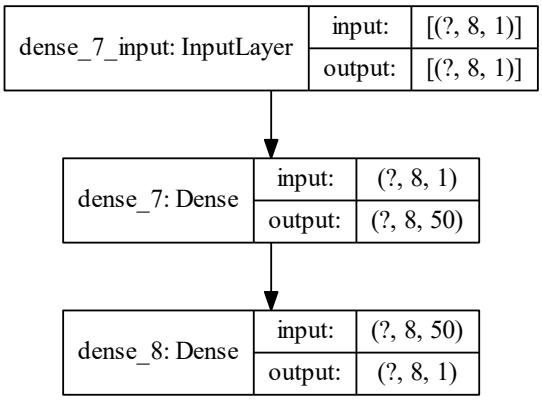


Figure 3.2: Second fully connected architecture for pattern search

3.2 Baseline with Fully Connected Networks

Starting out with smaller architectures, in the first experiment I used a network with two layers, the first one with 8 neurons (with ReLU - Rectified Linear Unit - activation function), and the second one with 1 neuron (and a sigmoid activation function) aggregating the output to a number between 0 and 1 (see Figure 3.1). Following the convention for binary classification, the classifier predicts 1, if the last activation is greater than 0.5, otherwise it predicts 0. I did the experiment with 5 random initializations, and let the network train for 5000 epochs in each

initialization. The accuracies of each run are in the Table 3.1. This network is often stuck in local minima.

Often, the performance of neural networks increases with increasing the width of the layers. The price of this approach is lower computational efficiency along with more iterations needed for successful convergence. Nevertheless, I tried this approach, replacing 8 neurons in the first layer with 50 neurons (see Figure 3.2). Again, I did the experiment with 5 random initializations, and let the network train for 5000 epochs in each initialization. In each of the 5 cases, the training converged to a perfect classifier, but in each case, a large number of epochs was needed for convergence, see Table 3.2. In the next experiments with the CNNs, the focus was thus on convergence with less epochs and less trainable parameters.

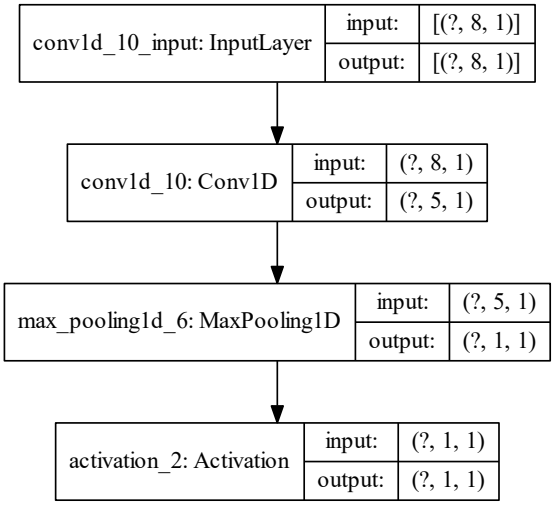


Figure 3.3: 1-D CNN for pattern search

3.3 1D Convolutional Neural Networks

Finally, the CNN architecture for the pattern search was following: the input (vector of size 8) was fed to a 1-dimensional convolutional layer with a single filter of size 4, the output of the convolutional layer (vector of size $8-(4-1)=5$) was then fed to a max-pooling layer. The result from maximum pooling was run through a sigmoid activation function and outputted as the result of the network (see Figure 3.3). As in the previous cases of binary classification, a simple rule is applied to the output $o \in (0,1)$ to determine the class: $class = 1$ *iff* $o > 0.5$, otherwise $class = 0$.

As previously, I did the experiment with 5 random initializations, and let the network train for 5000 epochs in each initialization. The experiments confirmed the fact that convolutional neural networks are very unstable with respect to parameter initialization.

Random Init	1	2	3	4	5
Epochs before Convergence	None	98	None	None	217

Table 3.3: Epochs needed for perfect classification in with CNN (None, if perfect classification not achieved)

In 3/5 cases, the networks did not converge to a global minimum, having roughly $74 - 80\%$ accuracy. In the remaining 2 cases, however, the networks converged very quickly, needing 98 and 217 epochs, respectively, see Table 3.3.

3.4 Functions Invariant to Pattern Translation - Conclusion

A weight sharing approach in form of convolutional neural network is nowadays widespread and empirical results show that it is very useful, too. I demonstrated that a much smaller convolutional network can be a better alternative over conventional, symmetry-unaware fully connected network even in very simple cases of learning to classify a presence of a pattern, or in general for any functions invariant to a pattern translation. However, the price is a higher chance of running into a local optimum, if the convolutional filter is randomly initialized to undesirable values.

Chapter 4

Functions Invariant to Input Permutations

For this class of functions (see Section 1.1.1), which is often called just *symmetric functions*, I firstly returned to a scaled-up, 3-dimensional version of the XOR problem. I explored whether the lessons learned from the 2-dimensional XOR could be applied with advantage, either by directly reusing learned weights, or by learning with a similar sharing schemes.

The section about functions invariant to input permutations is then concluded by discussing the general problem of learning functions with sets (of varying length) as their inputs. This problem implicitly assumes invariance of those functions to input permutations since sets are unordered collections. I compared the performance of a vanilla fully connected network with an architecture designed for the purposes of learning functions defined on sets, Deep Sets (and with other architectures as well), on the problem of integer set sum.

4.1 N-Dimensional XOR

Building up on the 2-dimensional XOR problem (Section 2), it is natural to deal with more complex instances of logic functions, concretely more complex instances of the XOR problem. Increasing the number of dimensions is one way to do so. One of the arguments for the increasing complexity is the work of Blum and Rivest from 1992 [3]. They showed that for n-input neural network with two hidden units it is NP-complete to decide whether the architecture can learn to approximate the desired function perfectly (in terms of accuracy). This complexity even lead some researchers to the use of evolutionary algorithms for the task of finding the optimal neural architecture [19].

