



**F3**

**Faculty of Electrical Engineering  
Department of Computer Science**

**Master's Thesis**

# **Scaling Up Deep Relational Learning**

**Bc. Jan Neumann**

**janneumannprg@gmail.com**

**May 2024**

**Supervisor: Ing. Gustav Šír, Ph.D.**



## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Neumann** Jméno: **Jan** Osobní číslo: **483732**  
Fakulta/ústav: **Fakulta elektrotechnická**  
Zadávající katedra/ústav: **Katedra počítačů**  
Studijní program: **Otevřená informatika**  
Specializace: **Umělá inteligence**

## II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

**Škálování hlubokého relačního učení**

Název diplomové práce anglicky:

**Scaling up Deep Relational Learning**

Pokyny pro vypracování:

Deep relational learning [1] aims to generalize neural networks for learning from structured data, such as graphs, exploited with the recently popularized Graph Neural Networks (GNNs) [2], all the way to complex structures expressible in relational logic [3]. A salient feature of these approaches is that the sparse, irregular, and heterogeneous structure of the input data is also reflected in the structure(s) of the respective neural computation graph(s), which makes it very difficult to accelerate using standard parallelization of the common dense, regular, and homogeneous tensor operations exploited in classic deep learning. The aim of this thesis is to explore ways for efficient acceleration of this unorthodox computation regime, enabling to scale up to real-world structured datasets, such as those stored in relational databases.

- 1) Get acquainted with the principles of deep relational learning [1] and popular models such as GNNs [2].
- 2) Review existing deep (relational) learning frameworks [3] with focus on their respective acceleration practices [4].
- 3) Review the common principles of GPU acceleration, with focus on sparse, irregular, and heterogeneous computation.
- 4) Explore possibilities of the Graphcore's Intelligence Processing Unit (IPU) to address the respective limitations of GPUs [5].
- 5) Based on your research from 1-4, propose possible ways for accelerating the compute regime of deep relational models.
- 6) Design and implement suitable model acceleration strategies within the relational learning framework of NeuraLogic [3].
- 7) Demonstrate your advancements via deep learning directly from relational databases, beyond the currently explored single-table setup [6].

Seznam doporučené literatury:

- [1] Šír, Gustav. Deep Learning with Relational Logic Representations. IOS press, 2022.  
[2] Zhou, Jie, et al. "Graph neural networks: A review of methods and applications." AI Open 1 (2020): 57-81.  
[3] Sourek, Gustav, Filip Zelezny, and Ondrej Kuzelka. "Beyond Graph Neural Networks with Lifted Relational Neural Networks." arXiv preprint arXiv:2007.06286 (2020).  
[4] Fey, Matthias, and Jan Eric Lenssen. "Fast graph representation learning with PyTorch Geometric." arXiv preprint arXiv:1903.02428 (2019).  
[5] Graphcore IPU documentation: <https://www.graphcore.ai/developer>  
[6] Badaro, Gilbert, Mohammed Saeed, and Paolo Papotti. "Transformers for Tabular Data Representation: A survey of models and applications." Transactions of the Association for Computational Linguistics 11 (2023): 227-249.

Jméno a pracoviště vedoucí(ho) diplomové práce:

**Ing. Gustav Šír, Ph.D. Intelligent Data Analysis FEL**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **08.08.2023**

Termín odevzdání diplomové práce: **24.05.2024**

Platnost zadání diplomové práce: **16.02.2025**

\_\_\_\_\_  
Ing. Gustav Šír, Ph.D.  
podpis vedoucí(ho) práce

\_\_\_\_\_  
podpis vedoucí(ho) ústavu/katedry

\_\_\_\_\_  
prof. Mgr. Petr Páta, Ph.D.  
podpis děkana(ky)

### III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

\_\_\_\_\_  
Datum převzetí zadání

\_\_\_\_\_  
Podpis studenta

## Acknowledgement / Declaration

I offer my sincerest thanks to my mum, whom I am forever indebted to for her neverending and unconditional love and support, which she has always shown me in all of my endeavors. I am also beyond grateful to my supervisor, for tasking me with such an interesting field and topic to work on, and for his trust and support along the way.

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 also gratefully acknowledged.

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, May 24, 2024

## Abstrakt / Abstract

*Hluboké relační učení* zevšeobecňuje principy neuronových sítí na učení na relačních datech, čímž umožňuje využít přirozeně strukturální povahu takových dat (tvořenou např. cizími klíči v relačních databázích) jako součást samotného učení. Přestože takový přístup má teoretický nevyužitý potenciál přinést novou revoluci do strojového učení otevřením dveří k přímému využití jednoho z patrně nejpopulárnějších formátů pro ukládání dat na světě, podobné disciplíny hlubokého učení se dosud nestaly příliš populárními, patrně z důvodu obtížné škálovatelnosti trénování takových neuronových sítí. Navzdory tomu, grafové neuronové sítě (GNN), samy o sobě podmnožinou hlubokého relačního učení, se v posledních letech staly relativně populárními a úspěšnými pokusy o hluboké učení na strukturálních (grafových) datech. Jmenované výzvy z hlediska škálovatelnosti se jim podařilo do značné míry překonat, hlavně díky schopnosti efektivního využití grafických karet pro trénování, které byly a nadále zůstávají jedním z hlavních pilířů drtivé většiny úspěchů strojového učení, které v posledních letech vidáme. Tato diplomová práce představuje „kompilátor“ pro neuronové sítě hlubokého relačního učení, který úspěšně využívá principy efektivního učení grafových neuronových sítí pro celou jejich nadmnožinu hlubokých relačních sítí. Měření rychlostí běhů jednotlivých trénovacích procedur vychází pro představený kompilátor nejen na stejné úrovni jako existující řešení pro grafové neuronové sítě, ale mnohdy vychází dokonce lépe, navzdory tomu, že představený kompilátor je daleko všeobecněji použitelný.

*Deep relational learning* generalizes neural networks to relational data, allowing to utilize the inherent structural nature of such data (the foreign keys in relational databases) as part of the learning itself. While this has the theoretical untapped potential of revolutionizing deep learning yet again, allowing the utmost utilization of arguably one of the most widespread data storage representations for deep learning, similar deep learning fields have not gained significant traction yet, most likely due to challenges with respect to the scalability of training such neural networks. Nonetheless, *graph neural networks* (GNNs), themselves a subset of deep relational learning, have in recent years become an unexpectedly popular and successful endeavor into deep learning on top of structural (graph) data. Despite said challenges, GNNs are relatively successful in terms of scalability of their training, namely through successful utilization of GPUs, hardware that has been one of the crucial components to the many recent successes of deep learning as a whole. This thesis introduces a compiler for deep relational networks, which utilizes the principles forming the backbone of efficient GNN training for the full class of deep relational learning architectures, thereby opening the doors to efficient deep learning on top of relational data. Performance benchmarks show that in terms of performance of the neural network training procedure itself, networks trained with the use of the compiler not only match, but often exceed the performance of existing solutions for GNNs, despite the compiler being significantly more generally applicable.

# Contents /

<b>1 Introduction</b>	<b>1</b>		
1.1 Deep Learning Parallelization . . .	2		
1.2 Chapter Structure . . . . .	4		
<b>2 Literature Review</b>	<b>5</b>		
2.1 Deep Learning on Structured Inputs . . . . .	6		
2.1.1 Dynamic Computational Graphs . . . . .	6		
2.1.2 Graph Neural Networks . . . . .	7		
2.1.3 Graphcore IPUs . . . . .	7		
2.1.4 Deep Relational Learning . . . . .	8		
<b>3 Graph Convolution Computation</b>	<b>10</b>		
3.1 Terminology and Notation . . . . .	10		
3.1.1 Graph . . . . .	11		
3.1.2 Graph Convolution . . . . .	12		
3.2 Vectorized Graph Representation . . . . .	13		
3.3 Vectorized Image Convolution Computation . . . . .	14		
3.4 Vectorized Graph Convolution Computation . . . . .	16		
3.4.1 Gather/Scatter . . . . .	19		
3.4.2 Sparse Matrix Multiplication . . . . .	20		
3.4.3 Segment CSR . . . . .	23		
<b>4 NeuraLogic</b>	<b>25</b>		
4.1 Syntax . . . . .	25		
4.2 Example . . . . .	27		
4.2.1 More Complex Examples . . . . .	29		
4.3 Computational Graph Structure . . . . .	30		
4.3.1 Rules as Computational Graphs . . . . .	31		
4.3.2 Optimizations . . . . .	32		
<b>5 Implementation</b>	<b>33</b>		
5.1 Vectorization . . . . .	34		
5.1.1 Computational Graph Definition . . . . .	34		
5.1.2 The Input Computational Graph Operations . . . . .	35		
5.1.3 Node Group Vectorization . . . . .	36		
5.1.4 Vectorized Input Order Discrepancy . . . . .	37		
5.1.5 The Vectorized Computational Graph . . . . .	38		
5.1.6 Note About Aggregation Nodes . . . . .	40		
5.1.7 Generalization . . . . .	41		
5.1.8 The Batching Problem . . . . .	42		
5.1.9 Implementation Details . . . . .	44		
5.2 Optimizations . . . . .	45		
5.2.1 Basic Gather Optimizations . . . . .	46		
5.2.2 Basic Scatter Optimizations . . . . .	46		
5.2.3 Basic Linear Layer Optimizations . . . . .	47		
5.2.4 Advanced Linear Layer Optimizations: Reordering and Padding . . . . .	49		
5.2.5 Advanced Linear Layer Optimizations: Advanced Reordering/Padding . . . . .	50		
5.2.6 Deduplication: Downward Propagation of Gathers . . . . .	51		
5.2.7 Upward Propagation of Gathers . . . . .	52		
5.2.8 Unit Fact Processing . . . . .	54		
5.2.9 Further Optimizations for More Complex Architectures . . . . .	54		
5.2.10 The Full Implementation . . . . .	55		
<b>6 Results</b>	<b>58</b>		
6.1 Datasets . . . . .	58		
6.2 Graph Neural Networks . . . . .	58		
6.2.1 Performance . . . . .	58		
6.2.2 CPU Performance . . . . .	63		
6.2.3 Computational Graphs – GCN Example . . . . .	63		
6.3 Relational Architectures . . . . .	69		
6.4 The Backward Pass . . . . .	71		
6.5 Graphcore Intelligence Processing Units (IPUs) . . . . .	72		
<b>7 Conclusion</b>	<b>76</b>		
<b>A Complex Computational Graph Example</b>	<b>79</b>		
<b>B Code Guide</b>	<b>81</b>		
<b>C Additional Figures</b>	<b>82</b>		


<b>D Glossary</b>	<b>93</b>
<b>References</b>	<b>94</b>



## / Figures

<b>3.1</b>	Row-Major Ordering .....	11
<b>3.2</b>	Column-Major Ordering .....	11
<b>3.3</b>	Implicit Pixel Lattice Graphs in CNNs .....	14
<b>3.4</b>	Vectorized Pseudo-Algorithm For CNN Computation .....	16
<b>4.1</b>	Example NeuraLogic Com- putational Graph.....	28
<b>4.2</b>	Example NeuraLogic Edge Predicate Effect on Compu- tational Graphs .....	30
<b>5.1</b>	Vectorization Input Order Discrepancy Example – In- terleaving .....	37
<b>5.2</b>	Vectorization Input Order Discrepancy Example – Rep- etition.....	37
<b>5.3</b>	Vectorization Multi-Input Gather Example.....	39
<b>5.4</b>	Computational Graph Vec- torization .....	41
<b>5.5</b>	Computational Graph Vectorization Suboptimal Grouping Example .....	43
<b>5.6</b>	NeuraLogic Recursive Rules Vectorization Example .....	44
<b>5.7</b>	Basic Linear Layer Optimiza- tion .....	47
<b>5.8</b>	Basic Linear Layer Optimiza- tion: Dimension 2 Repeating ..	48
<b>5.9</b>	Basic Linear Layer Optimiza- tion: Padding.....	50
<b>5.10</b>	Upward Gather Propagation Optimization .....	53
<b>6.1</b>	GNN Forward Pass Perfor- mance Comparison on GPU ...	60
<b>6.2</b>	GNN Forward Pass Perfor- mance Comparison on GPU (Close-Up).....	61
<b>6.3</b>	GNN Forward Pass Perfor- mance Comparison on GPU (Best Result) .....	62
<b>6.4</b>	GNN Forward Pass Perfor- mance Comparison on CPU ...	64

<b>6.5</b>	GNN Forward Pass Performance Comparison on CPU (Close-Up).....	65
<b>6.6</b>	GNN Forward Pass Performance Comparison on CPU (Best Result) .....	66
<b>6.7</b>	Example PyTorch Geometric GCN Forward Pass.....	67
<b>6.8</b>	Example Unoptimized Vectorized NeuraLogic GCN Forward Pass .....	67
<b>6.9</b>	Example Optimized Vectorized NeuraLogic GCN Forward Pass.....	68
<b>6.10</b>	Example Optimized Vectorized NeuraLogic GCN Forward Pass, With Absolute Upward Gather Propagation ..	69
<b>6.11</b>	Example Relational Forward Pass Performance Comparison on GPU (Close-Up) .....	70
<b>6.12</b>	Performance Comparison Between Scatter and Segment CSR For Forward/Backward Pass .....	71
<b>6.13</b>	GNN Backward Pass Performance Comparison on GPU ...	73
<b>6.14</b>	GNN Combined Forward + Backward Pass Performance Comparison on GPU .....	74
<b>C.1</b>	GNN Backward Pass Performance Comparison on CPU ...	83
<b>C.2</b>	CNN Combined Forward + Backward Pass Performance Comparison on CPU .....	84
<b>C.3</b>	GNN Forward + Backward Pass Performance Comparison on GPU .....	85
<b>C.4</b>	GNN Forward + Backward Pass Performance Comparison on GPU (Close-Up) .....	86
<b>C.5</b>	GNN Backward Pass Performance Comparison on GPU ...	87
<b>C.6</b>	GNN Backward Pass Performance Comparison on GPU (Close-Up).....	88



<b>C.7</b>	GNN Forward + Backward Pass Performance Comparison on CPU .....	89
<b>C.8</b>	GNN Forward + Backward Pass Performance Comparison on CPU (Close-Up).....	90
<b>C.9</b>	GNN Backward Pass Performance Comparison on CPU ...	91
<b>C.10</b>	GNN Backward Pass Performance Comparison on CPU (Close-Up).....	92



# Chapter 1

## Introduction

The efforts towards making computers *learn* to perform various tasks from experience as opposed to giving them exact instructions have had meaningful initial success as early as three quarters of a century ago, depending on when we start counting. The vast majority of methods covered by the all-encompassing umbrella term “machine learning” is designed to operate on input in the form of fixed-size  $n$ -dimensional dense numeric arrays, often called *tensors*. No matter whether we are discussing methods such as decision trees [1], support vector machines [2], or whether we turn to the most widely discussed approach of the last decade – deep learning [3], which has allowed us to achieve impressive results in domains such as image recognition, speech recognition or natural language processing [4], the format of input data used remains relatively unchanged. The conversion of e.g. image data to the feature tensor format is straightforward. The same can be said with relative ease for ‘streaming’ data such as text (see e.g. [5]), video (a sequence of images), or audio.

Unlike all of the above, real-life data is often non-sequential, structural and irregular; in fact, arguably the most popular data storage format in enterprise applications is *relational database management systems* (RDBMS) [6], dominating other data storage systems in industries such as finance or medicine, as well as, e.g., in technologies used on the Web. The keyword *relational* refers to the basis of the design model of RDBMS, which is rooted in<sup>1</sup> *relational algebra* [7], a theory for modeling, structuring and querying data, proven to be equivalent in its expressive power to (*domain*) *relational calculus* [8], based on *first-order logic* (FOL).

One of the main reasons why deep learning has enjoyed tremendous success in recent years is that the training has successfully been massively parallelized and distributed [9–17], specifically on the *graphics processing unit* (GPU) [13–17]. However, problems arise for NN architectures operating on top of structural data, as they, in the general case, do not inherently possess the properties needed for effective parallelization.

Dynamic NN architectures have long been proposed and studied for various applications [18–20]; however, they have not gained meaningful traction, possibly due to said challenges in terms of scalability. An exception from recent years has been *graph neural networks* (GNNs), designed to operate on top of graph data representations,<sup>2</sup> based on extending the concept of a convolution (and of convolutional neural networks – CNNs [21]) to non-Euclidean domains (such as graphs) [22].

Since graphs, as a data representation, are a subset of relational data, the aim of this thesis is to study the principles guiding efficient, parallelized GNN training, and to extend said principles onto relational data, and, by extension, onto deep relational neural

<sup>1</sup> With relaxed requirements and additional features for the ease of use as a database, diverging a bit from its theoretical roots.

<sup>2</sup> In the case of GNNs, please make sure not to confuse the data on which they operate – *graphs* – with their *computational graphs*, i.e., the representation of the algorithm itself as the composition of simple operations. To make the distinction more apparent, the former are referred to as *input graphs*, and the latter as *computational graphs*.

networks. To demonstrate that this is indeed possible, a Compiler for dynamic structural computational graphs will be introduced, where the input computational graphs corresponding to the forward passes of arbitrary deep relational networks, granular to the degree of individual neurons (i.e., graphs of many operations on individual scalar values or small tensors, as such not trivially parallelizable, e.g., on a GPU), will be converted into vectorized equivalents of significantly lower granularity, consisting of a low number of highly parallelizable standard operations. Next, multiple optimizations as part of the Compiler will be introduced, as well as a degree of flexibility in the resulting computational graphs, allowing to reduce the overall runtime even further in exchange for increased memory usage whenever possible.

In terms of performance, it will be shown that the subset of deep relational learning that corresponds to GNNs yields not only equivalent, but often even better performance in training when utilizing the Compiler, compared to using existing frameworks for the training of GNNs (namely PyTorch Geometric [23]). What is more, it will be demonstrated that the Compiler successfully applies these concepts onto the full range of deep *relational* networks, operating on top of arbitrary *relational* data, such as data from relational databases. This allows the parallelized training of arbitrary deep relational networks.

This demonstration will be done on top of neural networks built using NeuraLogic [24–25], an open source framework for the construction, training and inference of deep *relational* learning networks, using the formalism of first-order logic, equivalent to relational data querying languages, for their construction. NeuraLogic trains its networks exclusively on the CPU, and as such does not trouble itself with the task of vectorizing their computational graphs. Together with the Compiler, the immense expressive power of deep relational networks, as offered by NeuraLogic, is enhanced with the capabilities of the parallelization of its training, including on specialized hardware, which until now was accessible only to the graph neural network subset of the possible networks.

## 1.1 Deep Learning Parallelization

The ability to parallelize the training (as well as the inference) via the use of hardware such as GPUs, has been crucial to the success that deep learning has had, as, e.g., training a neural network (NN) of a “mere” 60 million parameters on two consumer-grade GPUs can take 5 to 6 days [26], and the same effort performed sequentially, e.g., on consumer-grade CPU (*central processing unit*) hardware of the same retail price, would likely require orders of magnitude longer training time. The necessity of doing so on CPUs would thus have stumped any efforts of training large models. Thankfully, training deep learning models on GPUs has now long been widely accessible to the public via open source frameworks such as TensorFlow [27] and PyTorch [28], and the most demanding models of today, with billions or even trillions of learnable parameters, are trained by industry leaders on clusters of 1000s of GPUs [29], thus widening the performance gap achieved by massive parallelism even further.

The ability to parallelize existing deep learning models with ease is attributable to the fact that both the training and inference algorithms commonly rarely consist of much more than elementwise mathematical operations on large dense tensors (i.e., operations on large consecutive blocks of memory where individual values can be processed independently, and thus are parallelizable trivially), and matrix multiplication of large dense tensors, which is now a well-studied and scalable task in terms of parallelization/dis-

tribution as well [30]. Furthermore, the *computational graphs* of such models (i.e., the representation of the underlying computation – the algorithm – itself, in the form of a composition of simple operations) are *static*, i.e., the same for any input data, since the input data structure itself is regular and fixed. This lends an inherently trivial solution to input batching, which is a necessary precursor to effective parallelization specifically on the GPU.

Neural network architectures operating on top of structural data, on the other hand, do not inherently have these properties: Firstly, they have *dynamic* computational graphs, i.e., computational structures *unique* to the specific input. With respect to the GPU, this poses an additional challenge, where batching the input-unique dynamic computational graphs together (i.e., aligning operations that are common together across examples) is an NP-hard task in the general case [31]. Using a suboptimal solution to this problem impairs the degree to which a given neural network can be parallelized on the GPU, and in some cases even the optimal solution may not allow for full parallelism at all times. Secondly, even when successfully batched and parallelized, additional data shuffling/reordering operations may be needed due to the structural nature of the input data, adding extra computational overhead compared to deep learning on plain feature tensors, especially on the GPU.

GNNs have a trivial solution to the computational graph batching/alignment problem, as the graph convolution operations are typically defined as operations made on *arbitrary* input graphs using *static*, hand-crafted computational graphs of vectorized operations, which means that the task of batching is merely the task of modifying the input, not the computation. This is done trivially, as the batching of input graphs into a single input graph requires mere concatenation of tensors, or an equally simple operation, depending on the exact input graph representation. Furthermore, graph convolutions can typically be computed using operations with commonly available vectorized implementations, e.g., on the GPU. Aside from operations frequently used in deep learning on plain feature tensors, this requires either matrix multiplication operations on *sparse tensor representations*, or principally equivalent operations in the form of the so-called “gather”/“scatter” operations, discussed in detail in Chapter 3. This means that the resulting computational graphs are in fact static, even on the GPU.