Therefore, in the following sections, the neural architecture will be based mainly on the intuition and results obtained from the 2-dimensional experiments. As a representative of N-dimensional XOR, I am solving the 3-dimensional XOR problem in the following subsections. Similarly to the section about binary logic functions, I will first experiment with a baseline without weight sharing, and only then I will adopt the weight sharing approach, so that the results can be compared in the end. There will be two kinds of baselines. One with vanilla fully connected neural networks (bigger than the ones used for binary XOR), and the other one with a recurrent neural network.

4.1.1 XOR and Recurrent Neural Networks

Before talking about recurrent networks and XOR, I will start with a definition of a *recurrent neural network*:

Definition 4.1. A *recurrent neural network*, or RNN, is a model for sequential data that, for a time point t, takes a t^{th} input sequence and its own output h_t (called a *hidden state*), and produces output h_{t+1} by applying arbitrary neural transformations to it.

Apart from the sequential nature of RNN, which distinguishes it from a conventional neural network, the architecture can be very similar to a fully connected neural network. Each recursive cycle may consist of a stack of linear layers.

When visualizing a recurrent network, we speak about folded and unfolded computation graphs. The folded graph contains a cycle from the output layer to the input layer, see Figure 4.1 from the experiments as an example, the second one does not, since we expanded the expected number of recursive cycles, see Figure 4.2 as an example of unfolded graph with 2 recursive cycles. Generally, a recurrent network is not restricted to a fixed size input. Thus the number of recursive cycles can vary between the data points.

Remark 4.2. Note, that the structure of a recurrent neural networks itself induces another type of symmetry, one that could be called a compositional (fractal) symmetry. Here, the same computations are recursively performed on parts of the input while calculating their compositions with the previous (the hidden state).

Following the intuition from 2-dimensional XOR where we used an architecture with two hidden neurons, it turns out N-dimensional XOR can be intuitively described as a recurrent network. To elaborate on that, the N-dimensional XOR needs to be defined. I will define it the same way as was done in [9]. It is a function $f : \{0,1\}^2 \to \{0,1\}^N$ and $f(x_1, x_2, \ldots, x_n) = f(\vec{x}) = 1 \iff \sum_{i=1}^{N} x_i \mod 2 = 1$. In other words, N-dimensional XOR yields 1 as output for an odd number of ones in the input vector and thus it is sometimes called *bit parity problem*. The N-dimensional XOR is equivalent to applying left-associative binary operator XOR $(n-1)$ times: $f(\vec{x}) = x_1 \; XOR \; x_2 \; XOR \; x_3 \ldots \; XOR \; x_n$, or with a recursive definition: $f(x_1, x_2) = x_1 \; XOR \; x_2$ and $\forall n > 2 : f(x_1, \ldots x_n) = x_n \; XOR \; f(x_1, \ldots x_{n-1})$. Thus a recurrent neural network architecture is plausible, with the initial hidden state $h_0 = x_1$ and sequence of inputs $x_2, \ldots x_n$.

For the following baseline experiments I used a small initialization grid of all the combinations arising from each weight being initialized to either 0.01 or -0.01. Thus in each experiment, I ran the network 2^W times, where W is the number of weights (excluding the biases).

4.1.2 3-Dimensional XOR - Baselines

Starting with the vanilla MLPs baseline, the first experiments were carried out with a neural network with 4 hidden units in its single hidden layer (this was selected as a reasonable minimal architecture). This reasoning follows the intuition that one needs four linear separators to divide the two output classes for 3-dimensional XOR. The reasoning is very similar to the one with the 2-dimensional XOR, but now the

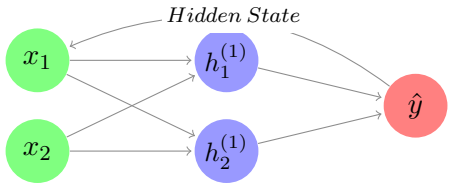


Figure 4.1: Recurrent neural network for N-dimensional XOR

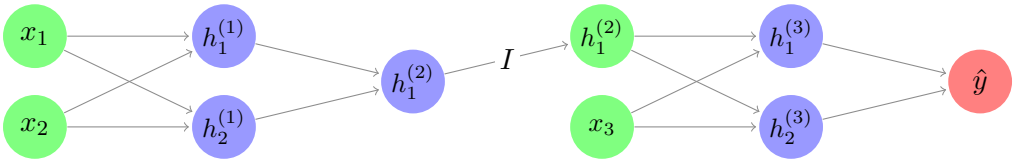


Figure 4.2: Unfolded recurrent network for 3-dimensional XOR

feature space is a cube alternating 0 and 1 labels on its vertices. This architecture, with 16 weights and 5 biases, was run with more than $65,000$ $(= 2^16)$ different weight initializations, yet no network converged to a perfect classifier.

The second baseline was a recurrent network (as described in Section 4.1.1; see Figures 4.1 and 4.2) with two recursive cycles, 14 weights and 6 biases. It was run on more than $16,000$ $(= 2^14)$ weight initializations and did not converged to perfect classification in any of them.

4.1.3 3-Dimensional XOR - Weight Sharing

I considered two ways to solve 3-dimensional XOR via weight sharing. One way (easier and more efficient) would be to take one of the weight sharing schemes from 2-dimensional XOR, learn it on 2-dimensional XOR, and only then to insert the learned weights to the recurrent architecture (without any need for further learning). This is also an approach used by Ha et al. in his work on hypernetworks [10]. The second way would be to reuse some of the weight sharing schemes from 2-dimensional XOR, but then learn the weights on the 3-dimensional data already.