Even so, specialized hardware, such as the Intelligence Processing Units (IPUs) by Graphcore, has been developed to achieve better parallelism when training models such as GNNs [32–34], by introducing a hybrid between the highly flexible nature of MIMD (multiple instruction, multiple data) architecture of CPUs, and the specialized SIMD (single instruction, multiple data) architecture of GPUs. The benefit of this is that the IPU allows for parallelization of computational graphs *without* their batching, even when the computation differs across input examples; something that is impossible to do on the GPU. This allows for parallelism in arbitrary NN architectures, and in the case of GNNs, potentially allows for a different approach to their computation entirely. Nonetheless, the IPU architecture itself has restrictions with respect to the dynamicity of its computational graphs, as they must be fixed *across* batches, resulting in the need for clever solutions to NP-hard batching problems of a different nature [32].

## 1.2 Chapter Structure

Chapter 2 goes into further detail about the vast array of existing machine learning methods for relational data, as well as the deep learning on structural data in general, GNNs, and deep relational learning.

Chapter 3 explains in detail the main principles underlying the efficient training of graph neural networks, which will be utilized in later chapters for equivalently efficient training of deep relational networks.

Chapter 4 offers an introduction to deep relational learning by introducing the NeuroLogic framework. The goal is to familiarize the reader with deep relational learning architectures, and how their design, as well as their computational graph, relate to relational data representations.

Chapter 5 introduces the main contribution of this thesis, i.e., the Compiler. In this chapter, it will be thoroughly explained how the Compiler transforms the computational graph into highly parallelizable equivalents, as well as the additional optimizations that it applies to the resulting computational graphs in order to further improve their performance.

Chapter 6 compares the performance of the resulting system on classical GNN architectures and commonly available datasets with PyTorch Geometric, and demonstrates how it operates on more complex architectures as well. Some example computational graphs are also shown in this chapter, so that the reader is equipped with some initial intuition into how some of the most important configuration parameters of the Compiler affect the computational graph transformations, as well as how they affect the resulting performance itself. Lastly, the Graphcore IPU is also discussed with respect to the Compiler in this chapter.

Chapter 7 serves as the conclusion and discussion of possible next steps.



## Chapter 2

### Literature Review

Learning on top of relational data representations has been the subject of study of a field known as *Inductive logic programming* (ILP), a term coined in the early 1990s as an intersection of inductive learning (i.e. machine learning) and logic programming [35]; logic programming here referring to the use of FOL formalism for expressing the input data (positive and negative examples), as well as some background knowledge in the form of FOL rules. The goal of ILP then was to obtain an algorithm for inferring hypotheses, i.e. the rules seemingly governing the positive/negative examples (ones that are not necessarily provable with full certainty given the background knowledge, but ones that cannot be proven incorrect either), to be able to classify and/or generate further examples [36]. Soon after, *Statistical relational learning* (SRL) introduced the modeling of uncertainty into the mix [37–38]. Unfortunately, approaches based on symbolic inference, such as SRL and ILP, typically do not scale well, especially for significantly large problems.

Despite the omnipresence of relational data in enterprise applications, as discussed in Chapter 1, the business go-to approach for training machine learning models on such data is *not* to use it in its original relational form. Instead, the industry standard is to preprocess such data into numeric feature vectors/tensors. This technique is known as *propositionalization* [39]. It is often used together with *feature engineering*, which propositionalization itself can be viewed as a form of. The resulting data is typically used with decision tree ensemble models such as XGBoost [40] to achieve state-of-the-art performance, as they appear to perform better than recent deep learning models for plain tabular data [41]; let alone relational data. Propositionalization-based solutions have had meaningful commercial success [42–44].

Unfortunately, the use of methods such as decision trees or SRL over deep learning strips the models of the *latent representation learning* capability inherent to deep learning, which the unmatched success of deep learning in such a wide array of complex tasks can easily be attributed to (among other reasons) [45]. What is more, propositionalization is by principle inferior to finding ways to keep the data in its original, fully relational form. This is because propositionalization is inevitably *lossy*, as in order to present the data in flattened form, it must discard the relational data structure, and as such simplify the data, potentially to a great degree. Technically speaking, it is possible to perform lossless flattening of relational data (known as a *universal relation*), but only to a limited depth, as the result may otherwise be infinite. Furthermore, doing so typically leads to an exponential explosion of the size and complexity of the data, as well as to a lot of repetition [46]. This is not only demanding in terms of memory usage, but also for the model to be able to utilize the relationships within the data, which are now no longer explicit, but rather implicit. Therefore, feature engineering is used in order to (to a limited degree) restore the information that has been lost as a result of flattening the data, while preventing the exponential blowup. However, feature engineering is by nature driven mainly by heuristics, which makes it close to manual representation engineering, i.e. the exact opposite of latent representation learning. Since propositional-

alization is the go-to method nonetheless, tabular data (and by extension relational data) has been labeled as “the last unconquered castle for deep learning” [47].

## 2.1 Deep Learning on Structured Inputs

The two most widely known and used deep learning frameworks/libraries for developing and training deep NNs are undeniably TensorFlow and PyTorch. Both the aforementioned frameworks allow the user to define neural network architectures by composing sets of pre-made, reusable array operations together into static computational graphs, done using programming interfaces exposed in the Python programming language. In terms of training/inference itself, TensorFlow compiles the network architecture down into an executable program beforehand, executing it on the given hardware after [27], whereas PyTorch executes code on the GPU asynchronously, interleaving GPU hardware operation execution with the eager execution of the Python code [28].<sup>1</sup>

Irrespective of this architectural difference of the two frameworks, however, neither is particularly concerned with any form of direct support for dynamic computational graphs, as the networks are meant to be designed static, via the explicit use of operation sequences based on array operations commonly available on standard hardware.

### 2.1.1 Dynamic Computational Graphs

For the specific purpose of training neural networks with dynamic computational graphs, there have been solutions developed in the past, such as TensorFlow Fold [50] or DyNet [51]. Both of these frameworks perform *dynamic batching*, i.e. they take the individual computational graphs corresponding to the individual inputs, and convert them into a single computational graph of a lower granularity, merging matching operations from different computational graphs (corresponding to different input examples) together. This batching algorithm is necessary because the computational graphs of individual examples no longer batch together trivially, since they can have arbitrary structures, and the batching is needed for parallelism on SIMD hardware such as GPUs.

Both TensorFlow Fold and DyNet use heuristic approaches to do this, albeit different algorithms are used in each respective framework. In the end, the resulting batched computational operations are the same as those commonly available in e.g. PyTorch or TensorFlow. For example, TensorFlow Fold facilitates the batched execution via the use of TensorFlow’s `gather` and `concat` operations [27].

Despite the low popularity of either of the two frameworks (compared to GNNs at the very least), efforts have continued to improve the on-demand batching algorithms, as well as to keep the extra overhead of using these algorithms during runtime sufficiently low, considering that the original algorithms are based on heuristics, and therefore suboptimal [52–55, 31]. The task of optimal batching of arbitrary (computational) graphs has been shown to be NP-hard [31], a property that should come as no surprise.

<sup>1</sup> Newer versions of PyTorch add optional support for just-in-time (JIT) compilation, as well as tracing compilation, operating closer to how TensorFlow operates, should the user prefer this [48]. Conversely, TensorFlow now offers eager execution, in the spirit of the original PyTorch proposal, as well [49].

## ■ 2.1.2 Graph Neural Networks

After the initial proposal of a *graph convolutional network* (GCN) [56], many GNN proposals similar in spirit, defining more complex graph convolution operators, have soon appeared [57–62], all being available for use in e.g. the PyTorch Geometric library [23].

A very important property of GNN convolutions is that they commonly allow for being represented directly as *static* computational graphs operating on top of arbitrary graph input represented as a pair  $(V, E)$ , where  $V$  is an  $n + 1$ -dimensional tensor of node values of shape  $(|V|, m_1, \dots, m_{|V|})$ , and  $E$  is either an edge index tensor, or an adjacency matrix of the whole graph.<sup>2</sup> Training e.g. a GCN does not require a dynamic computational graph per se, but allows for a fixed computational structure (irrespective of the exact values contained in the input tensors). Technically speaking, this can be done using only the set of operations commonly used in deep learning on feature tensors, namely (dense) matrix multiplication. However, in practice, the adjacency matrix of the input graph is commonly a *sparse* matrix, and as such requires a specialized representation, in order to prevent inefficient memory use and inefficient computation. A custom matrix multiplication algorithm is also needed for this. Common sparse matrix representations are discussed in Chapter 3, together with an equivalent alternative solution in the case of the *edge index* input graph representation. Both such solutions are commonly used e.g. by PyTorch Geometric, which supports either graph representation.

This means that complex approaches for dynamic batching are often not needed for GNNs, as building a batch of inputs is simply equivalent to concatenating the individual  $V$  tensors together, and either concatenating individual  $E$  edge index tensors together as well (after the  $V$  tensor concatenation has been reflected in the individual indices in  $E$ ), or, in the case of adjacency matrices being used in place of  $E$ , building a *block-diagonal* matrix from them [23]. In either case, the operation has linear (i.e. negligible) time complexity, and is done ahead-of-time, prior to training.

Ultimately, it is possible that GNNs jumped ahead in terms of popularity over other dynamic NN architectures precisely as a result of the simplicity with which they can be built/trained/evaluated on commonly available hardware, using paradigms not too different, nor significantly more complex, to those commonly already used for NNs for non-structural data.

## ■ 2.1.3 Graphcore IPUs

The Graphcore IPU, as already introduced in Chapter 1, offers a specialized hardware architecture for training/inference of atypical neural network architectures. The design of the IPU allows parallel computation of computational graphs without the need for the alignment of their individual components, i.e. without dynamic batching as done in [50–55, 31].

The computational paradigm of the IPU involves the scheduling of small computational tasks onto a large array of independent processing units called *tiles*, interleaved with centralized task scheduling and memory exchange operations [63].

The programming paradigm for the IPU requires defining the full computational graph AOT, which must then be compiled AOT into the actual instruction sets for the individual hardware components; only then can the program be executed. This computational graph must be *static*, but any operations within said graph that are truly independent

<sup>2</sup> In the case of the input examples being individual small graphs, a full batch of examples is then a graph consisting of the individual example graphs as its connected components.

can be executed in parallel, such that as many tiles as possibly are utilized at any given moment. The task of determining and scheduling operations based on this onto the individual tiles is done as part of the AOT compilation. The advantage of this is that operations can run in parallel irrespective of what they are (i.e. the IPU can perform multiple *different* operations in parallel, unlike a GPU); the only requirement is that they are independent (i.e. that the output of one is not needed as the input of the other), but unlike on the GPU, the computational instructions can differ.

This computational graph can then be reused multiple times with different numeric inputs.

This means that while the IPU does not require clever ways of batching the *instructions within a single batch*, it benefits from (or needs, depending on the task) clever ways of batching the *data across batches*. Since the computational graph must remain fixed for a given program, then when multiple runs (for multiple batches of data) are performed, it must be ensured that the computational graph can remain the same for each batch.

For neural network architectures with dynamic computational graphs, this often translates into NP-hard problems as well, albeit different ones than in the case of GPU execution. For example, in the case of GNNs trained on the IPU, this means that padding becomes needed for the  $V$  input tensor. This is because the  $V$  input tensor must have the exact same shape across batches on the IPU, including the batch dimension, meaning that additional padding must be used in the tensor for batches where the total no. of nodes is lower than the maximum one. Therefore, the work in [32] for the IPU introduces an advanced batching method *across batches*, finding the optimal pairwise disjoint split of the input examples into individual batches, such that the padding needed is minimal, i.e. such that the total numbers of nodes are roughly the same for each batch. Note that such padding is not determined by the number of examples (i.e. graphs), but by the number of *nodes*, which makes the problem of “*batch packing*” (as named by the paper) an NP-hard problem as well [32].

Conversely, a GNN written e.g. in PyTorch Geometric, being executed on the GPU, does not require the same batch size across different batches, as the size of the batch dimension of the  $V$  tensor is not important, even when the sequence of GPU operations is kept the same across batches.

## ■ 2.1.4 Deep Relational Learning

NeuraLogic [24–25] is one of the very few examples of deep learning being studied specifically for relational data in connection with the broader structural deep learning approaches discussed above. While many methods do find a link between deep relational learning and GNNs [64–67], NeuraLogic comes from the opposite end of attempting to bridge the gap between deep learning and the original work of ILP and SRL.

Specifically, NeuraLogic uses the FOL formalism for defining both the input data and the NN architectures. However, for those uninitiated about FOL, NeuraLogic also supports working directly with datasets stored in RDBMS [68]. Direct use of RDBMS as input for deep learning has also recently been explored by the same authors in [66], utilizing both GNNs and the Transformer architecture [69] in PyTorch, instead of NeuraLogic, for training directly on data with relational structures. This work also provided a set of helper tools for directly accessing RDBMS, and a publicly available directory of relational datasets for benchmarking with other methods. A parallel, duplicate effort, specifically for deep learning on relational data, appeared mere months later in [67].

Nonetheless, NeuraLogic itself remains interesting outside of the research trend of approaching deep relational learning as an extension of GNNs. This is because NeuraLogic allows designing not only typical GNN architectures using FOL, but also an entire class of more complex architectures, which closely reflect the (often arbitrary) structure of the input data, as designing such novel NN architectures becomes second nature using the NeuraLogic FOL formalism.

As a result of this, unfortunately, the problem of dynamic computational graphs does not escape NeuraLogic. NeuraLogic, however, has thus far not been concerning itself with solving this problem, as the authors found the CPU computation, performed directly on the computational graph built AOT, to be significantly faster than other frameworks, including PyTorch Geometric, for the unbatched case [70–71]. However, this should not come as a surprise considering that batching is often utilised precisely as the main approach towards achieving parallelism in NN training. On the other hand, it must be said in favor of NeuraLogic that batching has altogether been shown to be unfavorable in terms of the generalization capabilities of the resulting NN models [72–73], and as such, the CPU approach of NeuraLogic may in fact be the better choice.

Nevertheless, since NeuraLogic is able to, among other things, construct architectures equivalent to commonly known GNNs [74], the resulting computational graph should in principle be convertible to one similar to that returned e.g. by PyTorch Geometric for the same NN, as such allowing for parallel computation, including for the full batch case. In other words, in this special case at the very least, the dynamic graph batching problem should have a simple solution, ideally applicable to other architecture instances as well, allowing for simple batching of at least a subclass of NeuraLogic NNs.

# Chapter 3

## Graph Convolution Computation

The computational graphs produced by the Compiler share key characteristics with how GNNs are typically computed on the GPU, in both their batched and unbatched form. The convolution operation will be explained first, followed by an explanation of how e.g. PyTorch Geometric performs the computation.

### 3.1 Terminology and Notation

Before we proceed, we shall define some terminology and notation used later in the text:

- A *tensor* is a  $D$ -dimensional array of scalar numbers, e.g.  $\mathbf{T} \in \mathbb{R}^{m_1 \times m_2 \times \dots \times m_D}$ .
- The *shape* of tensor  $\mathbf{T} \in \mathbb{R}^{m_1 \times m_2 \times \dots \times m_D}$  is a sequence  $\mathbf{S} = (m_1, m_2, \dots, m_D) \in \mathbb{R}^D$  denoting the individual dimensions of the tensor.
- A 1-dimensional tensor is also known as a *vector*.
- A 2-dimensional tensor is also known as a *matrix*.

To obtain a value from a tensor, we will be using the notation  $\mathbf{T}(i, j)$  instead of the more commonly used  $\mathbf{T}_{i,j}$  due to readability. Indexing starts at 1. For example, given  $\mathbf{T} \in \mathbb{R}^{m \times n}$  as defined in Equation (1), it holds for  $i \in \{1, \dots, m\}$  and  $j \in \{1, \dots, n\}$  that  $\mathbf{T}_{i,j} = \mathbf{T}(i, j)$ . This notation can be extended intuitively to any  $D$ -dimensional tensor.

$$\mathbf{T} = \begin{bmatrix} \mathbf{T}_{1,1} & \mathbf{T}_{1,2} & \cdots & \mathbf{T}_{1,n} \\ \mathbf{T}_{2,1} & \mathbf{T}_{2,2} & \cdots & \mathbf{T}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{T}_{m,1} & \mathbf{T}_{m,2} & \cdots & \mathbf{T}_{m,n} \end{bmatrix} \quad (1)$$

For brevity, to denote the domain of real tensors of shapes  $\mathbf{S} = (m_1, m_2, \dots, m_D)$ , we may write  $\mathbb{T}^{\mathbf{S}}$  instead of  $\mathbb{R}^{m_1 \times m_2 \times \dots \times m_D}$ .

Furthermore, we may also use the above indexing notation to obtain tensors of lower dimensionality from other tensors. For example, for a  $D + 1$ -dimensional tensor  $\mathbf{R}$ ,  $\mathbf{R}(1)$  is a  $D$ -dimensional tensor,  $\mathbf{R}(1, 1)$  is a  $(D - 1)$ -dimensional tensor, etc. To index along dimensions other than the leftmost, we will be using “-” to indicate the skipped dimensions. For example,  $\mathbf{R}(-, 1)$  is also a  $D$ -dimensional tensor. Furthermore, for any index  $i$ ,  $\mathbf{R}(i) = \mathbf{R}(i, -) = \mathbf{R}(i, -, -) = \mathbf{R}(i, -, \dots, -)$ .

Specifically for the matrix  $\mathbf{T}$  from Eq. (1), e.g.  $\mathbf{T}(2)$  is the second *row* of  $\mathbf{T}$ , i.e.  $\mathbf{T}(2) = [\mathbf{T}_{2,1}, \mathbf{T}_{2,2}, \dots, \mathbf{T}_{2,n}]$ , and  $\mathbf{T}(-, 2)$  is the second *column* of  $\mathbf{T}$ , i.e.  $\mathbf{T}(-, 2) = [\mathbf{T}_{1,2}, \mathbf{T}_{2,2}, \dots, \mathbf{T}_{m,2}]$ .

Let us also define the notation for tensor *slicing*. We will extend the notation of indexing. For a given tensor  $\mathbf{R} = [\mathbf{R}(1), \dots, \mathbf{R}(m)]$ , a slice of said tensor along the first

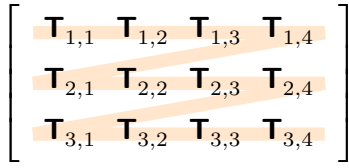


Figure 3.1. Row-major ordering

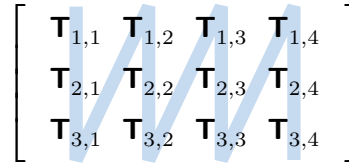


Figure 3.2. Column-major ordering

dimension from  $i$  to  $j$  ( $i < j$ ) is  $\mathbf{R}(i \rightarrow j) = [\mathbf{R}(i), \mathbf{R}(i+1), \dots, \mathbf{R}(j)]$ . This can be applied to other dimensions as well (e.g.  $\mathbf{R}(-, i \rightarrow j)$ ), or to multiple dimensions.

Let us also define the notion of *reshaping* tensors. Since any tensor is essentially merely a consecutive sequence of scalar values, any tensor  $\mathbf{R} \in \mathbb{R}^{m_1 \times \dots \times m_D}$  can simply be thought of as a vector  $\mathbf{R} \in \mathbb{R}^m$ , where  $m = \prod_{i \in \{1, \dots, D\}} m_i$ . This is known as *flattening* a tensor. Moreover, you may also choose to flatten a tensor only partially, i.e. for select dimensions only (keeping the remaining dimensions as-is), or the inverse operation of un-flattening can be done, or a sequence of a partial flattening followed by a partial un-flattening of a different subset of dimensions can be done as well. All such operations are together given the umbrella term *reshaping*. For any computer representation of a tensor, the underlying memory representation is typically a consecutive sequence of scalars irrespective of the exact reshaping of the tensor, which means that it typically does not change as a result of reshaping, only indexing of the tensor is affected. In other words, reshaping is computationally nearly free; a non-operation.

However, we must also explain how the consecutive memory is accessed in order to resolve ambiguity in how reshaping is understood with respect to indexing. This is best understood on the simple example of a 2-dimensional tensor  $\mathbf{T} \in \mathbb{R}^{m \times n}$  of shape  $(m, n)$ : Given an index  $(i, j) \in \{1, \dots, m\} \times \{1, \dots, n\}$ , in the underlying consecutive array/vector, the value corresponding to  $(i, j)$  can either be found at position  $(i-1) * n + j$ , or at position  $(j-1) * m + i$ . The former is known as *row-major ordering*, and the latter as *column-major ordering*. This is illustrated in Figures 3.1 and 3.2, respectively. This idea extends to  $D$ -dimensional tensors inductively.

In line with how PyTorch, TensorFlow, and NumPy [75–76] libraries operate with matrices and tensors, we will be using *row-major* ordering, depicted in Fig. 3.1. To provide an additional example, this means that if a  $D$ -dimensional tensor  $\mathbf{R}$  of shape  $(a, b, c_3, \dots, c_D)$  is reshaped e.g. to a  $(D-1)$ -dimensional  $\mathbf{R}'$  of shape  $(a * b, c_3, \dots, c_D)$ , then the sub-tensor  $\mathbf{R}(i, j)$  is now found at  $\mathbf{R}'(i * b + j)$ , and this sub-tensor is, given the reshaping, exactly the same, with the shape  $(c_3, \dots, c_D)$ .

Lastly,  $+$  and  $\odot$  will be used to denote elementwise addition and multiplication on tensors, respectively. The latter is circled in order to not be confused with linear matrix multiplication.

### 3.1.1 Graph

Let us begin with a simple definition of a graph, i.e. the data structure that the graph convolution operates on. Before we concern ourselves with how the computation is performed on hardware, let us use a definition of a graph that abstracts any computational details away from us for the sake of simplicity.

A graph is a triple  $(V, E, \mathcal{T})$ , where the following applies:

- $V$  is a set of nodes with some arbitrary identifiers, e.g.  $V = \{1, 2, \dots, |V|\} \subset \mathbb{N}$ .
- $E$  is a set of (directed) edges,  $E \subseteq V \times V$ .

- There exists a directed edge from node  $i \in V$  to node  $j \in V$  if and only if  $(i, j) \in E$ .
- $\mathcal{T}: V \mapsto \mathbb{R}^{m_0 \times m_1 \times \dots \times m_D}$  is a mapping of the nodes to their corresponding  $D$ -dimensional tensors, all of which have the shape  $\mathbf{S} = (m_0, m_1, \dots, m_D)$ .

Let us also denote  $\mathbb{G}^{\mathbf{S}}$  as the domain of all graphs with tensors of shape  $\mathbf{S}$ .

Furthermore, let us denote  $\mathbb{T}_V^{\mathbf{S}}$  as the domain of all possible mappings  $\mathcal{T}: V \mapsto \mathbb{R}^{\mathbf{S}}$ , and  $\mathbb{T}^{\mathbf{S}}$  as the union of all  $\mathbb{T}_V^{\mathbf{S}}$  over all possible  $V$ .

For brevity, given the shape  $\mathbf{S} = (m_0, m_1, \dots, m_D)$ , we will also write  $\mathbb{R}^{m_0 \times m_1 \times \dots \times m_D}$  as  $\mathbb{R}^{\mathbf{S}}$ .

### ■ 3.1.2 Graph Convolution

Mathematically speaking, a graph convolution  $\mathcal{O}: \mathbb{G}^{\mathbf{S}} \times \mathbb{N} \mapsto \mathbb{R}^{\mathbf{S}'}$  is a mapping that takes a graph  $G = (V, E, \mathcal{T})$  and a node  $i \in V$  on input, and produces a new value  $\mathcal{O}(G, i) = \mathcal{T}'(i) \in \mathbb{R}^{\mathbf{S}'}$  for the node, taking into account not only the original value of the node  $\mathcal{T}(i)$ , but also other information contained in the graph  $G$ , including the values  $\mathcal{T}(j)$  of other nodes  $j \in V$ , as well as the graph structure given by  $E$ .

A better intuitive way to view this is that the graph convolution in fact produces a whole new mapping  $\mathcal{T}'$  for the input  $G = (V, E, \mathcal{T})$ , i.e. new value tensors for all nodes in the graph. A perhaps even better intuitive view is that the graph convolution produces a graph  $G' = (V, E, \mathcal{T}')$ , with the same structure as the original graph, i.e. the same sets of nodes and edges, but with new value tensors. The original definition allows the simplest way of formally defining concrete convolutions, though.

For readability, let us also represent the mappings  $\mathcal{T}$  and  $\mathcal{T}'$  using sequences  $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{|V|})$  and  $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_{|V|})$ , where for a node  $i \in V$ ,  $\mathbf{x}_i = \mathcal{T}(i)$  and  $\mathbf{y}_i = \mathcal{T}'(i)$ .

So far, we have been defining what is a generic tensor operation on a graph, not necessarily a convolution. A convolution typically has the general form shown in Equation (2) for all  $i \in V$ , given the following:

- $\mathcal{N}(i)$  – the neighbors of node  $i \in V$ , i.e.  $j \in \mathcal{N}(i)$  if and only if  $(i, j) \in E$
- $\bigoplus$  – any differentiable, permutation-invariant function, e.g. sum
- $\phi, \gamma$  – any differentiable functions (e.g. multi-layer perceptrons)

$$\mathbf{y}_i := \gamma \left( \mathbf{x}_i, \bigoplus_{j \in \mathcal{N}(i)} \phi(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{i,j}) \right) \quad (2)$$

The  $\mathbf{e}_{i,j}$  term is only used in some convolutions, and allows e.g. assigning tensor values to individual edges and involving them in the computation, thus extending the expressiveness of the graph data representation. This is something that we will not need for the purposes of explaining relevant computational mechanisms, and we will offer similar features differently via the relational representation. Therefore, from now on, we will omit the term from the general graph convolution definition; it has only been included here for the sake of completeness.

A typical example of a convolution would be that of the GCN convolution [56], defined in Equation (3) (please note the enforced self-loops via the added summation term with  $j = i$ ).



$$\mathbf{y}_i := \sum_{j \in \mathcal{N}(i) \cup \{i\}} \left( \frac{1}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}} \mathbf{W}^T \mathbf{x}_j \right) + \mathbf{b} \quad (3)$$

The matrix  $\mathbf{W}$  and the vector  $\mathbf{b}$  denote the learnable parameters. The  $\deg(i)$  function is the degree (i.e. the total number of neighbors) of node  $i$  in the graph, i.e.  $\deg(i) := |\mathcal{N}(i)|$ . The scaling factor  $\frac{1}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}}$  will later be referred to as normalization.

The GCN convolution fits into the general graph convolution scheme by defining the individual operations  $\phi$ ,  $\oplus$ ,  $\gamma$  respectively as shown in Equation (4).

$$\begin{aligned} \phi(\mathbf{x}_i, \mathbf{x}_j) &:= \left( \frac{1}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}} \mathbf{W}^T \mathbf{x}_j \right) + \mathbf{b} \\ \oplus &:= \sum \\ \gamma(\mathbf{z}) &:= \mathbf{z} \end{aligned} \quad (4)$$

This assumes that  $\forall i: i \in \mathcal{N}(i)$ . Should that not be the case, self-loops must be added to the graph prior to execution. Alternatively, we may define  $\gamma(\mathbf{x}_i, \mathbf{z}) := \phi(\mathbf{x}_i, \mathbf{x}_i) + \mathbf{z}$ , but then we must *remove* all existing self-loops from the graph, otherwise the  $\phi(\mathbf{x}_i, \mathbf{x}_i)$  term would be added twice for nodes with self-loops. The normalization factor (i.e. the  $\deg(i)$  function) should be updated accordingly as well.

## 3.2 Vectorized Graph Representation

Instead of the original representation of a graph  $(V, E, \mathcal{T}) \in \mathbb{G}^{\mathbf{S}}$ , we may use a representation  $(\mathbf{V}, \mathbf{E})$ , where  $\mathbf{V}$  and  $\mathbf{E}$  are both tensors. Let us assume that the original set  $V$  has  $|V|$  total nodes, and  $\mathcal{T}$  has shape  $\mathbf{S} = (m_0, \dots, m_D)$ , meaning that for every  $i \in V$ , each value tensor  $\mathcal{T}(i)$  is  $D$ -dimensional and has shape  $\mathbf{S}$ . Without loss of generality, let us also assume that  $V = \{1, 2, \dots, |V|\}$ .

We may represent  $V$  directly as a  $D+1$ -dimensional tensor  $\mathbf{V}$  consisting of the individual values  $\mathcal{T}(i)$  for each  $i \in V$ . Given that  $V = \{1, 2, \dots, |V|\}$ , the value for  $i \in V$  can be found at the  $i$ -th position along the first dimension of tensor  $\mathbf{V}$ , i.e.  $\mathbf{V}(i) = \mathbf{x}_i = \mathcal{T}(i)$ .

There are multiple ways to represent the set of edges  $E$ . Firstly, we may use the *adjacency matrix* representation, which is a 2-dimensional tensor (i.e. a matrix)  $\mathbf{E} \in \{0, 1\}^{|V| \times |V|}$ , where  $(i, j) \in E$  if and only if  $\mathbf{E}(i, j) = 1$ , otherwise  $\mathbf{E}(i, j) = 0$ .

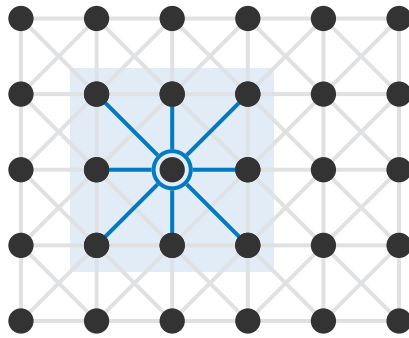
Alternatively, we may represent  $E$  using a 2-dimensional *edge index* tensor (matrix)  $\mathbf{E} \in \mathbb{N}^{2 \times |E|}$ , which essentially corresponds to a sequence of the values in  $E$ . Therefore,  $(i, j) \in E$  if and only if there exists a  $k \in \{1, \dots, |E|\}$  such that  $\mathbf{E}(-, k) = (i, j)$ , i.e.  $\mathbf{E}(1, k) = i$  and  $\mathbf{E}(2, k) = j$ .

The edge index representation of  $E$  imposes an ordering onto edges based on their positions  $k$  in the  $\mathbf{E}$  edge index tensor. This ordering can be arbitrary.

The general definition of a graph convolution, shown earlier in Eq. (2), can be redefined using  $\mathbf{V}$  as shown in Equation (5).<sup>1</sup>

$$\mathbf{V}'(i) := \gamma \left( \mathbf{V}(i), \bigoplus_{j \in \mathcal{N}(i)} \phi(\mathbf{V}(i), \mathbf{V}(j)) \right) \quad (5)$$

<sup>1</sup> The neighborhood  $\mathcal{N}$  is still used instead of  $\mathbf{E}$  because the two different representations of  $\mathbf{E}$  would require two different definitions, and we do not need to bother with this just yet.



**Figure 3.3.** Implicit pixel lattice graphs in convolutional neural networks (node neighborhood highlighted in blue)

### 3.3 Vectorized Image Convolution Computation

Before we discuss how the vectorized computation of a graph convolution is performed in the case of a general graph, we shall first discuss the special case of an image convolution, i.e. the backbone of convolutional neural networks (CNNs), operating on top of *images* (i.e. plain tensors) instead of *graphs*. This is important because it will help us understand why computing graph convolutions is computationally a more complex task than computing more classical, non-structural deep learning architectures, such as CNNs, and what the differences are.

A CNN is typically not viewed as having anything to do with graphs, and an image tensor is typically not understood as a graph, but we may view an image convolution as a special case of a graph convolution, where the graph is a two-dimensional lattice (grid), with diagonal edges and self-loops, where nodes correspond to pixels. This means that the neighborhood of a pixel consists of the pixel itself and its neighboring pixels. This results in a 3 by 3 pixel neighborhood, which is illustrated in Figure 3.3. Often, a 5 by 5 pixel neighborhood is used instead, which is only a slightly more complex lattice, with edges  $E'$  defined from the 3 by 3 lattice  $E$  such that if  $(i, j) \in E$  and  $(j, k) \in E$ , then  $(i, j), (i, k) \in E'$ . Let  $n$  be the width/height of the pixel neighborhood (e.g. 3 or 5).

Only the image tensor is needed on input, not the actual graph, as the graph structure is fixed. In other words, the graph is essentially only implicit, and the convolution operates directly on images.

The convolution, as used in CNNs, fits our definition of a graph convolution, as the output of the convolution is a graph of the same structure, with different node values, i.e. an image of the same dimensions, with different pixel values.<sup>2</sup>

For a given pixel  $\mathbf{y}_i$ , a typical image convolution is computed as written in Eq. (6), where  $\mathbf{W}$  is the learnable weights in the form of a tensor (of  $|\mathcal{N}(i) \cup \{i\}|$  total weights).

$$\mathbf{v}'(i) := \sum_{j \in \mathcal{N}(i) \cup \{i\}} \mathbf{w}(j) \mathbf{v}(j) \quad (6)$$

While image convolution is typically not defined as in Eq. (6), we can afford to do this because our general graph definition allows us to use the  $\mathcal{N}$  symbol. Commonly, however, the image isn't represented as a tensor of shape  $(|V|, \dots)$ , but rather in a

<sup>2</sup> You may notice that in actual CNNs, edge pixels are typically handled differently; either excluded, or padded. This is ignored for now, for the sake of simplicity.

reshaped form that maintains the image structure, i.e. with shape  $(h, w, \dots)$ , where  $h$  is the image height (i.e. number of rows of pixels),  $w$  is the image width (i.e. number of columns of pixels), and  $|V| = w * h$ .  $W$  is also typically in matrix form rather than in vector form, with shape  $(n, n)$ . Given  $\mathbf{I}$  being the reshaped image tensor,  $\mathbf{W}'$  being the reshaped parameter matrix, and  $\mathbf{I}_{\mathcal{N}(r,c)} := \mathbf{I}((r - \lfloor n/2 \rfloor) \rightarrow (r + \lfloor n/2 \rfloor), (c - \lfloor n/2 \rfloor) \rightarrow (c + \lfloor n/2 \rfloor))$  being the neighborhood slice of  $\mathbf{I}$  induced by the pixel at position  $(r, c)$  (as shown in Fig. 3.3), the convolution is typically defined as in Equation (7).

$$\mathbf{I}'(i, j) := \sum_{x \in \{1, \dots, n\}, y \in \{1, \dots, n\}} \mathbf{W}'(x, y) \cdot \mathbf{I}_{\mathcal{N}(i,j)}(x, y) \quad (7)$$

Please note that for this to work, we need to ensure that the neighborhoods  $\mathcal{N}(i, j)$  always form matrices of shape  $(n, n)$ . However, as you probably realize, this is not the case for edge pixels. This can be solved e.g. by adding extra padding to the edges of the tensor in the width and height dimensions, or by skipping the edge pixels entirely, using only those pixels, the neighborhoods of which form  $(n, n)$  matrices. We will be doing the latter for simplicity, even though the output image tensor will have shape  $(h-n+1, w-n+1, \dots)$  instead of  $(h, w, \dots)$ . Therefore, we can rewrite the convolution as in Equation (8).

$$\begin{aligned} \forall i \in \{1, \dots, h-n+1\}, j \in \{1, \dots, w-n+1\}: \\ \mathbf{I}'(i, j) &= \sum_{x \in \{1, \dots, n\}, y \in \{1, \dots, n\}} \mathbf{W}'(x, y) \cdot \mathbf{I}_{\mathcal{N}(i+\lfloor n/2 \rfloor, j+\lfloor n/2 \rfloor)}(x, y) = \\ &= \sum_{x \in \{1, \dots, n\}, y \in \{1, \dots, n\}} \mathbf{W}'(x, y) \cdot \mathbf{I}(i+x-1, j+y-1) \end{aligned} \quad (8)$$

This allows us to rewrite the convolution further, using a formula for the whole output tensor  $\mathbf{I}'$ , as opposed to a formula for individual  $\mathbf{I}'(i, j)$ . This is shown in Equation (9).

$$\mathbf{I}' = \sum_{x \in \{1, \dots, n\}, y \in \{1, \dots, n\}} \mathbf{W}'(x, y) \cdot \mathbf{I}(x \rightarrow (h-n+x), y \rightarrow (w-n+y)) \quad (9)$$

If it is not immediately clear why we can do this, perhaps Equation (10) will provide intuition, where we merely replace the  $\mathbf{I}'(i, j)$  value assignment from Eq. (8) with a sum over all  $\mathbf{I}'(i, j)$  values, and then simplify the resulting formula.

$$\begin{aligned} &\sum_{i \in \{1, \dots, h-n+1\}, j \in \{1, \dots, w-n+1\}} \sum_{x \in \{1, \dots, n\}, y \in \{1, \dots, n\}} \mathbf{W}'(x, y) \mathbf{I}(i+x-1, j+y-1) = \\ &= \sum_{x \in \{1, \dots, n\}, y \in \{1, \dots, n\}} \sum_{i \in \{1, \dots, h-n+1\}, j \in \{1, \dots, w-n+1\}} \mathbf{W}'(x, y) \mathbf{I}(i+x-1, j+y-1) \\ &= \sum_{x \in \{1, \dots, n\}, y \in \{1, \dots, n\}} \mathbf{W}'(x, y) \sum_{i \in \{1, \dots, h-n+1\}, j \in \{1, \dots, w-n+1\}} \mathbf{I}(i+x-1, j+y-1) \\ &= \sum_{x \in \{1, \dots, n\}, y \in \{1, \dots, n\}} \mathbf{W}'(x, y) \sum_{i \in \{x, \dots, h-n+x\}, j \in \{y, \dots, w-n+y\}} \mathbf{I}(i, j) \end{aligned} \quad (10)$$

Please note the  $i, j$  re-indexing in the inner sum in the final rewrite. The inner sum is essentially a sum over all values in a slice of the tensor  $\mathbf{I}(x \rightarrow (h-n+x), y \rightarrow (w-n+y))$ . Equivalently, a similar re-indexing has been done in Eq. (9).

```

I' := new tensor of zeros, of shape  $(h - (n - 1), w - (n - 1), \dots)$ 
for  $x \in \{1, \dots, n\}$ ,  $y \in \{1, \dots, n\}$  do:
   $(x_{\text{start}}, x_{\text{end}}) := (x, h - n + x)$ 
   $(y_{\text{start}}, y_{\text{end}}) := (y, w - n + y)$ 
  Islice := I $(x_{\text{start}} \rightarrow x_{\text{end}}, y_{\text{start}} \rightarrow y_{\text{end}}, \dots)$ 
  Imult := W' $(x, y) \odot$  Islice
  I' := I' + Imult
end for

```

**Algorithm 3.4.** Vectorized pseudo-algorithm for the computation of an image convolution

Equation (9) is close to how the image convolution operation is typically computed efficiently, allowing to utilize high parallelization. We can trivially rewrite it into an algorithm, shown in Figure 3.4.

This algorithm is highly parallel because the inner loop need not be performed sequentially, but can be performed simply by slicing the input image tensor, followed by vectorized (elementwise) multiplication and addition. All of these operations, including slicing, are typically implemented on hardware, including GPUs. The outer loop can be parallelized as well, but this will not be discussed.

While both TensorFlow and PyTorch implement image convolution directly in low-level programming languages close to hardware, Algorithm 3.4 can be used in either framework to reimplement it directly in Python via the composition of the three aforementioned simpler operations. The backward pass will be handled by the two frameworks automatically.

To understand the image convolution in greater detail, including the backward pass, see [3; Section 9.5].

The important conclusion for us is that to compute a vectorized convolution on a graph with nice properties, such as the lattice graph we discussed, only *tensor slicing*, *elementwise multiplication*, and *elementwise addition* operations are needed.

### 3.4 Vectorized Graph Convolution Computation

Now that we have a tensor representation of a graph formally defined, and we understand the parallel computation of an image convolution, we can now proceed onto the explanation of how the parallel computation of a graph convolution is implemented e.g. in PyTorch Geometric.

To compute even the simplest possible convolution operation, such as the GCN convolution (Eq. (3), p. 13) excluding the normalization factor, we need to be able to perform operations over node neighborhoods for *arbitrary* graphs.

To demonstrate one way of solving this, given a graph  $G$ , let us say that we want to retrieve a tensor  $\mathbf{n}$  such that its  $i$ -th position  $\mathbf{n}(i)$  contains the list of values  $\mathbf{V}(j)$  for all neighbors  $j \in \mathcal{N}(i)$ . Since each node in a graph can have a different number of

neighbors, we will have to pad the tensor. The resulting shape of tensor  $\mathbf{n}$  will thus be  $(|V|, \Delta(G), \dots)$ , where  $\Delta(G) := \max\{\deg(i) \mid i \in V\}$ .

We will need to generalize this concept onto any tensor, so let us do it as follows: Given a graph  $G = (V, E)$  and an arbitrary tensor  $\mathbf{T} \in \mathbb{R}^{|V| \times \dots}$  (please note that  $\mathbf{T}$  is required to have  $|V|$  values in its first dimension), let us define an operation  $\mathcal{V}_{\mathbf{E}}: \mathbb{R}^{|V| \times \dots} \mapsto \mathbb{R}^{|V| \times \Delta(G) \times \dots}$ , such that  $\mathcal{V}_{\mathbf{E}}(\mathbf{T})$  is a tensor of shape  $(|V|, \Delta(G), \dots)$ , built by gathering the values  $\mathbf{T}(j)$  for  $j \in \mathcal{N}(i)$ , much like in the case of the tensor  $\mathbf{n}$ , but generalized onto any value tensor  $\mathbf{T}$ , not just the original  $\mathbf{V}$ .

Therefore, assuming an operation  $\mathcal{V}_{\mathbf{E}}$  is available to us, then for a given  $i \in V$ , to compute e.g.  $\mathbf{y}_i := \sum_{j \in \mathcal{N}(i)} \mathbf{W}^T \mathbf{x}_j + \mathbf{b}$  will become possible as shown in Equation (11), assuming padding is done with zeros.

$$\begin{aligned} \mathbf{v}' &= \left[ \sum_{k \in (0, \dots, \Delta(G))} \mathbf{W}^T \cdot (\mathcal{V}_{\mathcal{N}}(\mathbf{V})(i, k)) + \mathbf{b} \mid i \in |V| \right] \\ &= \left[ \sum_{k \in (0, \dots, \Delta(G))} \mathcal{V}_{\mathbf{E}}(\mathbf{W}^T \cdot \mathbf{V}(i))(-, k) + \mathbf{b} \mid i \in |V| \right] \\ &= \sum_{k \in (0, \dots, \Delta(G))} \mathcal{V}_{\mathbf{E}}([\mathbf{W}^T \cdot \mathbf{V}(i) \mid i \in |V|])(-, k) + \mathbf{b} \end{aligned} \quad (11)$$

Having such an operation would thus work very well, as we could perform the computation by simply doing the following:

1. For each  $\mathbf{V}(i)$ , perform matrix multiplication  $\mathbf{W}^T \cdot \mathbf{V}(i)$ . Since the same operation (matrix multiplication with  $\mathbf{W}^T$ ) is being performed for each  $i$ , this is easy to parallelize on SIMD hardware such as the GPU. This is typically known as *broadcasted* matrix multiplication.
2. Apply the  $\mathcal{V}_{\mathbf{E}}$  operation onto the result.
3. Sum the result along its second dimension (i.e. along the neighborhoods), i.e. to obtain a tensor of shape  $(|V|, c_2, \dots, c_D)$  from an input tensor of shape  $(|V|, \Delta(G), c_2, \dots, c_D)$ .
4. Add  $\mathbf{b}$ .