The first approach worked perfectly. Take e.g. the weight sharing architecture from Figure 2.13 together with the weights to which this architecture converged in the 2-dimensional XOR experiments. When I inserted the weights into the recurrent architecture (with one recursive cycle), the network indeed computes 3-dimension XOR without any further learning. Similar results were observed with other successful weight sharing architectures from 2-dimensional XOR experiments.

The second approach, reusing just the sharing schemes was not successful, probably because of the combination of two factors. The first factor could be that the hidden state h_1 which is the output of the first recursive cycle (which should intuitively compute 2D XOR for the first two input features) is an output of a logistic sigmoid function, which means it is from an open interval $(0, 1)$. But then, it is used as an input for the second recursive cycle (which should intuitively compute 2D XOR for the first result of XOR and the third input feature) which should be 0 or 1 rather than a number from $(0, 1)$. The second factor is the restricted initialization grid which could impose some hidden symmetry-breaking problems.

4.1.4 3-Dimensional XOR - Conclusion

The baseline methods were unsurprisingly unsuccessful with 0% convergence. The successful approach turned out to be reusing the sharing schemes and learned weights from 2-dimensional XOR and inserting them into the recurrent architecture. This works very well and it is also a scalable approach that should work for higher dimensional XOR as well.

4.2 Integer Set Sum

Moving to the next example problem, I trained neural networks to output sum of an integer set of varying length. By considering sets as inputs, it is implicitly assumed that the learned function is invariant to input permutations. Another added level of complexity is the varying length of the input. I could have tried the same approach for this problem as with XOR, but it is tough to construct a weight sharing scheme for two reasons:

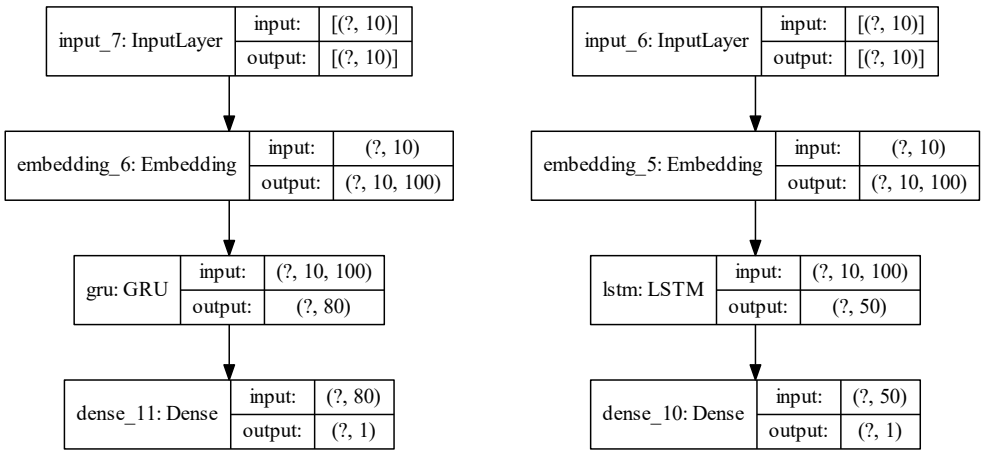
1. the proper schemes may vary together with the length of the input,
2. for high-dimensional input (even if the dimension is fixed), it is hard to find good schemes since the number of possible weight sharing schemes grows rapidly

So instead of weight sharing schemes, an architecture with input permutation symmetry built into it, working well on input of different, often high, dimensions was needed. Also, since we are restricted to a finite number of training data and thus we cannot train the model on all different input dimensions, it was necessary to choose an architecture that would learn on training input with dimension from some closed interval $[m, n]$, and the learned weights of the architecture would generalize (could be reused) on testing input of dimension $l, l \notin [m, n]$.

This description fits well with the Deep Sets architecture [28], already described in the introduction (see Section 1.2.3).

Similarly to the previous sections, I began with more traditional, symmetry-unaware approaches, before comparing them to the Deep Sets network. With a vanilla MLP, I made a case that learning the sum function even in the case of a fixed-sized input vector is not an easy task. Then, I reproduced the results achieved by Deep Sets' authors with recurrent architectures (for introduction to RNNs, see Section 4.1.1): gated recurrent units - GRU and long short-term memory network - LSTM. It became obvious that while these architectures can handle the varying input dimension very well, they lack behind in terms of learning the intrinsic symmetry of the problem. Finally, I reproduced the results of Deep Sets architecture introduced by Zaheer et al. [28] (see Figure 4.4).

Remark 4.3. There are generally two options when adjusting neural architectures to input of varying length. First option is to use some form of a recurrent network (see Definition 4.1) which processes the input sequentially. The other option is to convert the input to a fixed-sized embedding from which a prediction can be made regardless of the original input dimension. Where applicable, I will proceed with the Zaheer's choice, which was the second option for all three architectures they experimented with – GRU, LSTM, and Deep Sets.



(a) : GRU architecture **(b) :** LSTM architecture

Figure 4.3: Baseline architectures from the integer set sum experiments

4.2.1 Multi-Layer Network for a Fixed-Sized Vector

Let's start with the baseline of an MLP. It became clear that learning a sum of a set can be hard even when the length of the input is fixed. For this experiment, I fixed the length of the input to 10 and tried handful of different fully connected architectures with 2 or 3 layers. The narrower networks (with tens of units in each layer) gave poor results with average accuracy ranging from 0% to 10%. Only when the number of neurons in at least the first of the 3 layers was increased to hundreds, the network was capable to properly learn the sum of a set (with 100% accuracy). This showed that unless very wide networks were used, even the sum of a fixed-sized vector is a non-trivial learning problem for a conventional neural network. Thus, a specialized architecture was worth our attention.

4.2.2 LSTM and GRU Approach

Examples of specialized architectures are neural architectures with LSTM and GRU built into them (the concrete used architectures are in Figures 4.3b and 4.3a). Both architectures were trained for 50 epochs, with batch size of 128. Then, they were tested on unseen data with length of $5, 10, 15 \ldots$ all the way up to 95.

Both of these architectures performed well on short inputs (inputs of the same length as the training data - with maximum length of 10). However, their performance rapidly decreases with increasing length of the input, see Figure 4.5[1], where results of both architectures are compared with results of Deep Sets (they will be discussed in the next section).

[1]Image was generated by code from Github repository [27] accompanying the Deep Sets paper [28]

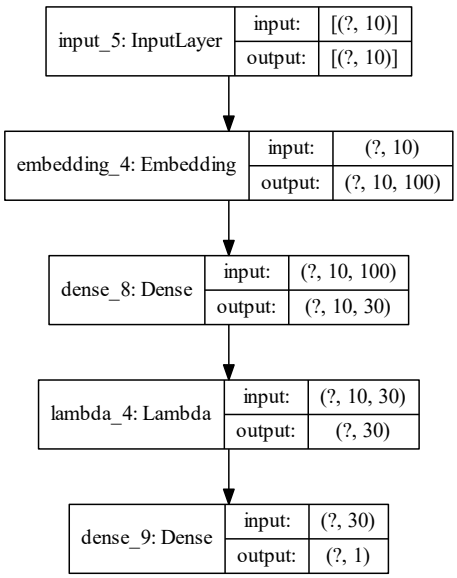


Figure 4.4: Deep Sets architecture from the integer set sum experiments

4.2.3 Deep Sets

Next, I replicated the learning process of the Deep Sets architecture. Deep Sets were trained only for 10 epochs, with batch size of 128. Again, they were tested on unseen data with a length of $5, 10, 15 \ldots$ all the way up to 95. The results of the Deep Sets architecture can be seen in the already mentioned Figure 4.5. The accuracy is as impressive, as the authors stated, gaining 100% for all lengths of the input vector, although the architecture was trained only on vectors with a maximum length of 10.

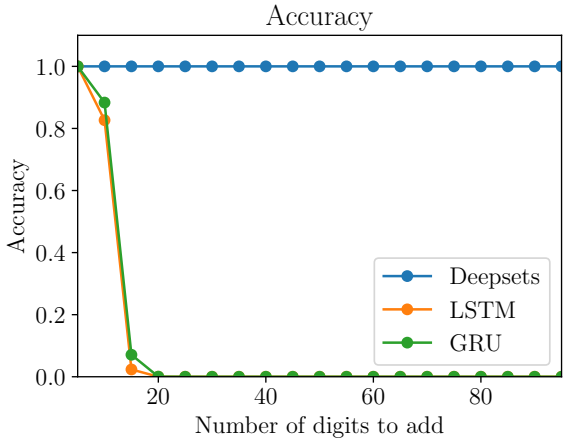


Figure 4.5: Accuracy of the used architectures [27] [28]

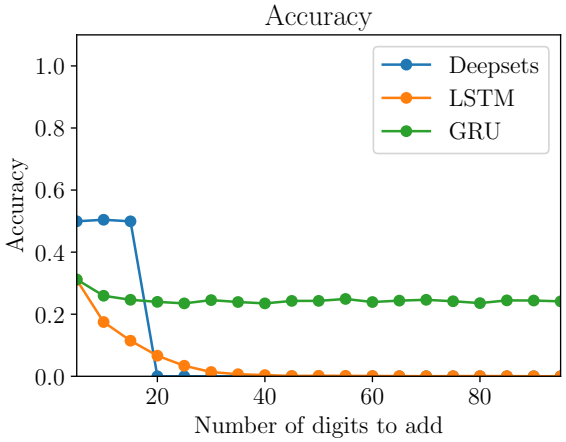


Figure 4.6: Accuracy of the used architectures on XOR Problem [27] [28]

4.2.4 XOR with Deep Sets, GRU, and LSTM

Lastly, I tried the suggested Deep Sets architecture (along with GRU and LSTM) on my previous example problem, XOR. I changed the dimension of the embedding from 10 to 2, since XOR, unlike set sum, has only two possible digits in its input, 0 a 1. The results were poor on all three architectures, see Figure 4.6[2] with accuracies depending on the length of the input. This was a disappointment from the Deep Sets architecture since XOR is not so much different from a set sum - N-dimensional XOR can be computed as the sum of the input *mod* 2. The fact that Deep Sets did not work well on the problem suggests that the addition layer (as described in the introduction, see Section 1.2.3), which is the signature part of the architecture, is working better for problems connected with addition, while for problems involving different operations (such as the modulo operation in XOR) it might not perform so well. This would, unfortunately, mean that Deep Sets are not as general of an architecture as they promised to be.

4.2.5 Deep Sets Conclusion

Deep Sets is an interesting symmetry-aware architecture building block. Yet, it is no magic wand. To use it successfully on a new problem, the practitioners have to come up with most of the architecture themselves as the example architectures given by the authors do not generalize well to other problems.

The symmetry-aware architecture suggested by authors was indeed working very well on the problem of set sum, and the accuracy of Deep Sets significantly surpassed the accuracies of some of the more traditional models. But trying a similar architecture on a related problem, N-dimensional XOR, was disappointing. Tuning a slightly modified architecture provided by the authors of Deep Sets for the problem of sum of integer sets was not successful, and, for higher input lengths, the other

[2]Image was generated by code from Github repository [27] accompanying the Deep Sets paper [28]

two architectures (LSTM and GRU) were actually more accurate.