Of course, the remainder of our discussion will revolve around how to efficiently perform  $\mathcal{V}_{\mathbf{E}}(\mathbf{T})$ , or how to efficiently compute graph convolutions similarly, as all of the other operations that we used here, are commonly used in deep learning, whereas  $\mathcal{V}_{\mathbf{E}}$  is exclusive to deep learning on structure inputs, such as to GNNs.

Please note that from this point onwards, any matrix multiplication will be generalized to use the broadcasted variant if needed. We will denote broadcasted matrix multiplication using  $\circ$ . Since this is inexact, we will only be using it sparingly, when it is clear how the broadcasting is performed, merely to avoid unnecessary verbosity.

This allows us to rewrite Eq. (11) as Equation (12).

$$\mathbf{v}' = \sum_{k \in (0, \dots, \Delta(G))} \mathcal{V}_{\mathbf{E}}(\mathbf{W}^T \circ \mathbf{V})(-, k) + \mathbf{b} \quad (12)$$

If  $\mathbf{E}$  is an adjacency matrix, we do not need to have the exact equivalent of the  $\mathcal{V}_{\mathbf{E}}$  operation. By doing  $\mathbf{E} \cdot (\mathbf{W}^T \circ \mathbf{V}) + \mathbf{b}$ , we obtain the full result of the computation of

the GCN convolution. Matrix multiplication with  $\mathbf{E}$  directly sums the results over the neighborhoods, so we have performed steps 2 and 3 in a single step, also being able to avoid padding.

However, for the adjacency matrix, it often holds that  $\left(\sum_{i,j \in V} \mathbf{E}(i,j)\right) \ll |V|^2$ , which means that only a small number of elements is non-zero<sup>3</sup>, i.e. that the matrix is *sparse*. Therefore, the common representation of the matrix as a consecutive array of values, known as a *dense* representation, is often too memory intensive, and performing matrix multiplication with such a matrix in dense representation would also be computationally quite expensive, performing many unnecessary multiplication operations with zeros. Therefore, sparse matrix representations and sparse matrix multiplication algorithms are typically used instead of the common, dense ones. They are discussed in Section 3.4.2.

Alternatively, to avoid the padding that  $\mathcal{V}_{\mathbf{E}}$  requires, we may instead choose to define a similar operation  $\mathcal{V}'_{\mathbf{E}}$  that also outputs neighborhoods of individual nodes in order, but does so all in the first dimension, i.e. resulting in a tensor with shape  $(\sum_{i \in V} \text{deg}(i), \dots) = (|E|, \dots)$ , so that padding is not needed.

What  $\mathcal{V}'_{\mathbf{E}}$  does is that it *gathers* individual values from the first dimension of  $\mathbf{V}$  in the order of indices  $[j \mid \forall i \in V, j \in \mathcal{N}(i)]$ .

When using  $\mathcal{V}'_{\mathbf{E}}$  instead of  $\mathcal{V}_{\mathbf{E}}$ , aggregation can no longer be done along a single dimension, which the padding allowed us to do, but must be done using a more clever method, as the boundaries between individual neighborhoods in the first dimension are no longer clear. Therefore, we need an operation that is, loosely speaking, inverse to  $\mathcal{V}'_{\mathbf{E}}$ , such that it is able to produce a tensor of shape  $(|V|, \dots)$  from input of shape  $(|E|, \dots)$ , based on node neighborhood groupings. Since  $|V| \leq |E|$ , this inverse operation must be able to perform aggregation as well. We will thus denote it as  $\mathcal{V}'_{\mathbf{E}, \oplus^{-1}}$ , to suggest that it uses the  $\oplus$  operation to aggregate values from individual groups.

We can rewrite the GCN convolution specifically using  $\mathcal{V}'_{\mathbf{E}}$  and  $\mathcal{V}'_{\mathbf{E}, \Sigma^{-1}}$  as shown in Equation (13).

$$\mathbf{V}' = \mathcal{V}'_{\mathbf{E}, \Sigma^{-1}} \left( \mathcal{V}'_{\mathbf{E}} (\mathbf{W}^T \circ \mathbf{V}) \right) + \mathbf{b} \quad (13)$$

This also indicates that in the case of  $\oplus = \Sigma$  and the adjacency matrix  $\mathbf{E}$  representation, it holds that  $\mathcal{V}'_{\mathbf{E}, \Sigma^{-1}} (\mathcal{V}'_{\mathbf{E}} (\mathbf{T})) = \mathbf{E} \cdot \mathbf{T}$ .

Equation (14) generalizes the use of  $\mathcal{V}_{\mathbf{E}}$  to an arbitrary graph convolution, as defined in Eq. (5) (p. 13).

$$\mathbf{V}'(i) = \gamma \left( \mathbf{V}(i), \bigoplus_{j \in \{1, \dots, \Delta(G)\}} \phi(\mathbf{V}(i), \mathcal{V}_{\mathbf{E}}(\mathbf{V})(i, j)) \right) \quad (14)$$

Unfortunately, we cannot use  $\mathcal{V}_{\mathbf{E}}$  directly to simplify the graph convolution in the general case. We also cannot use  $\mathcal{V}'_{\mathbf{E}}$  and  $\mathcal{V}'_{\mathbf{E}, \oplus^{-1}}$  as they are defined. Instead, we will need operations known as “gather” and “scatter.”  $\mathcal{V}'_{\mathbf{E}}$  is a special case of the gather operation, and  $\mathcal{V}'_{\mathbf{E}, \oplus^{-1}}$  is a special case of the scatter operation. Gather and scatter operations themselves are special cases of sparse matrix multiplication. The two complementary operations will be defined and discussed in greater detail in Section 3.4.1.

<sup>3</sup> Please remember that the values of an adjacency matrix are in the  $\{0, 1\}$  domain.

### 3.4.1 Gather/Scatter

The *gather*<sup>4</sup> operation  $\mathcal{G}_I$  on a tensor  $\mathbf{T}$  of shape  $(m, \dots)$  is an operation that produces a reordering of  $\mathbf{T}$  of shape  $(n, \dots)$  along its first dimension based on a mapping of indices  $\mathcal{M}: \{1, \dots, m\} \mapsto \{1, \dots, n\}$ . The mapping itself is defined using a sequence of indices  $I \in \{1, \dots, m\}^n$  ( $|I| = n$ ), where  $\mathcal{M}(i) = j$  if and only if  $I(j) = i$ . This means that *the inverse mapping to  $\mathcal{M}$  is injective*.

The result of the gather operation can be written as shown in Equation (15).

$$\mathcal{G}_I(\mathbf{T}) := [\mathbf{T}(i) \mid i \in I] \quad (15)$$

The *scatter* operation  $\mathcal{S}_J^\oplus$  on a tensor  $\mathbf{T}$  of shape  $(n, \dots)$  is an operation that produces a reordering of  $\mathbf{T}$  of shape  $(m, \dots)$  along its first dimension (with possible aggregations of multiple rows) based on a mapping of indices  $\mathcal{M}^{-1}: \{1, \dots, n\} \mapsto \{1, \dots, m\}$ . The mapping itself is defined using a sequence of indices  $J \in \{1, \dots, m\}^n$  ( $|J| = n$ ) where  $\mathcal{M}^{-1}(j) = i$  if and only if  $J(j) = i$ . This means that *the mapping  $\mathcal{M}^{-1}$  is injective*.

The result of the scatter operation can be written as shown in Equation (16).

$$\mathcal{S}_J^\oplus(\mathbf{T}) := \left[ \bigoplus_{\substack{j \in \{1, \dots, n\}, \\ J(j)=i}} \mathbf{T}(j) \mid i \in \{1, \dots, m\} \right] \quad (16)$$

The above shows that the operations are complementary in the sense that a *gather* operation allows us to explicitly define the *input* value index  $i$  for each *output* index  $j$ , whereas a *scatter* operation has us define the *output* value index  $j$  for each *input* index  $i$ . This means that the operations also have complementary limitations: a gather operation is restricted to using each *output* index exactly once (i.e. placing exactly one input value to each output position), whereas a scatter operation is restricted to placing each *input* value exactly once. Since the scatter operation may (unlike the gather operation) place multiple input values onto the same output index, a reduction operation must be defined alongside the mapping, to merge said input values into one (most commonly used reduction methods are elementwise sum, mean, minimum or maximum of the input values). Conversely, the gather operation is unrestricted in being able to duplicate an input value onto multiple output positions, which the scatter operation cannot do.

Gather and scatter operations can be seen as special cases of sparse matrix multiplication, as will be shown below. Nonetheless, specialized algorithmic implementations are usually provided for gather and scatter (outside from the generic sparse matrix multiplication algorithms), taking advantage of the specifics of the two operations.

The gather operation on a matrix  $\mathbf{M} \in \mathbb{R}^{m \times o}$  is an operation equivalent to  $\mathbf{A} \cdot \mathbf{M}$ , where  $\mathbf{A} \in \mathbb{R}^{n \times m}$  is a sparse matrix of values from the two-value domain of  $\{0, 1\}$ , and where the sum of each row is equal to 1, i.e., for each row  $i$ , there is exactly one column  $j$  where  $\mathbf{A}(i, j) = 1$ , and the remaining columns in said row contain zeros. Note that the matrix  $\mathbf{A}$  can have an arbitrary number of rows. The gather operation on a higher-dimensional tensor  $\mathbf{T}$  of shape  $(m, \dots)$  can be defined as  $\mathbf{A} \circ \mathbf{T}$ .

<sup>4</sup> What PyTorch refers to as a “gather” operation is in fact a generalized version of the operation described here, where the “gathering” is performed independently in all dimensions of the input tensor. What is being referred to as a “gather” operation in this text, is implemented in PyTorch in its specialized, simplified form as `index_select`.

The scatter operation on a matrix  $\mathbf{M} \in \mathbb{R}^{n \times o}$  is an operation equivalent to  $\mathbf{A} \cdot \mathbf{M}$ , where  $\mathbf{A} \in \mathbb{R}^{m \times n}$  is a sparse matrix of values from interval  $[0, 1]$ . However, there is a restriction that for each column  $j$ , it holds that  $\sum_i [\mathbf{A}(i, j)] = 1$  (where the  $[\ ]$  operation is applied to the matrix values elementwise), meaning that each column  $j$  of matrix  $\mathbf{A}$  can have a non-zero value at exactly one position  $(i, j)$ , for some row index  $i$ . If the reduction method is summation, then values of  $\mathbf{A}$  are either 0 or 1. If mean reduction is used, then all non-zero values in a given row  $i$  are equal, summing up to 1, i.e.  $\forall j_1, j_2: \mathbf{A}(i, j_1) = \mathbf{A}(i, j_2)$  and  $\forall i: \sum_j \mathbf{A}(i, j) = 1$ . The scatter operation on a higher-dimensional tensor  $\mathbf{T}$  of shape  $(n, \dots)$  can be defined as  $\mathbf{A} \circ \mathbf{T}$ .

Based on this, using the *edge index* representation of  $\mathbf{E}$ , we can rewrite the general case of the graph convolution using  $\mathcal{G}_I$  and  $\mathcal{S}_J^\oplus$ , as shown in Equation (17). Note that  $\phi$  is applied pairwise.

$$\mathbf{V}'(i) = \gamma(\mathbf{V}(i), \mathcal{S}_{\mathbf{E}(1)}^\oplus(\phi(\mathcal{G}_{\mathbf{E}(1)}(\mathbf{V}), \mathcal{G}_{\mathbf{E}(2)}(\mathbf{V})))) \quad (17)$$

Equation (17) is how PyTorch Geometric typically computes its convolutions using the edge index representation of  $\mathbf{E}$ . The intuition behind this can be that the individual node values are first collected (*gathered*) from their individual locations in the graph into a dense tensor, then operated upon, and then placed back individually (i.e. *scattered* back) into their original positions in the graph.

Since both gather and scatter are special cases of sparse matrix multiplication, we may rewrite Eq. (17) as sparse matrix multiplication as well, using matrices  $\mathbf{A}_1, \mathbf{A}_2 \in \mathbb{R}^{|E| \times |V|}$ , corresponding to gathering using  $\mathbf{E}(1)$  and  $\mathbf{E}(2)$ , respectively, and matrix  $\mathbf{A}'_1 \in \mathbb{R}^{|V| \times |E|}$ , corresponding to scattering using  $\mathbf{E}(1)$ . See Equation (18).

$$\mathbf{V}'(i) = \gamma(\mathbf{V}(i), \mathbf{A}'_1 \circ \phi(\mathbf{A}_1 \circ \mathbf{V}, \mathbf{A}_2 \circ \mathbf{V})) \quad (18)$$

As we have seen in Section 3.4, a gather operation immediately followed by a scatter operation can be replaced with a single sparse matrix multiplication operation. This follows from the fact that both can be represented as matrix multiplications, and matrix multiplication is associative. We were able to utilize this when designing efficient computation of the GCN convolution, where the matrix multiplication with the adjacency matrix of the graph happens to be the operation that combines both the gathering and the scattering into a single operation. We cannot do this in the general case, only on a case-by-case basis given concrete graph convolutions, because in the general case, the operation  $\phi$  is arbitrary. Furthermore, GNN frameworks such as PyTorch Geometric can typically only use the result from Eq. (17), as the edge index tensor can be used for it as-is. However, they cannot use the result from Eq. (18) as-is, as the individual sparse matrices are not immediately available for use, at least not all of them (e.g. the sparse gather matrices  $\mathbf{A}_1$  and  $\mathbf{A}_2$  can be built with relative ease, by one-hot encoding  $\mathbf{E}(1)$  and  $\mathbf{E}(2)$ , respectively).

Ultimately, gather and scatter operations will be very useful for our purposes with respect to deep relational learning.

### ■ 3.4.2 Sparse Matrix Multiplication

Sparse matrix representations are designed to perform matrix multiplications involving sparse matrices such that they are less expensive both computationally, as well as in terms of memory usage. The representations typically involve a value container, holding



the non-zero values of the tensor, and an index container, holding the information on where the non-zero values are located in the tensor being represented.

The operation of particular interest to us given what we discussed in previous sections is that of “sparse-dense” matrix multiplication, i.e. matrix multiplication where the matrix on the left-hand side is in a sparse representation, the matrix on the right-hand side is dense, and the output is also a dense tensor. Methods for multiplying two such matrices, producing output in a sparse representation, also exist, as well as “sparse-sparse” multiplication algorithms, multiplying two matrices in a sparse representation together, but we will not be discussing those.

Of course, different algorithms must be used depending on the sparse representation used. We will only introduce sparse representation formats useful to us.

The so-called *coordinate list* format, also known as the COO format, represents a tensor as a pair of two tensors  $(\mathbf{D}, \mathbf{I})$ , where  $\mathbf{D} \in \mathbb{R}^n$  is the value vector ( $n$  being the total number of non-zero values), and  $\mathbf{I} \in \mathbb{N}_{0+}^{D \times n}$  is the index matrix ( $D$  being the dimensionality of the tensor being represented).  $(\mathbf{D}, \mathbf{I})$  then represents a tensor  $\mathbf{A} \in \mathbb{R}^{m_1 \times \dots \times m_D}$ . The representation works as follows: for a given index  $(i_1, \dots, i_D)$ , it holds that if there exists a  $k \in \{1, \dots, n\}$  such that  $\mathbf{I}(-, k) = (i_1, \dots, i_D)$ , then  $\mathbf{A}(i_1, \dots, i_D) = \mathbf{D}(k)$ , otherwise  $\mathbf{A}(i_1, \dots, i_D) = 0$ .

An interesting thing to point out is that for an arbitrary graph, the COO representation of its *adjacency matrix*  $\mathbf{A}$  is  $(\mathbf{D}, \mathbf{I})$  where the  $\mathbf{I}$  matrix is in fact the *edge index* of the graph.  $\mathbf{D}$  vector, on the other hand, is redundant in the case of the adjacency matrix, as it is a vector from the domain  $\{1\}^n$ , i.e. each its scalar value being 1, given the adjacency matrix definition. Therefore, using the edge index representation of the edges of a graph corresponds to using the adjacency matrix representation in the COO format.

Since the COO format (and equivalently, the edge index representation of a graph) in principle allows the  $\mathbf{I}$  matrix to contain the same  $(i_1, \dots, i_D)$  value in different columns, i.e. there may exist some  $k_1 \neq k_2$ , where  $\mathbf{I}(-, k_1) = \mathbf{I}(-, k_2)$ , COO tensors with such occurrences are known as *uncoalesced*, as they contain duplicate value entries for a single position in the tensor being represented. For example, PyTorch interprets this as the represented tensor having the *sums* of the corresponding  $\mathbf{D}$  value entries in its individual positions [77], i.e. in the sense as shown in Equation (19). Furthermore, it offers a “coalesce” operation that removes the duplicate entries from the  $\mathbf{I}$  matrix and sums the duplicate value entries together in the  $\mathbf{D}$ , keeping only up to a single value entry per each position in the represented tensor as a result. Since the indices in  $\mathbf{I}$  are also allowed to be in arbitrary order, the “coalesce” operation also updates both  $\mathbf{D}$  and  $\mathbf{I}$  such that the indices in  $\mathbf{I}$  are ordered lexicographically.

$$\mathbf{A}(i_1, \dots, i_D) = \sum_{\substack{k \in \{1, \dots, n\}, \\ \mathbf{I}(k) = (i_1, \dots, i_D)}} \mathbf{D}(k) \quad (19)$$

What is more, PyTorch supports an extension of the COO representation, where it allows the  $\mathbf{D}$  tensor to be multi-dimensional. This means that for  $\mathbf{D}$  with shape  $(n, n_1, \dots, n_N)$ , and  $\mathbf{I} \in \mathbb{N}_{0+}^{D \times n}$ , the represented tensor  $\mathbf{A}$  is  $(D + N)$ -dimensional, with  $D$  sparse and  $N$  dense dimensions, with shape  $(m_1, \dots, m_D, n_1, \dots, n_N)$ . The original definition of the COO format still applies without change.

A more efficient representation of sparse tensors is known as the *compressed sparse row* (CSR) format. It has lower requirements both computationally, as well as in terms

of memory usage. The CSR format typically represents a sparse matrix  $\mathbf{A} \in \mathbb{R}^{m_1 \times m_2}$  using a triple of vectors  $(\mathbf{D}, \mathbf{C}, \mathbf{R})$ , where  $\mathbf{D} \in \mathbb{R}^n$ ,  $\mathbf{C} \in \mathbb{N}_{0+}^n$ , and  $\mathbf{R} \in \mathbb{N}_{0+}^{m_1+1}$ .  $\mathbf{D}$  is the value vector,  $\mathbf{C}$  contains column indices, and  $\mathbf{R}$  contains row extents. Let us describe the CSR format in terms of its differences from the COO format, i.e. let  $\mathbf{I} \in \mathbb{R}^{2 \times n}$  be the corresponding COO index matrix of  $\mathbf{A}$ . The CSR format requires index pairs (i.e. the columns of  $\mathbf{I}$ ) to be ordered based on row indices, i.e.  $\mathbf{I}(1)$  is expected to be ordered, without any explicit restrictions being imposed on  $\mathbf{I}(2)$ . This means that the columns of  $\mathbf{I}$  can simply be sorted lexicographically, but permutations of individual  $(i_1, i_2)$  index pairs are allowed as well, as long as only the index pairs with matching  $i_1$  are permuted.

This restriction on  $\mathbf{I}$  allows us to define the vectors  $\mathbf{C}$  and  $\mathbf{R}$ . Using the CSR format, column indices are stored in an uncompressed way, i.e.  $\mathbf{C} = \mathbf{I}(2)$ . Row indices, on the other hand, are compressed. Since  $\mathbf{I}(1)$  is known to be ordered, it means that for each row  $r \in \{1, \dots, m_1\}$  of  $\mathbf{A}$ , there exists a *range* of indices defined by  $k_{\text{start}}^r$  (inclusive) and  $k_{\text{end}}^r$  (exclusive), where for each  $k \in (k_{\text{start}}^r, k_{\text{start}}^r + 1, \dots, k_{\text{end}}^r - 1)$ , it holds that  $\mathbf{I}(1, k) = r$ . Furthermore, from the fact that  $\mathbf{I}(1)$  is ordered, we can also infer that for any pair of consecutive row indices  $r, r + 1$ , it holds that  $k_{\text{end}}^r = k_{\text{start}}^{r+1}$ . Therefore, we can define  $\mathbf{R}$  based on  $\mathbf{I}$  as a vector of  $m_1 + 1$  total values, where for each row  $r \in (1, \dots, m_1)$  in  $\mathbf{A}$ ,  $\mathbf{R}(r) = k_{\text{start}}^r$  and  $\mathbf{R}(r + 1) = k_{\text{end}}^r$ . For rows  $r$  where all values in  $\mathbf{A}(r)$  are zero, we may simply set  $k_{\text{start}}^r = \mathbf{R}(r) = \mathbf{R}(r + 1) = k_{\text{end}}^r$ , and then no values in  $\mathbf{C}$  belong to the row  $r$ . Using this definition of  $\mathbf{R}$ ,  $\mathbf{I}(1)$  is fully reconstructible from  $\mathbf{R}$ .

Typically, of course, in algorithms operating with tensors in CSR format,  $\mathbf{I}(1)$  is not directly reconstructed from  $\mathbf{R}$ . Instead,  $\mathbf{C}$  is partitioned (sliced) based on the row extents in  $\mathbf{R}$ , where each such partition (slice) of  $\mathbf{C}$  then corresponds to a specific row  $r$  in  $\mathbf{A}$ .