4.3 Functions Invariant to Input Permutations - Conclusion

I presented two approaches towards implementing symmetry-aware neural networks for functions invariant to input permutations. With the first approach, weight sharing schemes, neural network successfully computed 3-dimensional XOR. Moreover, this approach is relatively easily scalable as was shown in Section 4.1.

The second approach was to use specialized architecture, Deep Sets, which promised to be generally applicable on problems defined on sets, thus also functions that are invariant to their input permutations. While the authors showed some impressive results, applying the architecture to new problems would probably require larger changes in the architecture (tuning hyperparameters did not help in my experiments) and thus using Deep Sets is no bulletproof approach, rather we should use it as another interesting building block when building our architectures.

Chapter 5

Functions Invariant to Graph Isomorphisms

Let me continue with the problem of graph isomorphism type of symmetry. As an example experiment, I trained graph neural networks to approximately differentiate, for a given base graph G, between graphs that are isomorphic to G and graphs that are nonisomorphic.

As was explained in Section 1.1.1, graph theorists need approximate algorithms to determine whether two graphs are isomorphic. Note that each graph convolutional layer performs a similar computation to a single iteration of a Color Refinement algorithm (compare Section 1.2.4 about graph convolutional layers and last paragraph of Section 1.1.1 about Weisfeiler-Leman test). Thus I tried to approximate this algorithm with the GNN. Further more, because the WL test cannot distinguish between some non-isomorphic graphs, when constructing the dataset, I included only graphs from isomorphic classes which were distinguishable by the WL test.

Concretely, I included the graphs in Figure 5.1 and their isomorphic equivalents constructed by renaming the original graph's nodes. From left to right, I will be using the names *triangle*, *square*, *pentagram*, *3-star*, and *4-star* in following paragraphs.

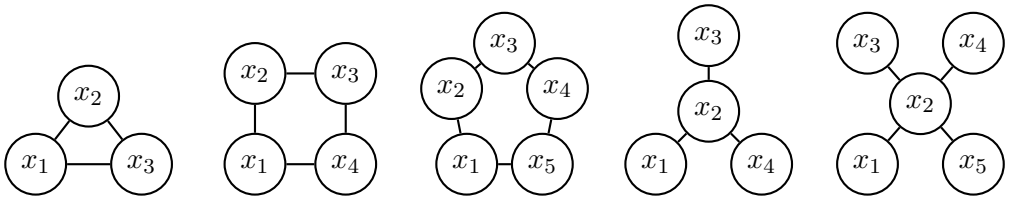


Figure 5.1: 5 different graphs used for the isomorphism problem

For simplicity, I approached this problem as binary graph classification, choosing 1 of the 5 graph classes above as a positive set of examples, and the remaining 4 classes as negative set of examples. As the positive class, I chose the *4-star* graph (and all its isomorphic variations) for reasons that are explained below.

Generating isomorphic graphs for a class of graphs means generating different renamings of the graph's nodes. After the renaming, the graph stays basically the same, but the computer representation (and subsequently also the visualization) changes, if the order of nodes is different.

This could affect the input of the network, the adjacency matrix, enormously, although the label is still the same. Thus, it is not a good idea to train a conventional

MLP for graph isomorphism problems, because (i) it is hard to find any correlation between the isomorphic graph matrices, and (ii) it inefficient to multiply sparse matrices, at least when using conventional ML libraries, and (iii) if the graphs vary in number of nodes, we cannot construct a vanilla MLP for them. *The last problem is why this is the only experiment in the thesis without a baseline of vanilla MLP - it is simply not possible to define such a network on inputs of different lengths.*

Lemma 5.1. *Given that the names of nodes in a graph with N nodes are $x_1, x_2, \ldots x_N$, there are $N!$ isomorphic graphs (including the original one) with different adjacency matrices.*

For our chosen classes of graphs, we have: $3! = 6$ isomorphic graphs for *triangle* graph, $4! = 24$ isomorphic graphs for *square* graph, $5! = 120$ isomorphic graphs for *pentagram* graph, $4! = 24$ isomorphic graphs for *3-star* graph, and $5! = 120$ isomorphic graphs for *4-star* graph.

The intuition behind choosing the *4-star* class of isomorphic graphs as the positive class was to have more balanced dataset with 120 positive and 174 negative examples. On the other hand, had we chosen e.g. *triangle* graph, we would have 6 positive and 288 negative examples.

5.1 Choosing the Graph Neural Network Architecture

When deciding on the architecture for approximating the Color Refinement algorithm, I needed to decide on two issues: (i) which graph convolutional layers to choose (already mentioned in Section 1.2.4), (ii) how many layers to stack before the pooling. Since graph convolutional layers perform a similar computation to an iteration of a WL test, the question of number of graph convolutional layers is deeply connected with the upper bound on iterations of WL test. As stated in Section 1.1.1, this upper bound is $n - 2$ for small graphs which we will talk about. Since the largest graph from our isomorphic classes has 5 nodes, I chose $5 - 2 = 3$ convolutional layers for the architecture.

5.2 Experiments with Graph Convolutional Layers

The experiments were carried out using the PyTorch Geometric framework [6] with the following architecture: three graph convolutional layers (each of them having a 32 channels/long node embedding), followed by a mean pooling to aggregate the embedded node features, and a final fully connected layer outputting a single number.

Compared to traditional usage of graph neural networks, the node features were not of an importance for classifying a graph as isomorphic or non-isomorphic to some other graph. Nevertheless, the graph convolutional layers require node features, otherwise they could not perform feature aggregation. Thus, I always assigned a single feature, 1, to all the nodes in a graph, since 1 is a neutral element with respect to multiplication operation that happens during the aggregation phase.