The CSR format is typically extended to support multi-dimensional tensors  $\mathbf{A}$  with shapes  $(b_1, \dots, b_B, m_1, m_2, n_1, \dots, n_N)$ . Please note that the only sparse dimensions are  $m_1$  and  $m_2$ , and there cannot be more than two sparse dimensions. The support for  $(n_1, \dots, n_N)$  can be added by allowing  $\mathbf{D}$  to be a tensor of shape  $(n, n_1, \dots, n_N)$ , much like in the case of the COO format. The support for  $(b_1, \dots, b_B)$  can be added simply by operating over  $\mathbf{R}$  and  $\mathbf{C}$  in batched form, where  $\mathbf{R}$  and  $\mathbf{C}$  have shape  $(b_1, \dots, b_B, m_1 + 1)$ , and  $(b_1, \dots, b_B, n)$ , respectively. Note that this means that each batch is only able to utilize the same  $n$  values in  $\mathbf{D}$ , as  $\mathbf{D}$  itself is *not* batched. This extension of the CSR format is supported in PyTorch [77].

It is also worth mentioning that the *compressed sparse column* (CSC) format also exists, where the only difference from the CSR format is that the roles of  $\mathbf{R}$  and  $\mathbf{C}$  are reversed, in the sense that the column index representation is compressed, and the row index representation is uncompressed, i.e.  $\mathbf{R} = \mathbf{I}(1)$  and  $\mathbf{C}$  is the compressed version of  $\mathbf{I}(2)$  (assuming that  $\mathbf{I}(2)$  is ordered).

An implementation of sparse-dense matrix multiplication is typically available for all the formats discussed, in their respective extended (multi-dimensional) forms. Specifically for the COO format, PyTorch also supports an implementation of sparse-dense matrix multiplication that is able to propagate a gradient through the sparse matrix. This is something that we will not need. It holds that gradient propagation *through the right-hand side dense matrix* is supported in the matrix multiplication algorithms for all sparse matrix formats, which is sufficient for the purposes of both GNNs and deep relational learning. It is advantageous to use CSR or CSC over COO due to the CSR/CSC algorithms typically being faster.

However, it is worth noting that in terms of performance, sparse-dense matrix multiplication typically does not provide competitive performance over matrix multiplication of matrices in dense formats, despite significantly higher memory requirements in the latter case. In fact, the GPU supplier Nvidia says that matrix multiplication operations typically benefit from sparse representations only when the sparsity is over 99% [78]. GPU architectures can then typically utilize sparsity well only in specific, highly symmetric instances, as discussed e.g. in [78]. Such highly specific instances are not something we will be exploring further.

### ■ 3.4.3 Segment CSR

Now that sparse matrix representations were introduced, let us return to the scatter operation  $\mathcal{S}_J^\oplus$  for a brief moment. For specific index sequences  $J$ , we are able to apply a similar compression principle to that underlying the CSR sparse matrix format. This is possible specifically when the sequence  $J$  is ordered. Intuitively, an ordered sequence of  $J$  means that values are to be aggregated in consecutive groups of different sizes, as opposed to being scattered arbitrarily. For example,  $J = (1, 1, 1, 2, 2, 3, 4, 4, 4)$  says that  $\mathcal{S}_J^\oplus(\mathbf{T}) = \left[ \bigoplus_{i \in \{1, \dots, 3\}} \mathbf{T}(i), \bigoplus_{i \in \{4, \dots, 5\}} \mathbf{T}(i), \mathbf{T}(6), \bigoplus_{i \in \{7, \dots, 9\}} \mathbf{T}(i) \right]$ . In instances such as this one,  $J$  can simply be represented via a sequence of group totals (i.e. counts), which for this specific example is  $(3, 2, 1, 3)$ .

Typically, however, the compression is not done by building a sequence of counts, like above. Instead, the compression of  $J$  is truly done in the same way as the compression of  $\mathbf{I}(1)$  to  $\mathbf{R}$  for the CSR representation of a sparse matrix, as follows:

Since  $J \in \{1, \dots, m\}^n$  represents the mapping  $\{1, \dots, n\} \mapsto \{1, \dots, m\}$ , the compressed form of  $J$  is  $J' \in \{1, \dots, n+1\}^{m+1}$ , where for each  $o \in \{1, \dots, m\}$ , there exists a range of indices defined by  $o_{\text{start}}$  and  $o_{\text{end}}$  such that  $o_{\text{start}} = J'(o) \leq J'(o+1) = o_{\text{end}}$ , and for each  $i \in \{o_{\text{start}}, o_{\text{start}} + 1, \dots, o_{\text{end}} - 1\}$ , it holds that  $J(i) = o$ .

For the example of  $J$  presented earlier, its corresponding  $J'$  is thus  $J' = (1, 4, 6, 7, 10)$ , when indexing starting by 1 is used. In programming, as opposed to mathematical notation, zero-based indexing is typically used instead, meaning that  $J'$  should in fact be  $J' = (0, 3, 5, 6, 9)$ .

This operation is typically known as (CSR) *segmentation*.

As a middle ground between the scatter operation and the CSR segmentation operation, sometimes *COO segmentation* is also used, which is a segmentation operation that expects the same input as the generic scatter operation (i.e.  $J$  need not be compressed), where  $J$  is merely expected to be *ordered* (so that the input can also be merely grouped/segmented, not scattered in the general sense). Both segmentation operations, however, are only usable if the assumption of  $J$  being ordered is applicable, whereas the scatter operation is more universal. COO segmentation is usually computationally faster (i.e. less expensive) than the generic scatter operation, and CSR segmentation is typically even faster than the COO variant.

An interesting example to return to is that of the operations  $\mathcal{V}'_{\mathbf{E}}$  and  $\mathcal{V}'_{\mathbf{E}, \oplus}^{-1}$  from Section 3.4, where the former is a gather operation on  $I = [j \mid \forall i \in V, j \in \mathcal{N}(i)]$ , and the latter is a scatter operation on  $J = [i \mid \forall i \in V, j \in \mathcal{N}(i)]$ . Since  $J$  is ordered,  $\mathcal{V}'_{\mathbf{E}, \oplus}^{-1}$  can in fact be replaced with either a COO or a CSR segmentation operation.

In various deep learning frameworks, segmentation operations may be found under names such as `segment_csr`, `segment_coo`, `segment_reduce`, or e.g. `segment_sum`.

While TensorFlow offers the COO variant in its core library [79], PyTorch users have access to both the COO and CSR segmentation variants through an extension library developed by the author of PyTorch Geometric [80].

## Chapter 4

### NeuraLogic

NeuraLogic [24, 70] is a deep learning library that supports the full range of expressibility of *deep relational learning*. Through this paradigm, it allows us to define not only graphs and GNNs, but also advanced graph convolutions (e.g. direct sub-structure/pattern matching), heterogeneous graphs and GNNs via multiple relations and object types, hypergraphs, nested graphs, relational databases, TreeNNs [81], etc. [74]. This is done via a representation based on FOL, and the process of building individual NN architectures is based on logical inference.

Thus, covering the vectorization task for the domain of architectures producible by NeuraLogic covers the task for deep relational learning itself.

While NeuraLogic does support direct use of e.g. existing graph datasets, including heterogeneous graph datasets, or *data directly from RDBMS*, which offers the same level of expressiveness as the logic representation itself, we will briefly introduce the FOL-based representation, as it translates well to how the NN architectures are built, and is thus illustrative.

#### 4.1 Syntax

The syntax of NeuraLogic is derived from the Datalog programming language [82], which is a restricted function-free subset of Prolog [83]. Both Datalog and Prolog are programming languages rooted in FOL, where the programming is based on logical inference. Datalog, in contrast to Prolog, is a truly declarative language, where the order of clauses has no impact on execution. Datalog itself was developed as a querying language for relational databases, and is therefore very similar in capabilities to e.g. SQL (*structured query language*) [84], a popular database querying language, although the syntax of the two languages is very different.

Datalog programs consist of rules that represent FOL statements. Each rule can be thought of as an “if ..., then ...” sentence. For example, a rule “if X is a person and X is a parent of Y, then Y is a person,” can be expressed in Datalog as shown in Equation (1).

$$\text{Person}(Y) \text{ :- Person}(X), \text{Parent}(X, Y). \quad (1)$$

In general, rules in either Prolog, Datalog, or first-order logic consist of the following:

- *Constant symbols*, e.g.  $h_1$ , represent specific elements in the domain, and are typically written in lowercase.
- *Variable symbols*, e.g.  $X$ , represent any element in the domain (i.e. the rule has an implicit “for each  $X$  in the domain, it holds that ...” prefix), and are typically written in uppercase, in order to differentiate them from constants.
- A *term* represents either a constant, or a variable.
- *Predicate symbols*, e.g. “Person” or “Parent,” represent relations, i.e. properties of terms, or relationships between multiple terms.

- An *atom* is a predicate symbol applied to a term or a sequence of terms, e.g.  $\text{Parent}(X, Y)$ .
- A *literal* is an atom or the negation of an atom. Negated atoms are denoted using  $\neg$  in FOL, e.g.  $\neg\text{Person}(X)$ .

What *NeuraLogic* calls “rules” is in fact a special type of *clauses*, which are disjunctions of literals. In first-order logic, a disjunction of literals (i.e. a clause) is typically written as  $l_1 \vee \dots \vee l_n$ , where  $l_1, \dots, l_n$  are arbitrary literals, and the  $\vee$  symbol represents “or.” Similarly, a conjunction of literals would be  $l_1 \wedge \dots \wedge l_n$ , where the  $\wedge$  symbol represents “and.”

In *Datalog* and *Prolog*, a sequence of literals (separated with the comma “,” symbol) corresponds to a *conjunction* of literals.

Rules in *Datalog* and *Prolog* represent the so-called *definite clauses*, which have the disjunctive form  $h \vee \neg a_1 \vee \dots \vee \neg a_n$  in FOL, where  $h$  is a single positive literal, and  $\neg a_1 \vee \dots \vee \neg a_n$  are all negative literals. Based on the rules of first-order logic, this can be rewritten in the form of an implication  $h \Leftarrow a_1 \wedge \dots \wedge a_n$ , which is thus itself also still a definite clause (even though the right-hand side now contains a conjunction of atoms), because its truth-value table is the same. This is written in *Prolog* and *Datalog* as “ $h :- a_1, \dots, a_n.$ ” and can be seen as “if all of  $a_1, \dots, a_n$  are true, then  $h$  is true.” For a definite clause,  $h$  is known as the *head* of the rule, and  $a_1, \dots, a_n$  is known as the *body*.

A definite clause without any negative literals (i.e. with an empty body) is known as a *fact*. This is because the head literal then applies unconditionally.

With respect to relational representations (e.g. RDBMS), facts in *Datalog* correspond to the data (with most facts consisting mainly, though not exclusively, of constants), and *Datalog* rules loosely correspond to relations (e.g. foreign keys in RDBMS). To initiate the logical inference process in *Datalog*, a *query* must be posed, e.g. “ $\text{Person}(p_{\text{John}})?$ ” or “ $\text{Person}(X)?$ ” This resolves either to a truth value (i.e. “true” or “false”), or to a sequence of combinations of constants for which the query is truthful, when the variables are substituted for the constants (a process known as *grounding*). The latter is equivalent to a typical query response in RDBMS, where the output is typically in the form of a table.

*NeuraLogic* extends the *Datalog* syntax with the ability to associate *weights* with facts, as well as with individual literals within rules. The semantics of this (i.e. how the weights are used as part of the resulting neural network architecture) will be explained on an illustrative example later. Weights are either fixed tensors with explicitly defined values, or learnable tensors, i.e. the parameters of the resulting neural network.

In *NeuraLogic*, a dataset can be represented directly using *ground* facts, i.e. facts consisting only of constants.<sup>1</sup> Using the FOL syntax to represent data may perhaps seem verbose when compared to other, less expressive forms of data representations; however, this representation offers the full expressive power to represent any relational data.

The rules with a non-empty body form what is known in *NeuraLogic* as the *template*, and their atoms typically contain mainly variables. Instead of a logical inference-based algorithm being used directly to answer queries, like in *Datalog*, *NeuraLogic* uses the

<sup>1</sup> To represent a non-ground fact as part of the dataset, one must explicitly rewrite it into the form of individual ground facts. Nonetheless, this is commonly not necessary to do, as ground facts are typically all that is needed to form a dataset, without any unnecessary verbosity.

template (and logical inference) to build the (dynamic) differentiable computational graph, which forms the neural network architecture, made to answer a specific query for arbitrary input. This can be understood as a differentiable version of Datalog, or a differentiable version of a relational database.

Alternatively, the template may simply be viewed as the dynamic NN architecture itself, and the dataset as the individual inputs that the architecture will loop over. The logical inference algorithm is then a form of pre-processing of the NN architecture, with respect to the given input dataset, forming a static differentiable computational graph induced by the input dataset. During inference, where the input is potentially different from the training dataset, the pre-processing must be done as well, as the resulting computational graph again depends on the input.

## 4.2 Example

Let us use a simple example of training a classifier using NeuraLogic on a dataset of molecules. A molecule representation sufficient for the purposes of this simple example will merely encode the molecular structure itself; nothing else. Using the logic representation of the dataset in NeuraLogic, the individual atoms are represented as constants, for which an “atom” predicate  $a$  is provided (to state that the items are atoms), and the molecular structure is represented by encoding the individual bonds between pairs of atoms using a binary “bond” predicate  $b$ .

You may immediately recognize this as a mere graph representation. This is true; this example is purposefully kept simple. More complex representations, e.g. heterogeneous graphs encoding the individual atoms’ elements, are also possible in NeuraLogic, and briefly discussed later.

Let us use two examples of data samples (molecules), defined in Equations (2) and (3). (Individual facts are separated/terminated using the dot “.” symbol.) Please note that this encoding *does not differentiate individual atom elements!* To differentiate e.g. that  $o_1$  is oxygen and  $h_1$  is hydrogen, instead of a single  $a$  predicate, we would need two predicates,  $o$  and  $h$ , respectively. This would also require a slightly more complex template than the one defined below. Again, this is not done here purely to keep the example brief, as its mere purpose is understanding the general approach of NeuraLogic.

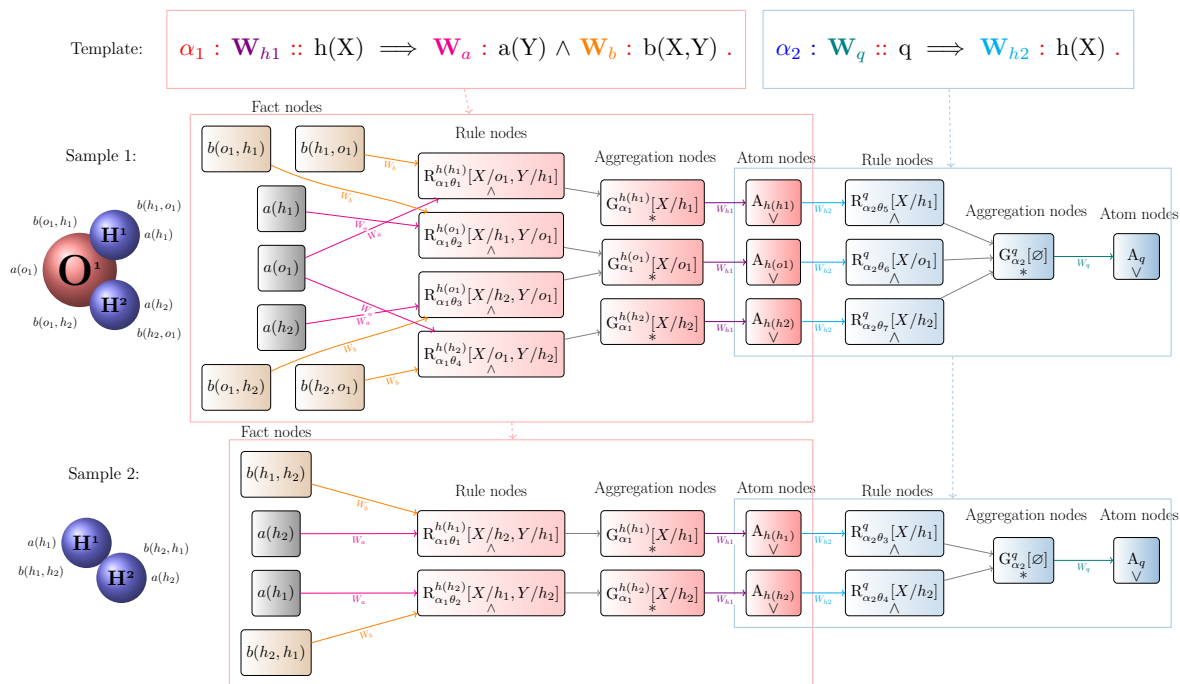
$$\begin{array}{lll} a(h_1). & a(o_1). & a(h_2). \\ b(h_1, o_1). & b(o_1, h_2). & \\ b(o_1, h_1). & b(h_2, o_1). & \end{array} \quad (2)$$

$$\begin{array}{ll} a(h_3). & a(h_4). \\ & b(h_3, h_4). \\ & b(h_4, h_3). \end{array} \quad (3)$$

Next, let us define the template using two rules. See Equation (4).

$$\begin{array}{l} h(X) :- a(Y), b(X, Y). \\ q :- h(X). \end{array} \quad (4)$$

These rules are incomplete as-is. We must explicitly state where individual weights, i.e. matrix multiplications, shall be placed. See Equation (5).



**Figure 4.1.** Example NeuroLogic computational graphs for two input examples, image taken from [70], with small modifications

$$\begin{aligned} \mathbf{W}_{h_1} h(X) &:- \mathbf{W}_a a(Y), \mathbf{W}_b b(X, Y). \\ \mathbf{W}_q q &:- \mathbf{W}_{h_2} h(X). \end{aligned} \quad (5)$$

The two rules are very simple, essentially defining a *structural* neural network with four linear layers and two aggregations in the forward pass. Firstly, all atoms and bonds are multiplied with learnable weight matrices  $\mathbf{W}_a$  and  $\mathbf{W}_b$ , respectively. Secondly, all such weighted atoms and bonds are paired based on the body of the first rule, and aggregated into a latent representation denoted by the predicate  $h$ . Based on the definition of the first rule,  $h(X)$  represents the aggregated information from all immediate neighbors of any atom  $X$ . Thirdly, all such  $h(X)$  are weighted first by  $\mathbf{W}_{h_1}$  and immediately after by  $\mathbf{W}_{h_2}$  (with a non-linearity in between), and aggregated into a single constant (or, zero-arity predicate)  $q$ . Lastly,  $q$  is weighted one last time by  $\mathbf{W}_q$ .

The resulting final computational graphs for the two examples from Eqs. (2) and (3) are drawn in Figure 4.1, where you can also see that the individual neurons form what can be perceived as “layers,” even though the full structure of the network is irregular.

Of course, a neural network such as this one must also contain non-linearities between any two consecutive linear layers. This is done in NeuroLogic either by placing a non-linear elementwise function (known in NeuroLogic as a *transformation*) after every computational node by default, or on a rule-by-rule basis using extended syntax for defining the metadata of individual rules. The aggregation method (e.g. “sum” or “mean”) can also be specified in this way.

This network is essentially a one-layer GCN (i.e. a single GCN convolution) without the normalization factor, followed by a sequence of multiple linear layers (and an additional aggregation near the end, which, if we perceive the two computational graphs as a single batched graph, is irregular). This is because the individual input samples are expressible as plain (i.e. not heterogeneous) graphs, and the first rule in Eq. (5)



encodes an aggregation across immediate neighbors in the graph, much like the GCN convolution. The second aggregation, as defined by the second rule in Eq. (5), merely aggregates all the intermediate  $h$  values into a single value.

The individual input atoms (i.e. predicates with constants in place of variables, e.g.  $a(h_2)$  or  $b(h_1, h_2)$ ) have no tensors explicitly assigned to them in this example. As such, their corresponding values are essentially implicit unit vectors, of an arbitrary shape. This means that matrix multiplication with an arbitrary weight matrix  $\mathbf{W}$  is automatically resolved to  $\mathbf{W}$  itself. This may seem to suggest that the resulting NN will not be capable of learning. On the contrary, as a result of this, the NN will be forced to learn based on the irregular structure itself of the individual input samples, encoded in the individual computational graphs.

### 4.2.1 More Complex Examples

To encode additional information into the network, we may e.g. define a heterogeneous graph by presenting individual predicates per each node type. In the context of the molecular example, a natural extension would be individual predicates per element, e.g. “Oxygen” or “Hydrogen.” Then, we can define a set of rules per each node type in the form of e.g.  $h(X) :- W_{\text{Oxygen}} \text{Oxygen}(X)$  and  $h(X) :- W_{\text{Hydrogen}} \text{Hydrogen}(X)$ . Since the implicit input atom value defaults are unit vectors, this effectively corresponds to a learnable embedding layer for the node types. Alternatively, NeuraLogic also supports direct assignment of concrete tensor values to individual predicates and/or input atoms, through which the node types could be encoded e.g. using a fixed one-hot encoding.