The last question remaining was whether to use the graph convolutional operator, as introduced in [15], or the graph isomorphism operator, as introduced in

[26]. Both intuitively and empirically, the second option (GI operator) proved to be more sensible. This is because the graph isomorphism operator (if the inner transformation is set to a single linear layer) is computing addition along the nodes during aggregation, while the graph convolutional operator is computing arithmetic mean along the nodes. Thus, with the graph convolutional operator, the layer would not be able to distinguish e.g. between a node with 3 neighbors with the same color and a node with 2 neighbors with the same color (average would be the same, while sum would be different).

Even though the graph convolutional layers can be sensitive to weight initializations, the results were good with the network converging to a perfect classifier usually between 9 to 20 epochs.

5.3 Functions Invariant to Graph Isomorphisms - Conclusion

Graph neural network is a powerful architecture that can, among many other uses, learn well-known graph theory algorithms, such as the Weisfeiler-Leman test. The conducted experiments could be further extended to learning any function that is invariant to graph isomorphism. In this case, the graph convolutional layers would serve as an operator transforming the whole class of isomorphic graphs to a single embedding.

Note that invariance to graph isomorphisms also includes other symmetries, e.g. invariance to graph rotations (for general discussion of rotation symmetry, see Section 1.1.1). This means that we could use graph neural networks for learning functions invariant to rotations (and even to input set permutations, if we consider a graph without edges), as long as we can construct a meaningful graph and divide the features, which are usually stored in a single consecutive array for conventional neural networks, between the graph nodes. This way, graph neural networks are generalization of all the previous architectures.

Chapter 6

Learning Rubik's Cube State Classification

Let me extrapolate the techniques from the previous chapters to a non-trivial problem involving non-obvious symmetries: the Rubik's Cube state classification. The choice of the problem is motivated by a paper published in Nature Machine Intelligence Journal by Agostinelli, McAleer, Shmakov, and Baldi from 2019 [1].

They applied their symmetry-unaware deep learning architecture called DeepCubeA to learning a 3x3x3 Rubik's Cube state evaluation, and this model was used during testing as a heuristic for a weighted A* search from random states. The technique was successful, as they were able to reach the goal state from 100% of the random test states, with the suggested path being the fastest solution in over 60% of the cases. Moreover, they claim that "[the architecture] generalizes to other combinatorial puzzles and is able to solve the 15 puzzle, 24 puzzle, 35 puzzle, 48 puzzle, Lights Out and Sokoban" [1].

After going through the previous experiments, an obvious question emerges, whether a symmetry-aware architecture would not achieve even better results.

6.1 Symmetries in Cube States

Given the fact that Agostinelli et al. used a fixed orientation of the cube, each state has $6 \times 4 - 1 = 23$ equivalent states that emerge only by rotating the whole cube, yet these states appear different to their chosen conventional deep neural architecture. Thus, their network had to spend considerable time for learning of the symmetry/equivalence of these states from the data.

6.2 Defining a Classification Problem for Rubik's Cube State Space

The aim of this thesis is to show the benefits of symmetry-aware neural architectures as compared to the conventional symmetry-unaware architectures. Therefore, I did not reproduce the whole experiment conducted by Agostinelli et al. and instead concentrated only on the aspect of learning symmetries. Thus, the following restrictions were made:

Firstly, instead of a 3x3x3 cube, I experimented with a 2x2x2 cube because the dimension of its state space is smaller. Thus, it is easier to show the difference

between symmetry-aware and symmetry-unaware architectures. Nevertheless, if an architecture is successful on the smaller cube, the chances are that the same architecture (but possibly wider in each layer) will generalize to the bigger cube, too. Secondly, instead of a reinforcement learning problem with value iteration and A* search (where the neural network is used only for the cube state evaluation), I restricted the problem to a binary classification problem that decides whether a cube state is a goal state or not. This problem could be later generalized to a multi-class classification problem or regression problem leading to the required state evaluation. This evaluation could then be simply used within the reinforcement learning loop to obtain the optimal policy.

For a set of 2x2x2 cube states $S(C_{2x2x2})$, a network transformation ϕ, its learnable parameters ρ, set of n training states $\{x_1, x_2, \dots x_n\} = X \subseteq S(C_{2x2x2})$, and a list of labels $y = [y_1, y_2, \dots y_n], y_i \in \{0, 1\}$ (1 for a goal state, 0 for a non-goal state), the optimization problem is defined as the minimization of the following cross-entropy loss:

$$\rho^* = \underset{\rho}{argmin} \sum_{i=1}^{n} y_i \cdot log(\phi_\rho(x_i)) + (1 - y_i) \cdot log(1 - \phi_\rho(x_i))$$

Finally, because there are only 24 goal states (which collapse to a single symmetric class with respect to the graph rotations) with the label 1, and it is desirable to have a balanced dataset, I chose another 24 non-goal states (which collapse to a different single symmetric class with respect to graph rotations) as the data with the label 0.

As a baseline, I experimented with a fully connected network with three layers, two with 24 neurons (and ReLU activations), and the last one with a single neuron followed by a sigmoid activation function, mapping the output to the $(0, 1)$ interval. As usual, the threshold for the classification function is set to 0.5. The label 1 is predicted *iff* $\phi(x) > 0.5$, otherwise label 0 is predicted.

For the symmetry-aware architecture, I chose two Generalized Graph Convolutional Layers, introduced by Li et al. [17], each having the node embedding of size 24, as well. The output was pooled across the nodes with a sum pooling layer, and the output of the pooling layer was run through the final fully connected layer with one neuron and a sigmoid activation function.