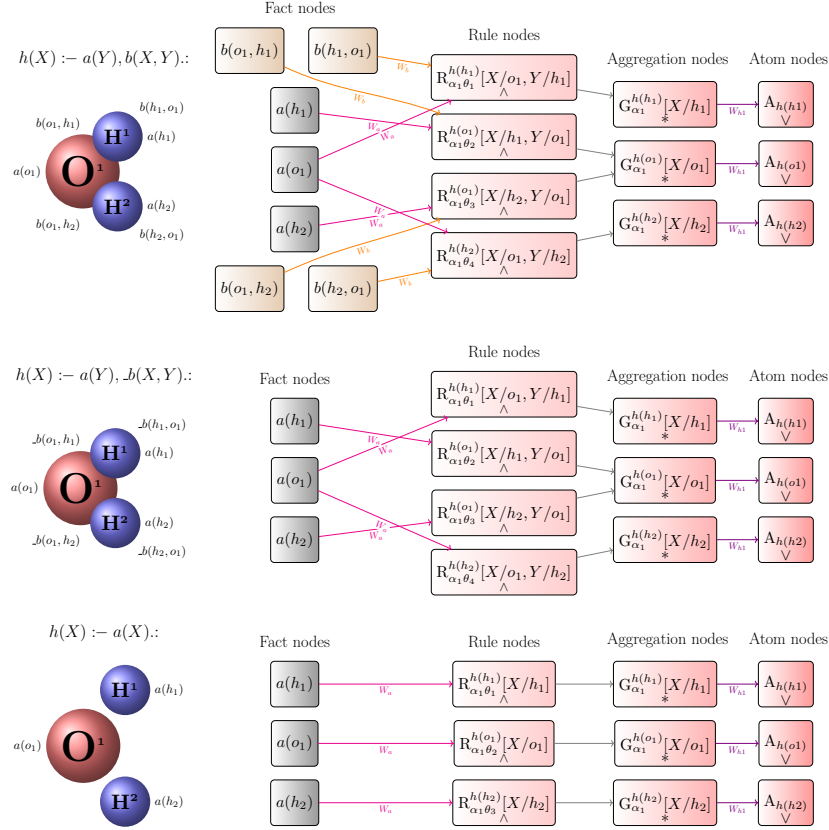
Worth noting is that the computational graphs in Fig. 4.1 are formed based on the structure determined by the  $b(X, Y)$  atoms, but they also involve the individual  $b(X, Y)$  atoms as part of the computation itself. Thus, we may either assign types to the bonds and weight<sup>2</sup> them as well, similarly as we have done it for the atoms of the molecules (e.g.  $b(X, Y) :- \mathbf{W}_{b_1} b_1(X, Y)$  and  $b(X, Y) :- \mathbf{W}_{b_2} b_2(X, Y)$ , etc.), or, we may instead remove the  $\mathbf{W}_b$  weight and mark the  $b$  predicate as *implicit*, given the fact that it adds almost<sup>3</sup> no additional useful information to the computation, other than forming the computational graph structure. Marking a predicate as implicit will result in the predicate merely affecting the computational graph structure, but not itself being directly propagated through the computational graph. In NeuraLogic, this is done by naming the predicate starting with the underscore “\_” symbol, e.g. “\_b.” In the original example, using the unweighted “\_b” bond predicate instead of  $b$  corresponds better to the actual GCN convolution definition, as the GCN convolution does not involve any  $\mathbf{e}_{ij}$  values directly either, only performs the aggregation over the  $j \in \mathcal{N}(i)$  relationships, determined by  $\mathbf{e}_{ij}$ .

To compare how the explicit/implicit predicate setting (e.g. “ $b$ ” versus “\_b”) affects the computational graphs, as well as to see the effect on the computational graphs when the “ $b$ ” predicate is removed entirely (e.g.  $h(X) :- a(X)$ ), see Figure 4.2.

NeuraLogic also supports the definition of e.g. the normalization factor  $\frac{1}{\sqrt{\deg(i)} \cdot \sqrt{\deg(j)}}$  used by the GCN convolution (as discussed in Chap. 3, p. 13), which can be defined by using additional rules with specific aggregation and transformation configurations.

<sup>2</sup> “To weight” meaning “to apply weight to,” as opposed to “to weigh,” meaning “to measure the weight of.” For more details see [85].

<sup>3</sup> It acts as bias.



**Figure 4.2.** Comparison of NeuroLogic edge predicate usages in terms of their effect on the computational graphs: explicit (top), implicit/underscored (middle), and none (bottom). Image based on the original from [70], modified.

To construct the generic case of the  $\bigoplus(\phi(\dots))$  part of the general graph convolution (as defined in Chap. 3, p. 12, Section 3.1.2), the following rule encapsulates the structural part:  $h(X) :- a(X), a(Y), b(X, Y)$ .

All of the aforementioned examples should indicate to the reader that writing more complex rules is possible, which allows designing neural networks with expressive power well beyond that of e.g. the general definition of GNN convolutions (Chap. 3, p. 12, sec. 3.1.2).

More examples, as well as a more detailed description of the example from Fig. 4.1, are available in [70].

### 4.3 Computational Graph Structure

The CPU implementation of the NeuroLogic computational graph forward/backward pass computation sees the individual nodes of the computational graph as fully independent neurons. In other words, there is no notion of “layers” in the implementation [86].

The underlying data structure of the computational graph for a given input sample begins with the output node (i.e. the single node for which there are no outgoing edges, e.g.  $A_q$  in Fig. 4.1). Each node then holds a sequence of pointers to its input nodes  $n_1, \dots, n_l$ , as well as a sequence of pointers to the weights  $\mathbf{W}_1^n, \dots, \mathbf{W}_l^n$ , which the corresponding inputs should be multiplied with (each  $i$ -th input corresponding to the

$i$ -th weight). Select neurons are also unweighted. Each neuron then also contains an optional aggregation function  $\oplus$  and an optional transformation function  $\mathcal{T}$ .

The computation of the output value  $\mathbf{V}_n$  of a neuron  $n$  (in the forward pass) then operates as follows:

1. Retrieve the individual (output) values  $\mathbf{V}_{n_1}, \dots, \mathbf{V}_{n_l}$  of the neurons  $n_1, \dots, n_l$  that are the inputs of the neuron  $n$ .
2. Optionally, for each  $\mathbf{V}_{n_i}$ , perform matrix multiplication with its corresponding weight  $\mathbf{W}_i^n$ .
  - Note: Some weights  $\mathbf{W}_i^n$  and/or some  $\mathbf{V}_{n_i}$  may be unit values. If either of the two is a unit value, the other is simply returned, instead of the actual matrix multiplication being performed. (If both are unit values, a unit value will be returned.)
3. If  $l > 1$ , apply an aggregation function  $\oplus$ .
4. Optionally, apply a transformation function  $\mathcal{T}$ .

These steps can be combined into a single Equation (6).

$$\mathbf{V}_n := \mathcal{T} \left( \bigoplus_{i \in \{1, \dots, l\}} (\mathbf{W}_i^n \odot \mathbf{V}_{n_i}) \right) \quad (6)$$

Bias is typically added as an extra input  $n_b$ , where  $b \in \{1, \dots, l\}$ , and  $\mathbf{V}_{n_b}$  is a unit value.

The computational graphs themselves begin with *fact neurons*, i.e. neurons without inputs, with fixed values, which are either unit values, or the values explicitly assigned to the inputs.

It is worth noting that while some weights  $\mathbf{W}_i^n$  are learnable, some have fixed values. For example, unit weights are not learnable.

### ■ 4.3.1 Rules as Computational Graphs

- For each ground fact from the dataset, there is a *fact neuron*.
- For each body of each grounding of each rule, there is a *rule neuron* with the individual (ground) literals as inputs. When an input literal consists of a predicate that is the result of some preceding rule(s), the input is the atom neuron (discussed below) of the predicate.
- For each head (i.e. left-hand side) of each grounding of each rule, there is an *aggregation neuron*, with the rule neurons corresponding to all applicable right-hand sides as input.
- For each predicate that is built from rules, i.e. comes as the head of at least one rule, there is an *atom neuron*, with individual rules deriving said predicate as inputs.

Or, intuitively speaking:

- Fact neurons are the dataset inputs.
- Rule neurons build the individual rules from their bodies. This is done for each instance (grounding) of each rule, separately.
- Aggregation neurons form the rules from their individual instances (groundings).
- Atom neurons build predicates by combining all the rules that define them.

This is described in greater detail in [70].

For our purposes, the following is important in terms of weights:

- Fact neurons are not weighted.
- Rule neurons have input weights (as well as inputs themselves) in the order of the elements in the rule bodies. For example, each rule neuron for  $\mathbf{W}_{h_1}h(X) :- \mathbf{W}_a a(Y), \mathbf{W}_b b(X, Y)$  has input weights  $(\mathbf{W}_a, \mathbf{W}_b)$ .
- Aggregation neurons are not weighted.
- Atom neurons have input weights (as well as inputs themselves) in the order of definition of the corresponding rules. E.g., given a unary predicate  $h$  defined by rules with heads  $\mathbf{W}_{h_1}h(X)$ ,  $h(X)$ , and  $\mathbf{W}_{h_2}h(X)$ , the input weights of its atom neuron are  $(\mathbf{W}_{h_1}, \text{Unit}, \mathbf{W}_{h_2})$ .

### ■ 4.3.2 Optimizations

Some neurons in the rule  $\rightarrow$  aggregation  $\rightarrow$  atom chain may be redundant. This may happen when a predicate is defined by only one rule, making its atom neuron redundant (as it is already built by the aggregation neuron), or when there is only one ground instance of a rule, making the aggregation neuron redundant (as the rule is built directly by the rule neuron), or when the body of a rule consists of a single unweighted literal, making the rule neuron redundant. The merging of redundant neurons is available in the optional “chain pruning” optimization.

Please note that this is not done on a “per-layer” basis, but on a per-neuron basis, as *NeuraLogic* has no concept of layers. This means that e.g. an aggregation neuron may be merged with the preceding neuron only in a few instances, when other aggregation neurons at the same depth are preserved.

A more advanced optimization, named *isomorphic compression*, is a stochastic<sup>4</sup> algorithm for merging neurons performing the same computations. If two neurons, no matter their type or location in the computational graph, are found to compute the same results (which is determined by them consistently returning identical values, irrespective of the current assignment of specific – random – values to the learnable parameters in the network), they are merged, as doing so will not change the network output with respect to the input and the learnable parameters. Of course, learnable parameters themselves cannot be (and are not) removed from the network by this. The algorithm is described in great detail in [87].

Isomorphic compression is also done on a per-neuron basis.

<sup>4</sup> With probability reaching 1 in the limit that the network output will remain unchanged by the use of this optimization algorithm.

# Chapter 5

## Implementation

The main contribution of this thesis is a computational graph processing solution for the conversion of differentiable computational DAGs (*directed acyclic graphs*) to their highly performant vectorized equivalents. The code for the implementation is available on GitHub,<sup>1</sup> and its structure is described in Appendix B.

While the result has mainly been tested on top of NeuraLogic inputs, it is important to understand that the implementation operates purely on computational graphs and is usable outside of the NeuraLogic context. The aim is to vectorize arbitrary differentiable computational graphs of deep relational learning, irrespective of how they were constructed in the first place. The approach is thus applicable on computational graphs produced by other frameworks as well. In this chapter, NeuraLogic will merely sometimes be used as an example.

The high-level idea behind the architecture of this algorithm is a two-phase approach:

The output of the first phase is a vectorized computational graph following a *generic* scheme, to which any input can be transformed. The goal of the second phase is to transform this graph further such that the final result is performant. In other words, the first phase tasks itself with the vectorization, and the second phase tasks itself with performance optimization. In terms of the outputs of the two respective phases, the domain of all possible outputs of the former is a *subset* of the latter's. In other words, the intermediate output is simpler, adhering strictly to a scheme that is consistent for any given input.

In the rest of this thesis, the first phase of the algorithm will be referred to as “the Vectorizer,” the second phase as “the Optimizer,” and the complete algorithm, consisting of the two aforementioned phases in sequence, as “the Compiler.”

The intermediate output of the Vectorizer is in theory executable as-is,<sup>2</sup> although its individual nodes translate to their most generic versions, thus potentially being computationally unnecessarily slow. This is done on purpose, so that even if no optimizations are applicable, the Compiler produces working vectorized outputs nonetheless. This two-phase approach to the Compiler architecture is motivated by the idea that even for unprecedented, extraordinarily difficult, potentially even adverse input instances, the Compiler should produce working, executable outputs. Of course, the vast majority of inputs is optimizable further; however, the two-phase design offers a fallback *inherently*, i.e., by design, even in hypothetical unprecedented situations, as the mandatory transformations operate only on a subset of the full domain of the intermediate representation(s).

The Optimizer phase works primarily by pattern matching and substituting operations in the computational graph with faster equivalents, where such substitution is imme-

<sup>1</sup> <https://github.com/neumannjan/nn-structural-graph-vectorizer-compiler>

<sup>2</sup> In practice, some transformations of this intermediate representation are still needed. However, the set of transformations necessary for the output to be executable is only a minimal subset of the full sequence of transformations involved in the Optimizer.

diately available. Furthermore, more advanced optimizations precede this, attempting more complex transformations of the computational graph, aiming to produce results consisting mainly of operations that *can* be substituted as such. The Optimizer algorithm is in the form of a sequence (a pipeline) of consecutive optimization operations, which operate in complement to one another.

The inspiration for this design comes from *programming language compilers*, i.e., algorithms that take programs written in high-level programming languages as input, and produce equivalent low-level instruction sequences, executable directly on select hardware. Higher-level programming languages typically abstract select complexities away from the programmer in exchange for improved ease of use of the language. This means that the compilers are tasked not only with the conversion itself, but also with the burden of compensating for the absence of the programmer’s input where the abstractions occur. However, a human programmer tasked with manually rewriting a program to its low-level equivalent has access to the domain knowledge of the specific program, and is thus much better equipped for the optimization task. Compilers, on the other hand, perform their optimizations without access to any specific domain knowledge for the given input. This means that man-made low-level instruction sequences typically have an edge over their compiler-produced counterparts in terms of performance. Nonetheless, compilers are typically capable of producing results with competitive performance in *most* instances, and domain knowledge is needed only in edge cases. This is similar to our task at hand.

Furthermore, the architectural design of compilers is typically also similar to that of our Compiler, performing a sequence of transformations of the input, with various intermediate representations of progressively increasing resemblance to the final representation, which is equivalent to the output language. Furthermore, various optimizations are applied along this process, also done by finding optimizable patterns and replacing them with more performant equivalents. An example programming language compiler is designed for educational purposes in [88], teaching the reader the whole process, as well as the individual optimizations involved. The design of our program is loosely inspired by this, even though it shares neither the domain, nor the individual optimizations with common programming language compilers.

## 5.1 Vectorization

The purpose of the Vectorizer is to merely convert the highly granular input computational graph, i.e., with a *high* number of nodes corresponding to operations on scalars and/or *small* tensors, to an equivalent computational graph that is of low granularity, i.e., with a *low* number of nodes, where each node corresponds to an operation on a *large* tensor. The goal is to do so such that any node in the resulting graph corresponds to an operation that can be performed in parallel on a GPU.

The terminology and syntax from Chapter 3 will be used below.

### 5.1.1 Computational Graph Definition

Firstly, let us formally define a *computational graph* by extending the definition of a graph from Section 3.1.1.

A computational graph  $G$  is a tuple of six values  $G = (V, E, \mathcal{T}, \mathcal{J}, \mathcal{S}, \mathcal{M})$ . The individual elements are defined as follows:

- $V$  and  $E$  are defined as in Sec. 3.1.1.

- $\mathcal{J}$  is a mapping  $V \mapsto \text{Seq}(V)$ , where  $\mathcal{J}(i)$  is an *ordered* sequence of the input nodes of node  $i \in V$ .  $\mathcal{J}$  determines the *order* of individual inputs of every node. Formally speaking, for each node  $i \in V$ , there is an ordered sequence of nodes  $\mathcal{J}(i)$ , where for each  $j \in V$  such that  $(j, i) \in E$ , it holds that  $j \in \mathcal{J}(i)$ .
- $\mathcal{T}$  is defined as in Sec. 3.1.1, with the following exception:  $\mathcal{T}$  does not contain value tensors for all nodes, but only for fact nodes, i.e., for the nodes  $i \in V$  that do not have any input nodes, i.e., for which  $\mathcal{J}(i)$  is an empty sequence.
- $\mathcal{S}$  is a mapping  $V \mapsto \cup_{i \in \mathbb{N}_+} \mathbb{N}_+^i$ , which determines the individual *shapes* that the individual nodes have when their values are computed. This is a substitute for non-fact nodes, which do not have a tensor in  $\mathcal{T}$ .  $\mathcal{S}(i)$  enforces a specific shape that any output of a given node  $i$  must have.<sup>3</sup>
- $\mathcal{M}$  is a mapping  $V \mapsto (\text{Seq}(\mathbb{T}) \mapsto \mathbb{T})$ , where  $\mathcal{M}(i)$  is the function needed to compute the output tensor  $\mathbf{T}$  of the node  $i$  from a sequence of tensors  $(\mathbf{T}_{j_1}, \dots, \mathbf{T}_{j_m})$ , where  $\mathcal{J}(i) = (j_1, \dots, j_m)$ . In plain English,  $\mathcal{M}(i)$  is the computation operation corresponding to the node  $i$ .

The computation is then performed for individual nodes in their topological order as follows:

1. The node  $i \in V$  receives the individual computed tensors  $\mathbf{T}_{i_1}, \dots, \mathbf{T}_{i_{|\mathcal{J}(i)|}}$  of its inputs  $i_1, \dots, i_{|\mathcal{J}(i)|} \in \mathcal{J}(i)$ . If it has  $\mathcal{T}(i)$  instead, it receives  $\mathbf{T}_i = \mathcal{T}(i)$ .
2. The output value for this node is then  $\mathbf{T}_i := \mathcal{M}(\mathbf{T}_{i_1}, \dots, \mathbf{T}_{i_{|\mathcal{J}(i)|}})$ .

### 5.1.2 The Input Computational Graph Operations

Without loss of generality, let us define the exact specifications of the operations  $\mathcal{M}(i)$  for nodes  $i \in V$  of the input DAG:

- A *fact* node  $i \in V$  has zero inputs and merely returns a tensor value. In other words, the  $\mathcal{J}(i)$  is present,  $|\mathcal{J}(i)| = 0$ , and  $\mathcal{M}(i)$  is the identity function, i.e.  $\mathcal{M}(i)(\mathbf{T}) = \mathbf{T}$ .
  - A *weight* node  $i \in V$  is a fact node. However, in the actual computational graph representation, the value  $\mathcal{T}(i)$  is not needed for weight nodes in the vectorization stage, and the shape  $\mathcal{S}(i)$  is sufficient. This is because weights are initialized immediately before the first execution of the computational graph. In the Compiler, which involves merely graph preprocessing, we do not need weight values. Their initial values may be present in  $\mathcal{T}$  optionally.
- A *linear* node  $i \in V$  performs matrix multiplication (i.e.  $\mathcal{M}(i)(\mathbf{A}, \mathbf{B}) = \mathbf{A} \circ \mathbf{B}$ ) and has exactly two inputs, i.e.  $|\mathcal{J}(i)| = 2$ , one for the left-hand side (usually the weight) and one for the right-hand side.
- An *aggregation* node  $i \in V$  has an arbitrary number of inputs, minimum of one, usually at least two. All its inputs must have the same shape. It performs aggregation of its inputs using a permutation-invariant aggregation function  $\mathcal{M}(i) = \bigoplus$ , and its output *typically* has the same shape  $\mathcal{S}(i)$  as all its individual inputs  $\mathcal{S}(j)$  for  $j \in \mathcal{J}(i)$ , though this is not enforced nor required.
- A *transformation* node  $i \in V$  must have exactly one input, i.e.  $|\mathcal{J}(i)| = 1$ . The transformation function  $\mathcal{M}(i) \in (\mathbb{T} \mapsto \mathbb{T})$  is expected to be a unary tensor function. The shape  $\mathcal{S}(i)$  is *typically* equal to the shape  $\mathcal{S}(j)$  of its only input  $j \in \mathcal{J}(i)$ .

This is different from the computational graphs produced by NeuraLogic, discussed in Chapter 4, where a node typically consists of multiple of the above: all of the matrix

<sup>3</sup> Do not confuse the graph shape mapping  $\mathcal{S}$  with the scatter operation  $\mathcal{S}_j^\oplus$ .

multiplications of its inputs, an aggregation, and a transformation, or an arbitrary subset of these three, in a single node. The individual inputs are not necessarily expected to have the same shape, but they all must have the same shape after the individual optional linear operations are applied. The NeuraLogic computational graph can be converted to the representation defined in this chapter, and vice versa, while satisfying all the conditions of the respective definitions. As such, the two computational graph representations are equivalent. For the purposes of this chapter, we will be using the input DAG definition as defined above, as it is simpler than the NeuraLogic one.

### 5.1.3 Node Group Vectorization

For an input DAG  $G = (V, E, \mathcal{T}, \mathcal{J}, \mathcal{S}, \mathcal{M})$ , *vectorizing* (or *merging*) a group of nodes  $N = (i_1, \dots, i_{|N|}) \subseteq V$  (where all nodes in  $N$  are of the same type, e.g., *fact*, or *aggregation*) refers to building a DAG  $G' = (V', E', \mathcal{T}', \mathcal{J}', \mathcal{S}', \mathcal{M}')$  where the following holds:

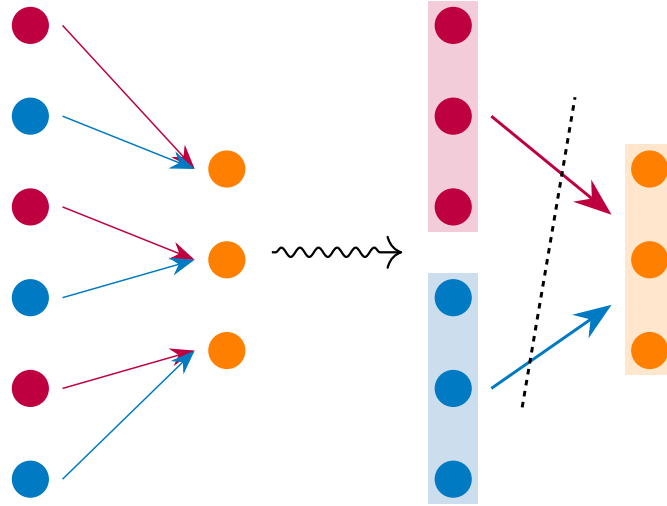
- $V' = (V \setminus N) \cup \{N\}$ , i.e. the nodes from  $N$  are replaced with a single node  $N$ .
- For any  $(i, j) \in E$  where  $i \in N, j \notin N$ , it holds that  $(N, j) \in E'$ .
- For any  $(i, j) \in E$  where  $j \in N, i \notin N$ , it holds that  $(i, N) \in E'$ .
- For any  $(i, j) \in E$  where  $i, j \notin N$ , it holds that  $(i, j) \in E'$ .
- $\mathcal{J}'$  is set as follows:
  - $\mathcal{J}'(N) := [k \mid \forall j \in (1, \dots, |N|), k \in \mathcal{J}(i_j)]$ .
  - For  $i \in V', i \neq N$ , where none of  $j \in N$  are in  $\mathcal{J}(i)$ , we may set  $\mathcal{J}'(i) := \mathcal{J}(i)$ .
  - For  $i \in V', i \neq N$ , where some  $j \in N$  are in  $\mathcal{J}(i)$ , we must set  $\mathcal{J}'(i) := (\mathcal{J}(i) \setminus N) \cup \{N\}$ . The operation  $\mathcal{M}'(i)$  must reflect this change, and it will be updated below to do so.
- $\mathcal{S}'(N) = \mathcal{S}(i_1) = \dots = \mathcal{S}(i_{|N|})$ . For any other  $j \in V', \mathcal{S}'(j) = \mathcal{S}(j)$ .
- For  $\mathcal{T}'$ , one of the two below cases applies:
  - $\mathcal{T}(i)$  is not present for any  $i \in N$ . In such case,  $\mathcal{T}'(N)$  is not present either, i.e.  $\mathcal{T}' = \mathcal{T}$ .
  - $N$  consists of fact nodes, meaning that  $\mathcal{T}(i)$  is present for all  $i \in N$ . Then,  $\mathcal{T}'(N) = [\mathcal{T}(i_j) \mid j \in (1, \dots, |N|)]$ . For any other  $i \in V', \mathcal{T}'(i) = \mathcal{T}(i)$  if  $\mathcal{T}(i)$  is present.
- $\mathcal{M}'(N)$  is determined based on the type of the nodes in  $N$  (e.g. *fact* or *aggregation*). This is explained in great detail below. For any other  $i \in V', \mathcal{M}'(i) = \mathcal{M}(i)$ .

In a more concrete sense, *vectorizing* refers to the underlying node operation(s) specifically, whereas *merging* is the term for the structural changes to the graph. However, since one cannot be done without the other, the terms may also be used interchangeably.

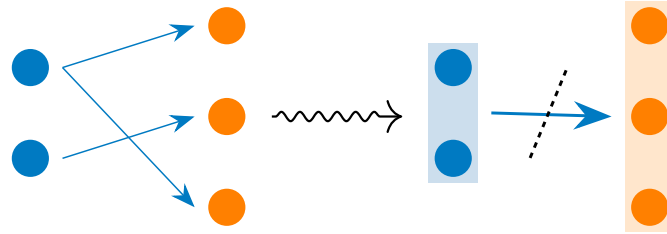
It goes without saying that only *pairwise independent* nodes can be merged, which can be defined formally as follows: Let  $G$  be a (directed) graph. Nodes  $i, j \in V$  can be defined as *dependent* when there exists a directed path between them, i.e. a sequence of nodes  $i, i_1, \dots, i_k, j \in V$  such that  $(i, i_1) \in E$ , for each  $l \in \{1, \dots, k-1\}$  it holds that  $(i_l, i_{l+1}) \in E$ , and  $(i_k, j) \in E$ . Nodes  $j, i \in V$  are dependent if  $i, j \in V$  are dependent. Nodes  $i, j \in V$  are *independent* if and only if they are not dependent. A set of nodes  $N \subseteq V$  is *pairwise independent* if for all pairs  $i, j \in N$  it holds that  $i$  and  $j$  are independent.

Eventually, all original nodes  $i \in V$  must be part of some  $N \subset V, N \in V'$ , even if the only possibility for a given  $i \in V$  is the trivial one, i.e.  $N = (i)$ . Therefore, eventually, after all vectorizations are performed,  $V' \cap V = \emptyset$ , and  $V'$  consists of disjoint sets of values from  $V$ . The resulting graph then is the *fully vectorized* graph  $G' = (V', \dots, \mathcal{M}')$ , where it holds that all  $\mathcal{M}'(i)$  operate on tensors that correspond to *lists of tensors from*





**Figure 5.1.** Example of broken input order as a result of vectorization due to interleaving. Different node colors indicate different node types and/or parameters, where nodes of equal color are vectorizable together.



**Figure 5.2.** Example of broken input order as a result of vectorization due to repetition. Different node colors indicate different node types and/or parameters, where nodes of equal color are vectorizable together.

the original graph  $G = (V, \dots, \mathcal{M})$ . In other words, any  $\mathbf{T}$  on input of any  $\mathcal{M}'(i)$  is expected to have shape  $(|N|, \mathbf{S})$  for some  $N \subset V$ , where for each  $j \in N$ ,  $\mathcal{S}(j) = \mathbf{S}$ . The operations  $\mathcal{M}'(i)$  are thus *vectorized* in the sense that they operate on vectorized inputs, and produce vectorized outputs.

#### 5.1.4 Vectorized Input Order Discrepancy

When the whole graph is vectorized as described above, *input order will inevitably be broken for most newly created nodes!* If there is a group of nodes  $N_1$  that can be vectorized, but the nodes are used as input of some subsequent nodes  $N_s$  in an order where they are *interleaved* with some other inputs  $N_2$ , then by vectorizing  $N_1$  we enforce a *consecutive* order on  $N_1$ , meaning that  $N_1$  values can no longer be interleaved with  $N_2$  on the input of  $N_s$ . This is illustrated in Figure 5.1. Similarly, if nodes from  $N_1$  are used as inputs by some  $N_s$  in a different order than  $N_1$  establishes, possibly even with some inputs requesting nodes from  $N_1$  repeatedly, input order is then also broken for  $N_s$ . This is illustrated in Figure 5.2.

Therefore, the change in the order of individual input values must be reflected in the vectorized nodes, so that they continue to produce equivalent results. Specifically, a node must reorder its inputs back to the expected order, prior to performing its computation. To do this, the node may use a *multi-tensor gather* operation.

A *multi-tensor gather* operation  $\bar{\mathcal{G}}_{I'}: \mathbb{T} \times \dots \times \mathbb{T} \mapsto \mathbb{T}$  is a gather operation performed on multiple tensors, using the index sequence  $I' \in \text{Seq}(\{1, \dots, N\} \times \mathbb{N}_+)$ . This allows a vectorized node to take its vectorized inputs  $\mathbf{T}_1, \dots, \mathbf{T}_N$  and reorder their individual values *in arbitrary order* into a single tensor, also allowing for interleaving values from different inputs, or for repetition of select inputs. The operation can be defined as shown in Equation (1).

$$\bar{\mathcal{G}}_{I'}(\mathbf{T}_1, \dots, \mathbf{T}_N) := [\mathbf{T}_i(j) \mid (i, j) \in I'] \quad (1)$$

For example, given two input tensors  $\mathbf{T}_1$  and  $\mathbf{T}_2$  of equal shapes  $\mathbf{S} = (N, \dots)$ , to gather their individual elements in an alternating order, we may use multi-tensor gather with the index sequence  $I'$  from Equation (2).

$$I' = ((1, 1), (2, 1), (1, 2), (2, 2), \dots, (1, N), (2, N)) \quad (2)$$

A vectorized implementation of such an operation is typically not available, e.g., for GPUs. Therefore, it is in practice replaced with a concatenation operation followed by a gather operation. The former concatenates the individual vectorized inputs along the first dimension into a single tensor. The latter reorders the individual values in the expected order. For the above  $I'$  example, given input tensors  $\mathbf{T}_1$  and  $\mathbf{T}_2$ , the concatenation operation could produce  $\mathbf{T} = [\mathbf{T}_1, \mathbf{T}_2]$ , and the subsequent gather operation  $\mathcal{G}_I$  would then use the index sequence  $I = (1, N + 1, 2, N + 2, \dots, N, N + N)$ .

To define the concatenation operation formally, see Equation (3).

$$\text{Concat}(\mathbf{T}_1, \dots, \mathbf{T}_N) := [\mathbf{T}_1, \dots, \mathbf{T}_N] \quad (3)$$

In general, given input tensors  $\mathbf{T}_1, \dots, \mathbf{T}_N$ , with respective shapes  $\mathbf{S}_1 = (l_1, \mathbf{S}), \dots, \mathbf{S}_N = (l_N, \mathbf{S})$ , the multi-tensor gather operation  $\bar{\mathcal{G}}_{I'}$  can be rewritten using  $\text{Concat}$  and  $\mathcal{G}_I$  as shown in Equation (4).

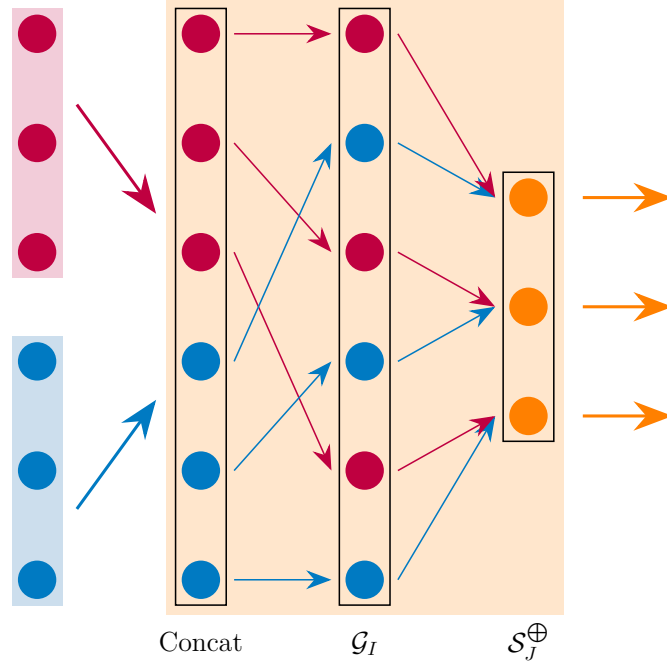
$$\begin{aligned} \bar{\mathcal{G}}(\mathbf{T}_1, \dots, \mathbf{T}_N) &= \mathcal{G}_I(\text{Concat}(\mathbf{T}_1, \dots, \mathbf{T}_N)), \\ \text{where } I &:= \left[ \left( \sum_{j \in \{1, \dots, n-1\}} l_j \right) + i \mid (n, i) \in I' \right] \end{aligned} \quad (4)$$

Figure 5.3 shows how this fixes the example from Fig. 5.1.

### 5.1.5 The Vectorized Computational Graph

Let us list how all the input nodes can be merged and vectorized, and what the resulting nodes look like:

- *Transformation* nodes  $N = (i_1, \dots, i_{|N|}) \subseteq V$  can be merged together as long as they are pairwise independent, they perform the same transformation operation  $M = \mathcal{M}(i_1) = \dots = \mathcal{M}(i_m)$ , and their shapes are equal, i.e.  $\mathbf{S} = \mathcal{S}(i_1) = \dots = \mathcal{S}(i_m)$ . The resulting operation  $\mathcal{M}'(N)$  is the same transformation  $\mathbb{T}^{\mathbf{S}} \mapsto \mathbb{T}^{\mathbf{S}'}$  as for any of the individual original nodes. However, the transformation now must be “broadcasted,” i.e., it must operate on a batched tensor of shape  $(\sum_{i \in \{1, \dots, |N|\}} |\mathcal{J}(i)|, \mathbf{S})$ . The output tensor has shape  $(\sum_{i \in \{1, \dots, |N|\}} |\mathcal{J}(i)|, \mathbf{S}')$ , where typically  $\mathbf{S}' = \mathbf{S}$ . The transformation is applied independently across the first dimension of the input tensor. When



**Figure 5.3.** Example of a multi-input gather solution for the problem from Fig. 5.1. The vectorized node (highlighted in light orange) now consists of three vectorized operations applied in sequence:  $\text{Concat}$ ,  $\mathcal{G}_I$ , and  $\mathcal{S}_J^\oplus$ . The first two serve the purpose of re-arranging input values back to the expected order, before the main operation (in this case aggregation) is applied.

combined with the preceding  $\text{concat}+\text{gather}$  operation, the resulting operation  $\mathcal{M}'(i)$  looks as follows ( $\gamma$  refers to the transformation itself):

$$\mathcal{M}'(i)(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|}) = \gamma(\mathcal{G}_I(\text{Concat}(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|})))$$

- *Aggregation* nodes  $N = (i_1, \dots, i_{|N|}) \subseteq V$  can be merged together as long as  $\oplus = \mathcal{M}(i_1) = \dots = \mathcal{M}(i_m)$  (please note that each original node  $i \in N$  is allowed to have a different number of inputs to be aggregated), and their input and output shapes are equal, i.e.  $\oplus: \mathbb{T}^{\mathbf{S}} \times \dots \times \mathbb{T}^{\mathbf{S}} \mapsto \mathbb{T}^{\mathbf{S}'}$ . The equivalent vectorized operation of the vectorized node is the scatter operation  $\mathcal{S}_J^\oplus$ . More specifically, it is in fact a segmentation operation as defined in Section 3.4.3. As such, it can be parametrized using a sequence of counts  $C = [|\mathcal{J}(i_j)| \mid \forall j \in (1, \dots, |N|)]$ . The input shape is  $(\sum_{i \in (1, \dots, |N|)} |\mathcal{J}(i)|, \mathbf{S})$  and the output shape is  $\mathcal{S}'(N) = (|N|, \mathbf{S}')$ . When combined with the preceding  $\text{concat}+\text{gather}$  operation, the resulting operation  $\mathcal{M}'(i)$  looks as follows:

$$\mathcal{M}'(i)(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|}) = \mathcal{S}_J^\oplus(\mathcal{G}_I(\text{Concat}(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|})))$$

- *Fact* nodes  $N = (i_1, \dots, i_{|N|}) \subseteq V$  can be merged together as long as they have identical shapes  $\mathbf{S} = \mathcal{S}(i_1) = \dots = \mathcal{S}(i_m)$ . Then, the resulting tensor is  $\mathcal{J}'(N) = \mathbf{T} = [\mathcal{J}(i_j) \mid j \in (1, \dots, |N|)]$ . The operation  $\mathcal{M}'(N)$  is still the identity function.
- *Linear* nodes  $N = (i_1, \dots, i_{|N|}) \subseteq V$  can be merged together as long as they have identical left-hand side shapes, and identical right-hand side shapes (note that for all  $i \in N$ , it holds that  $|\mathcal{J}(i)| = 2$ ). In other words,  $\mathcal{S}(\mathcal{J}(i_1)_1) = \dots = \mathcal{S}(\mathcal{J}(i_{|N|})_1)$ , and  $\mathcal{S}(\mathcal{J}(i_1)_2) = \dots = \mathcal{S}(\mathcal{J}(i_{|N|})_2)$ . Assuming that  $\mathbf{W}(i)$  are the left-hand side value tensors, and  $\mathbf{X}(i)$  are the right-hand side value tensors of all  $i \in N$ , the vectorized operation is then  $\phi(N)(\mathbf{W}, \mathbf{X}) = [\mathbf{W}(i_j) \cdot \mathbf{X}(i_j) \mid j \in (1, \dots, |N|)] = \mathbf{W} \circ \mathbf{X}$ . The

output shape is thus  $\mathcal{S}'(N) = (|N|, \mathbf{S})$ , where  $\mathbf{S} = \mathcal{S}(i_1) = \dots = \mathcal{S}(i_{|N|})$ . To obtain  $\mathbf{W}$  and  $\mathbf{X}$  from the actual vectorized node inputs  $(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|})$ , we must use concat+gather, as follows:

$$\begin{aligned}\mathbf{W} &:= \mathcal{G}_{I_1}(\text{Concat}(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|})) \\ \mathbf{X} &:= \mathcal{G}_{I_2}(\text{Concat}(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|}))\end{aligned}$$

To summarize, the following are the individual vectorized operations for the different node types, including the concatenation/gather operations:

- Fact nodes:  $\mathcal{M}'(i)(\mathbf{T}) = \mathbf{T}$ .
- Transformation nodes:  $\mathcal{M}'(i)(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|}) = \gamma(\mathcal{G}_I(\text{Concat}(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|})))$ .
- Aggregation nodes:  $\mathcal{M}'(i)(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|}) = \mathcal{S}_J^\oplus(\mathcal{G}_I(\text{Concat}(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|})))$ .
- Linear nodes:

$$\begin{aligned}\mathcal{M}'(i)(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|}) &= \\ &= \mathcal{G}_{I_1}(\text{Concat}(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|})) \circ \mathcal{G}_{I_2}(\text{Concat}(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|}))\end{aligned}$$

Linear nodes may be simplified further, as the two gathers only need to operate on their respective subsets of the input tensors  $\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|}$ . Therefore, we may instead redefine its operation  $\mathcal{M}'(i)$  (also updating its input order  $\mathcal{J}'(i)$  accordingly) as follows:

$$\begin{aligned}\mathcal{M}'(i)(\mathbf{W}_1, \dots, \mathbf{W}_W, \mathbf{T}_1, \dots, \mathbf{T}_T) &= \\ &= \mathcal{G}_{I'_1}(\text{Concat}(\mathbf{W}_1, \dots, \mathbf{W}_W)) \circ \mathcal{G}_{I'_2}(\text{Concat}(\mathbf{T}_1, \dots, \mathbf{T}_T))\end{aligned}$$

The resulting vectorized computational graph is also a DAG.

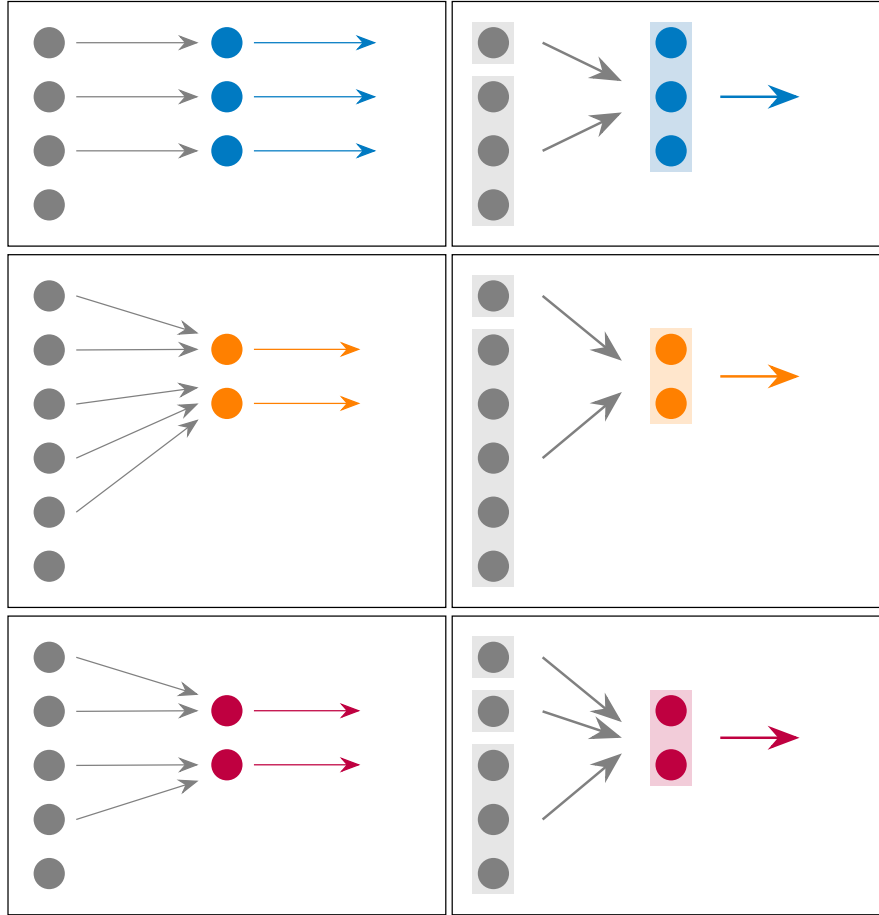
Figure 5.4 illustrates how individual operations (transformation, aggregation, and linear) are vectorized, and what the resulting computational graphs may look like with respect to the nodes' inputs, vectorized independently.

### ■ 5.1.6 Note About Aggregation Nodes

Figure 5.3 may suggest that the gather operation is redundant when immediately followed by a scatter operation, and that using the concatenation operation by itself with the scatter operation is sufficient. This is true in this particular example. However, keep in mind that the gather operation may also reuse input nodes multiple times in preparation for the scatter operation, and the scatter operation cannot use input values multiple times by itself. This was discussed in greater detail in Chapter 3, Section 3.4.1 (p. 19). The use of the gather operation also allows us to use a segmentation operation in *every* instance, instead of the generic scatter operation.

However, in instances where each input value is used exactly once, a scatter operation can be used without a preceding gather operation, instead of a gather operation followed by the segmentation operation.

Alternatively, sparse matrix multiplication can be used in place of any gather+scatter sequence.



**Figure 5.4.** Vectorization before/after for individual operation types: transformation (blue), aggregation (orange), and linear (purple). Notice that the vectorization of the input nodes operates completely independently, which is illustrated using gray nodes, of arbitrary configurations. This is why the total number of inputs for each node is arbitrary after vectorization, and is re-gathered appropriately at the beginning of each vectorized node.

### 5.1.7 Generalization

We could generalize this further and allow arbitrary vectorizable differentiable  $n$ -ary functions  $\odot$ , where the corresponding node operations would then follow a general scheme as follows:

$$\mathcal{M}'(i)(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|}) := \odot_{j \in \{1, \dots, n\}} \mathcal{G}_{I_j}(\text{Concat}(\mathbf{T}_1, \dots, \mathbf{T}_{|\mathcal{J}'(i)|}))$$

Such general scheme encompasses all of the node types that we defined, and others.

However, this is disadvantageous for the Optimizer, for two reasons: Firstly, many optimizations target the linear operations specifically. Secondly, many optimizations reduce the total number of gather operations by propagating gathers *through* aggregation/transformation layers. This is trivial for the transformation layers as defined, and for the aggregation layers, requires access to the metadata of their scatter operations. In other words, such optimizations may not be possible for general  $n$ -ary functions.