6.3 The Expressiveness of the Graph Representation

As for the graph representation, each 2x2 side of the cube was represented by 4 nodes with two types of edges between themselves (different edges for diagonal and vertical/horizontal connections between the facets) and another type of edge for the connections to nodes on the adjacent sides of the cube. This means 24 nodes and 60 edges for each graph in total. Each node had a feature vector of size 6, which was a one-hot vector describing its color. Differentiation of types of edges is achieved by specifying edge features which are then used in the aggregation phase of graph convolution as a multiplication factor. Representation of one such a side is then visualized in Figure 6.1.

An interested reader might ask why the differentiation between the edges is needed. It turns out that a graph model without edge features is not expressive enough. This

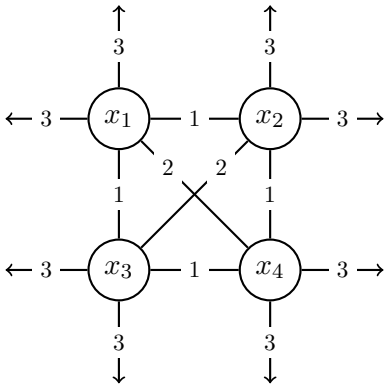


Figure 6.1: Rubik's cube side graph representation

was experimentally verified - the graph architectures without edge features were far from achieving the performance of their fully connected counterparts. Since the whole graph always contains a constant number of nodes with a particular color, and only their position changes, in the extreme case without the use of edge features, the pooling (which averages or sums the node embeddings) after the convolutional layers would map all the cube states to the same value because the aggregation of all nodes would always be the same. Thus it would not be able to differentiate between goal and non-goal states. Instead, by including different types of edge features, the relative position of the node colors is implicitly encoded.

6.4 Results of the Experiments

Firstly, I trained both architectures on the whole dataset of 48 samples. While a fully connected network did not have too many problems learning the classification function, converging to a perfect classifier in 25 to 30 epochs, I managed to get even a little bit better results with the graph convolutional network. Although graph neural networks are somewhat unstable to random initializations, the GNN was, in the end, able to converge in 17 epochs.

A more interesting experiment was to train these networks on half of the dataset (half of the positive and half of the negative examples) and measure the accuracy on the whole dataset. This gives a better idea about the generalization performance of the architecture. This time, while the GNN was able to classify the whole dataset correctly in 60 to 70 epochs, the fully connected counterpart could not converge to a classifier with 100% accuracy at all. Its accuracy ranged from 50% to 60%.

What happened is that the fully connected network overfitted on the training data. Then, seeing the same but rotated states, it could not extrapolate from the training data that they belong to one class or another because the input arrays were shuffled differently. On the other hand, the graph network had no problems converging to the optimum since the new data encountered in the testing phase produced the same embedding after the pooling layer. Thus, they were classified correctly.

6.5 Learning Rubik's Cube State Classification - Conclusion

Graph neural networks can help us solve complex problems, such as classifying Rubik's cube states, since they can deal with the graph isomorphism symmetry using global pooling on the output node embeddings. The rotation of the Rubik's cube is then a specific sub-instance of this type of symmetry. The GNN approach can improve upon the performance of fully connected networks in many cases.

But the usability of graph neural networks is surely not limited to the performed simplified experiment of goal state classification. Replicating the whole Deep Cube experiment [1] with a symmetry-aware GNN-based architecture is then an interesting possible direction for future work.

Chapter 7

Conclusion

While symmetries are present in many of the functions that machine learning practitioners try to approximate with neural networks, except for a few honorable mentions (such as the CNNs), they are not usually accounted for when developing a neural architecture. I have demonstrated that accounting for them by choosing some of the mentioned symmetry-aware architectures is often beneficial over a brute-force approach with deep fully connected networks in terms of computational efficiency and learning performance.

For logical functions invariant to reflection, weight sharing architectures achieved approximately 4× faster convergence on average than their symmetry-unaware counterparts, in terms of number of iterations needed for 100% accuracy. The disadvantage of this approach was that it was relatively easy to construct an ill-defined sharing scheme which was unable to approximate the function being learned.

Convolutional neural networks once again proved to be beneficial for learning problems involving symmetry in the form of pattern translations. While the example problem of pattern search in 1D binary array was relatively simple, and the symmetry-unaware, fully-connected networks were able to solve it eventually as well, CNN converged approximately 4× faster on average, if successful. The problematic side of CNNs is their instability to parameter initialization w.r.t. convergence, especially when using max-pooling after the last convolutional layer. As a result the CNNs got stuck in local minima even more often than the MLPs. A possible solution is thus running the initialization multiple times.

Simple generalization of weight sharing schemes from the chapter about reflection led to a successful recurrent architecture solving 3D XOR. Deep Sets architecture converged successfully and quickly on another problem invariant to input permutations – the integer set sum problem. In this case, the conventional approaches were not even able to achieve 100% accuracy. While performing well on this problem, the downside of Deep Sets is that generalization to other problems would require rebuilding the neural architecture significantly, as the presented version did not achieve good results on relatively similar problems, such as the XOR.

Graph convolutional neural networks worked well on both an example problem of classification of isomorphic and non-isomorphic graphs, and on a more practical application of the Rubik's cube state classification. In the former case, they were the only sensible choice, as the only input was the graph structure and the MLPs are not able to handle adjacency matrices of different sizes, as was the case with our

experiment. In the latter case, GNNs overperformed MLPs in terms of number of iterations needed for perfect classification, and, when presented with only a partial dataset, GNNs overperformed MLPs in accuracy as well, being able to generalize to unseen examples, unlike their fully connected counterparts.

7.1 Future of Symmetry-Aware Architectures

As the pool of low-hanging fruits of conventional deep learning successes is slowly drying up, the symmetry-first approach to developing neural architectures might become more and more important, and models such as graph neural networks could be commonly present in statistical model development.

The immediate possible future work involves applying graph neural architectures to problems, such as learning Rubik's cube policy, which have been, so far, solved by large fully connected architectures. Another possible line of work is to develop a more general framework around architectures like Deep Sets [28] or Deep Symmetry Networks [8].

Bibliography

[1] Forest Agostinelli et al. "Solving the Rubik's cube with deep reinforcement learning and search". In: *Nature Machine Intelligence* 1.8 (2019), pp. 356–363.

[2] Richard Bland. *Learning XOR: exploring the space of a classic problem*. Department of Computing Science and Mathematics, University of Stirling Stirling, 1998.

[3] Avrim L Blum and Ronald L Rivest. "Training a 3-node neural network is NP-complete". In: *Neural Networks* 5.1 (1992), pp. 117–127.

[4] François Chollet et al. *Keras*. `https://github.com/fchollet/keras`. 2015.

[5] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning". In: *arXiv preprint arXiv:1603.07285* (2016).

[6] Matthias Fey and Jan Eric Lenssen. "Fast graph representation learning with PyTorch Geometric". In: *arXiv preprint arXiv:1903.02428* (2019).

[7] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological Cybernetics* 36.4 (Apr. 1980), pp. 193–202. DOI: `10.1007/bf00344251`. URL: `https://doi.org/10.1007/bf00344251`.

[8] Robert Gens and Pedro M Domingos. "Deep symmetry networks". In: *Advances in neural information processing systems* 27 (2014), pp. 2537–2545.

[9] Luca Geretti and Antonio Abramo. "The synthesis of a stochastic artificial neural network application using a genetic algorithm approach". In: *Advances in imaging and electron physics* 168 (2011), pp. 1–63.

[10] David Ha, Andrew Dai, and Quoc V Le. "Hypernetworks". In: *arXiv preprint arXiv:1609.09106* (2016).

[11] Leonard GC Hamey. "XOR has no local minima: A case study in neural network error surface analysis". In: *Neural Networks* 11.4 (1998), pp. 669–681.

[12] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: `10.1038/s41586-020-2649-2`. URL: `https://doi.org/10.1038/s41586-020-2649-2`.

[13] Shell Xu Hu, Sergey Zagoruyko, and Nikos Komodakis. "Exploring weight symmetry in deep neural networks". In: *Computer Vision and Image Understanding* 187 (2019), p. 102786.

[14] Sandra Kiefer and Brendan D McKay. “The iteration number of colour refinement”. In: *arXiv preprint arXiv:2005.10182* (2020).

[15] Thomas N Kipf and Max Welling. “Semi-supervised classification with graph convolutional networks”. In: *arXiv preprint arXiv:1609.02907* (2016).

[16] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551.

[17] Guohao Li et al. “Deepergcn: All you need to train deeper gcns”. In: *arXiv preprint arXiv:2006.07739* (2020).

[18] PJG Lisboa and SJ Perantonis. “Complete solution of the local minima in the XOR problem”. In: *Network: Computation in Neural Systems* 2.1 (1991), pp. 119–124.

[19] Manish Mangal and Manu Pratap Singh. “Analysis of multidimensional xor classification problem with evolutionary feedforward neural networks”. In: *International Journal on Artificial Intelligence Tools* 16.01 (2007), pp. 111–120.

[20] Marvin Minsky and Seymour Papert. “Perceptrons: An introduction to computational geometry”. In: *Cambridge tiass., HIT* (1969).

[21] Graham Neubig et al. “DyNet: The Dynamic Neural Network Toolkit”. In: *arXiv preprint arXiv:1701.03980* (2017).

[22] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[23] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536.

[24] Ida G Sprinkhuizen-Kuyper and Egbert JW Boers. “The error surface of the simplest XOR network has only global minima”. In: *Neural Computation* 8.6 (1996), pp. 1301–1320.

[25] Boris Weisfeiler and Andrei Leman. “The reduction of a graph to canonical form and the algebra which appears therein”. In: *NTI, Series* 2.9 (1968), pp. 12–16.

[26] Keyulu Xu et al. “How powerful are graph neural networks?” In: *arXiv preprint arXiv:1810.00826* (2018).

[27] M. Zaheer. *DeepSets*. `https://github.com/manzilzaheer/DeepSets`. 2019.

[28] Manzil Zaheer et al. “Deep sets”. In: *arXiv preprint arXiv:1703.06114* (2017).

Appendix A

Comments on the Source Code, Libraries, and Cluster Usage

The code used for the experiments in this thesis can be found at `https://github.com/martin-krutsky/symmetries-in-deep-learning`. There are 7 directories that correspond to chapters of this thesis in the following way:

- Chapter 2 - 1_binlogic_baseline/, 2_binlogic_sharing/
- Chapter 3 - 5_cnns/
- Chapter 4 - 3_ndimlogic/, 4_deepsets/
- Chapter 5 - 6_gnns/
- Chapter 6 - 7_nontrivial/

The frameworks and libraries used for the experiments in this thesis are in the Table A.1.

Library	**Website**	**Usage**
NumPy [12]	`https://numpy.org`	experiments with logic functions
DyNet [21]	`https://dynet.readthedocs.io`	weight sharing for logic functions
PyTorch [22]	`https://pytorch.org`	experiments with CNNs
Keras [4]	`https://keras.io`	experiments with Deep Sets
PyTorch Geometric [6]	`https://pytorch-geometric.readthedocs.io`	experiments with GCNNs

Table A.1: Frameworks and libraries used for experiments

Some of the experiments were computationally demanding, and I would not be able to run them locally on my machine. Thus, the access to the computational infrastructure of the OP VVV funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics" is gratefully acknowledged.