Support for additional operations may in the future be added, but it is currently not implemented. Instead, it is encouraged to construct more advanced computations via

the composition of existing building blocks. For example, to compute the normalization factor  $\frac{1}{\sqrt{\deg(i) \cdot \deg(j)}}$ , one could use aggregation nodes (with “count” aggregation) for  $\deg(i)$ , followed by aggregation nodes (with scalar multiplication aggregation) for their pairwise products, followed by  $\gamma(x) := \frac{1}{\sqrt{x}}$  transformation nodes.

### ■ 5.1.8 The Batching Problem

The remaining topic left to discuss is the following question: How to choose the groups of nodes to merge together? While the question may appear simple, it is in fact no different from the NP-hard “batching” problem of dynamic computational graphs, discussed in Chapter 2, Section 2.1.1 (p. 6). This is especially the case when the inputs are batched, i.e. multiple input samples (independent sub-graphs) are contained in the computational graph; however, it might also be the case even when vectorizing a computational graph of only a single input sample.

However, specifically in the case of NeuraLogic, we may utilize domain knowledge and simply group nodes based on from which FOL rules they were created. Even better: we may group all nodes for which the *left-hand side predicate matches*. For example, if there are rules “ $\mathbf{W}_1 h(X) :- a(Y), b(X, Y)$ ” and “ $\mathbf{W}_2 h(X) :- a(X)$ ,” all of their nodes can be grouped together. In other words, the grouping occurs loosely based on the intuition of “layers” that you may intuitively have when looking at, e.g., Figure 4.1 (p. 28).

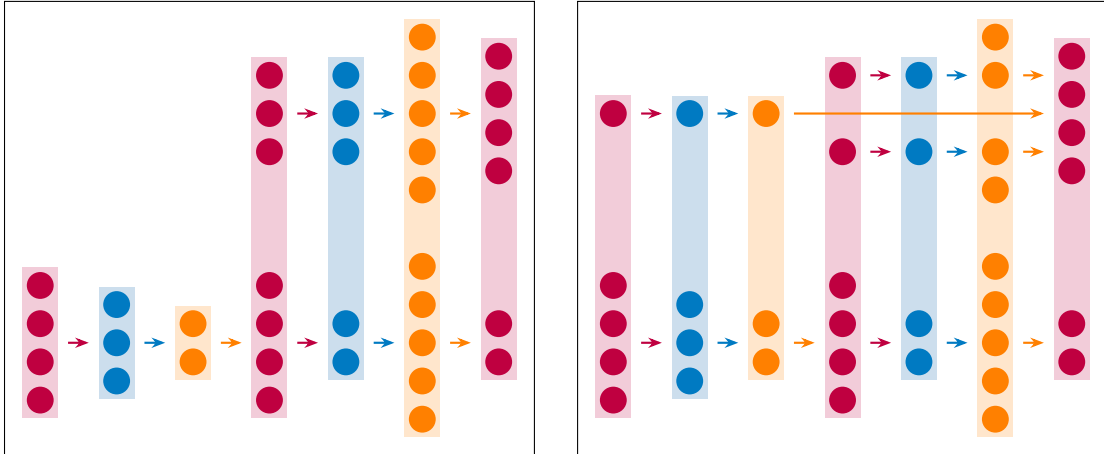
Of course, not *all* nodes can be grouped as such; we must split them further into groups based on their aggregation/transformation configurations, as well as the dimensions of their individual tensors.

For the example of the two rules “ $\mathbf{W}_1 h(X) :- a(Y), b(X, Y)$ ”, and “ $\mathbf{W}_2 h(X) :- a(X)$ ”, this means specifically:

- A *single* vectorized aggregation node will collect and aggregate the values for all ground rules of *both* the template rules at once. This requires gathering the values of all ground rules for both right-hand sides, i.e for all of  $a(X)$ , and all pairs of  $a(Y)$  and  $b(X, Y)$ , and then aggregating such that all corresponding pairs of  $a(Y)$  and  $b(X, Y)$  are aggregated, and all  $a(X)$  are propagated as-is. (In other words, the corresponding scatter/segmentation operation will have counts of 2 for the first rule, and counts of 1 for the second rule.) The output tensor has values for all  $h(X)$  from the first rule, as well as all  $h(X)$  from the second rule.
- A *single* vectorized linear node will then perform the matrix multiplication, gathering  $\mathbf{W}_1$  and  $\mathbf{W}_2$  for the left-hand side of the operation, depending on the rule sources of the corresponding values on the right-hand side.

There are some situations where multiple rules with matching left-hand side predicates, or even a single rule, may produce multiple vectorized nodes:

- A single rule  $h(\dots) :- \mathbf{W}_a a(\dots), \mathbf{W}_b b(\dots)$ , where tensors  $\mathbf{W}_a$  and  $\mathbf{W}_b$  have different dimensions, or where tensors for the predicates  $a$  and  $b$  have different dimensions, require two separate vectorized linear nodes. For subsequent aggregation, the dimensions must match nonetheless, and as such a single aggregation node remains sufficient.
- Two rules with the same predicate, but a different aggregation function require separate aggregation nodes.



**Figure 5.5.** Comparison of an optimal and a suboptimal grouping. Different node colors indicate different node types. Light-colored rectangles indicate how vectorization can be performed. Notice that there is an exponential number of suboptimal groupings with unnecessary skip connections (e.g., the one on the right), and only a single possible solution without skip connections (the one on the left).

- Two rules with the same predicate, but a different transformation function require separate transformation nodes.
- Two rules  $\mathbf{W}_1 h(\dots) :- \dots$  and  $\mathbf{W}_2 h(\dots) :- \dots$ , where tensors  $\mathbf{W}_1$  and  $\mathbf{W}_2$  have different dimensions, require two separate vectorized linear nodes. For subsequent aggregation, the dimensions must match nonetheless, and as such a single aggregation node remains sufficient.

It may happen that this approach to node grouping produces situations where multiple independent groups are still groupable, despite having different left-hand side predicates, as their node types, aggregation/transformation configurations, and operation input/output shapes are by chance equivalent. In such case the grouping is not optimal. The solution is trivial, though: the groups can simply be merged into one.<sup>4</sup>

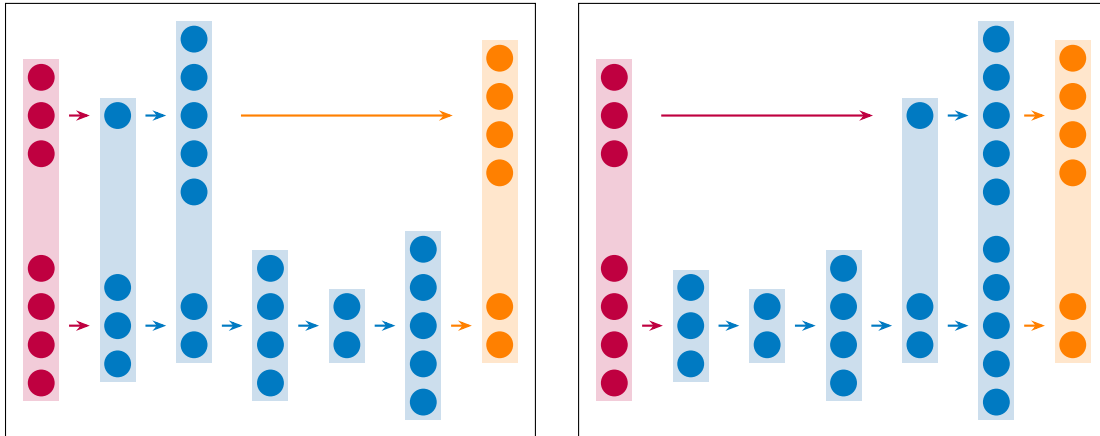
The reader may argue that the domain knowledge is unnecessary; that it is sufficient to group nodes based on their node types, dimensions, and operation configurations. This is not true, as without the domain knowledge, an exponential number of solutions, most of which are suboptimal, is possible. What happens when grouping is performed as such is illustrated in Fig. 5.5. To avoid this, a depth-based algorithm could be used, similar to that used by TensorFlow Fold. However, using depth is also suboptimal, as problematic examples still exist, such as those discussed in [31; Fig. 1], which are avoidable when domain knowledge is used.

In NeuraLogic instances where different input samples resolve to different depths due to recursive rule definitions, the recursively expanded layers can be merged across samples in two ways to minimize skip connections: at the beginning, or at the end. This is illustrated in Figure 5.6. The better choice of the two could be made e.g. such that memory consumption (i.e., the number of nodes in a single layer) is minimized.<sup>5</sup>

It is important to note that *this does not necessarily solve the NP-hard batching problem*, and the solution is not guaranteed to be optimal for NeuraLogic. However, the

<sup>4</sup> At the time of writing, this is not implemented, though doing so is not necessarily particularly difficult. The grouping is purely based on the predicates.

<sup>5</sup> At the time of writing, recursive NeuraLogic NNs have not been tested.



**Figure 5.6.** Example of NeuraLogic computational graphs with recursive rules, with different depths across independent sub-graphs. Note that the situation from Fig. 5.5 cannot occur, as domain knowledge is used, and thus any solution will properly follow layers. In this example, there is at least one skip connection needed nonetheless; however, there are exactly two solutions that do not have more than one skip connection, and we may choose either of them.

approach taken, which groups together nodes with corresponding semantics using domain knowledge from the NeuraLogic architecture design, is highly unlikely to produce problematic solutions in the vein of those discussed e.g. in [31; Fig. 1], as the domain knowledge-informed algorithm is not depth-based. This indicates that the groupings will be close to optimal for most computational graphs. For example, for many GNNs, including GCN or GraphSAGE [57], the solutions are optimal, since their resulting computational graphs do not contain any skip connections.

Please note that *none of the FOL syntax is present in the Vectorizer*. The Vectorizer operates on *computational graphs* directly, and is completely independent from NeuraLogic, suitable for applications on arbitrary computational graphs, unrelated to either FOL or NeuraLogic.

However, *no automated solver has been implemented for solving the batching/grouping problem in the general case*. The input API asks the user to input the computational graph(s) in pre-grouped form, which is how the domain knowledge-based grouping is done in the NeuraLogic case. It is up to the user of the Compiler to decide what grouping approach they choose to use for any custom computational graphs of their own.

The Vectorizer is thus very simple, merely producing the initial vectorized computational graph representation for the Optimizer, which is the main point of interest, as it ensures the vectorized computational graph is performant; a property that is *not* the main point of focus for the Vectorizer.<sup>6</sup>

### ■ 5.1.9 Implementation Details

At the time of writing, the actual Vectorizer implementation outputs all vectorized nodes as nodes of a single type. In other words, a single node performs the concatenate+gather operation(s), followed by an optional vectorized matrix multiplication, followed by an optional vectorized aggregation, and lastly an optional vectorized trans-

<sup>6</sup> Of course, the grouping problem must be resolved at the Vectorizer stage, which obviously affects performance, as the Optimizer expects this to be resolved on its input.



formation. This representation is equivalent to that described in this text, where the individual operations are represented as separate nodes. Any computational graph written using the former representation is convertible to the latter, and vice versa. Therefore, this is merely an implementation detail.

## 5.2 Optimizations

Before the motivation for optimizations is explained, let us first define some additional terminology to be used with vectorized computational graphs that are the immediate outputs of the Vectorizer. This will help make the discussion less verbose. Given a vectorized node  $i \in V'$ , let us define:

- The *output length* of  $i$  as the first (batch) dimension of its output tensor, i.e.,  $\mathcal{S}'(i)(1)$ .
- The *input length* of  $i$  as the sum of the output lengths of its input nodes:  $\sum_{j \in \mathcal{J}'(i)} \mathcal{S}'(j)(1)$ .
- The *inner length* of  $i$  as the length  $|I|$  of the index sequence  $I$  of any of its gather operations  $\mathcal{G}_I$ .

The computational graph output of the Vectorizer consists of a low number of nodes, where each node is typically “wide”, i.e., it has large inner/output lengths, meaning that it broadcasts the same operation across a large number of values. No matter the specifics, *this is done in any case by first resizing (gathering) the inputs from their input length to the inner length*, and then performing the broadcasted operation.

For example, for a linear node, this requires two gather operations, one for the left-hand side, and one for the right-hand side, both of which expand the two respective input tensors to the (wide) inner length, which only then is followed by the (wide) broadcasted matrix multiplication. However, for many linear nodes, at least one side (typically, but not exclusively, the weight side) has the number of unique inputs significantly lower than the inner length of the node. For example, a linear node may apply matrix multiplication  $n = 2000$  times, using only 3 unique weight inputs. It will nonetheless expand the concatenated weight tensor of shape  $(3, \dots)$  to shape  $(2000, \dots)$  first, before performing the matrix multiplication itself. This is redundant, which becomes even more apparent when the node only uses a single weight, meaning that it duplicates it 2000 times before performing the matrix multiplication, even though each of the 2000 values is the same value. A different interesting example is one where the linear layer multiplies every possible combination of its weight and input value pairs, as in such case both the lengths  $a$  and  $b$  of the two input sides are significantly lower than the output length  $n = a * b$ . In such case, both inputs have their values duplicated many times over redundantly.

What is more, the possibility for optimizations does not end here. We may do two following things:

- a) Remove duplicates from, e.g., linear nodes by propagating gathers “downwards,” i.e., to subsequent layers, thereby decreasing the memory footprint at the expense of (possibly) additional gather operations.
- b) Or, reduce the total number of gather operations by propagating gathers “upwards,” i.e., to preceding layers, thereby duplicating nodes and increasing memory footprint at the benefit of lower computational overhead.

For an arbitrary input computational graph, a balancing act between a) and b) can be done, where depending on its memory footprint, different configurations may yield different performance. Overall, a) is applied first to remove unnecessary duplicates nonetheless, and then the strength of b) can be configured to achieve different results.

Lastly, other low-hanging fruit optimizations can also be done, such as the replacement of gather/scatter operations with simpler equivalents in specific instances.

### ■ 5.2.1 Basic Gather Optimizations

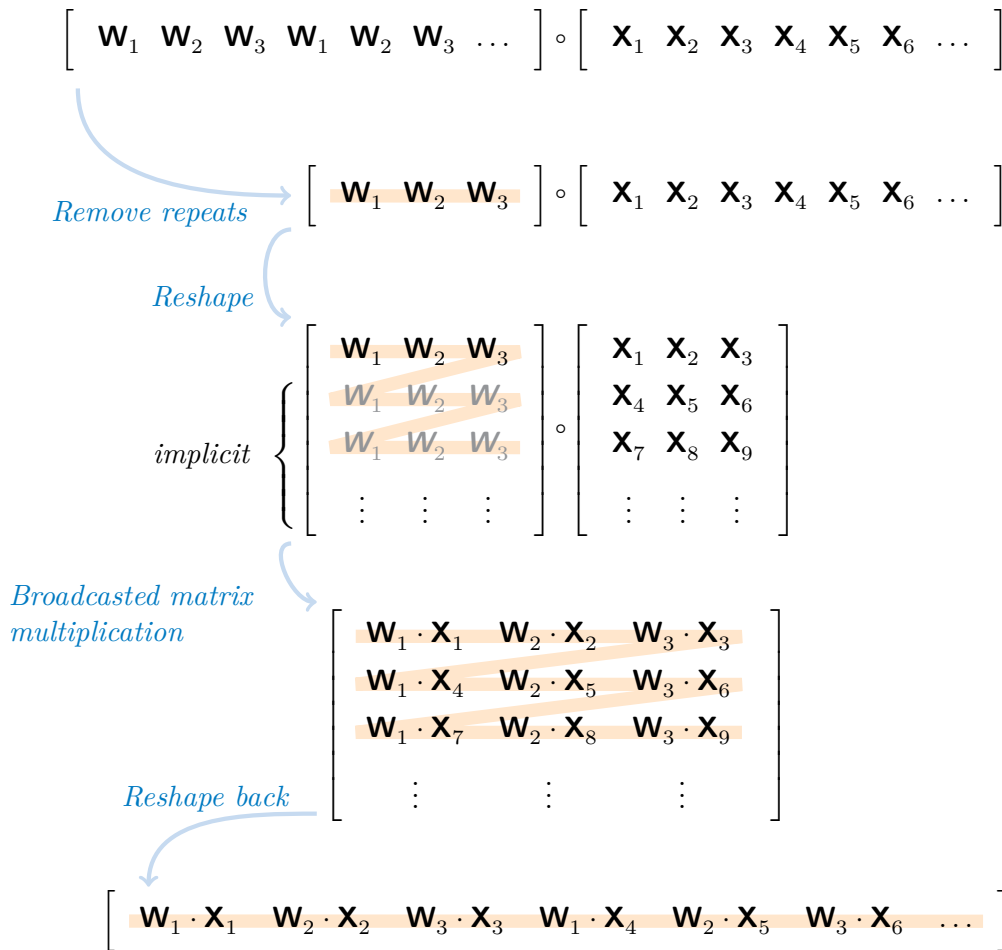
When individual gathers have very specific layouts, they can be simplified:

- When a gather only retrieves a single value, instead of using the computationally involved gather algorithm, we may simply retrieve the single value by indexing into the tensor.
- When a gather retrieves values in increasing order such that every consecutive index is exactly  $l$  values greater than the previous, we may simply use the symmetrical (and thus easier to parallelize) slicing operation. In other words, if for each  $i \in \{1, \dots, |I|\}$  it holds that  $I_{i+1} = I_i + l$ , then we may simply take each  $l$ -th value from the slice of the tensor starting at index  $I_1$  and ending at index  $I_{|I|}$  (incl.).
  - This of course simplifies further when  $l = 1$ , where we simply take the slice without skipping values.
  - Even further, when  $l = 1$ ,  $I_1 = 1$ , and  $|I|$  is equal to the input length, then the gather operation can be skipped entirely, as its output is equal to its input.
- When the gather index sequence  $I$  consists of a subsequence of values  $I_s$  repeated  $r$  times, such that  $I = I_s I_s I_s \dots I_s$  (e.g.,  $I_s = (i_1, i_2)$ ,  $I = (i_1, i_2, i_1, i_2, \dots, i_1, i_2)$ ), we may simply gather using the subsequence  $I_s$  (to which any of the above optimizations can also be applied), and then use the “repeat” operation, which copies memory  $r$  times such that the same result is obtained. However, it must be noted that in such situations, the use of the repeat operation is typically still suboptimal, and better optimizations, discussed later, typically exist, which avoid copying memory.
  - For future reference, let us refer to this as *dimension 1 repetition*.
- When the gather index sequence  $I$  is made from some value sequence  $I_s$  by duplicating each its individual value  $r$  times (e.g.,  $I_s = (i_1, i_2)$ ,  $I = (i_1, \dots, i_1, i_2, \dots, i_2)$ ), we may simply gather using the subsequence  $I_s$  (to which any of the above optimizations can also be applied), and then use the “interleave” operation, which copies memory  $r$  times such that the same result is obtained. However, it must be noted that in such situations, the use of the interleave operation is typically still suboptimal, and better optimizations, discussed later, are typically available, avoiding memory copying.
  - For future reference, let us refer to this as *dimension 2 repetition*.

### ■ 5.2.2 Basic Scatter Optimizations

When the index sequence  $J$  of the scatter operation  $S_J^\oplus$  can be represented via a sequence of counts, then it can be replaced by the segment CSR operation, as discussed in Chapter 3, Section 3.4.3, p. 23.

When the segmentation operation’s counts sequence  $C$  consists of the same value, i.e.  $C = (c, c, \dots, c)$  for some  $c$ , then simple reshaping followed by dense aggregation can be used. An input tensor with shape  $\mathbf{S} = (n, \dots)$  can be reshaped to  $(n/c, c, \dots)$ , and the aggregation operation  $\oplus$  (e.g., sum) can be performed on the dense tensor’s second dimension. It always holds that  $n$  is divisible by  $c$ , otherwise the segmentation



**Figure 5.7.** A visual explanation of a linear layer optimization on an example of a dimension 1-repeating left-hand side.

operation itself would be illegal in the first place.<sup>7</sup> The equivalence of the result when using such reshaping, followed by the aggregation along the 2nd dimension, follows from the fact that we use row-major indexing (Fig. 3.1, p. 11).

### 5.2.3 Basic Linear Layer Optimizations

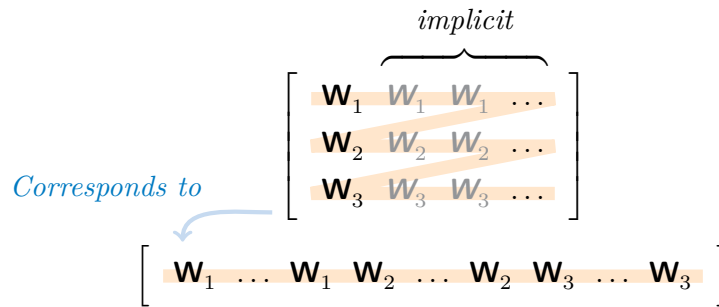
Continuing a similar pattern to the sections above, very specific layouts of the gather pairs of linear layers allow for significant simplification of the linear nodes.

Specifically, when either of the two gathers (or both) has a layout matching the last two bullet points from Section 5.2.1 (dimension 1 and dimension 2 repetition), then the full linear layer can be simplified.

When exactly one gather (either left-hand side or right-hand side) of a linear layer has the structure of dimension 1 repetition, then it is sufficient to gather the subsequence (*without repetition*) instead of the full sequence. Then, more advanced broadcasting for the matrix multiplication can be used:

Let  $n$  be the length of both of the original sequences. Let  $l$  be the length of the simplified gather's sequence. After said simplification of the gather, we have two tensors,

<sup>7</sup> Technically, a segmentation operation can operate in instances where  $\sum_i C_i < n$ , but in such case, the input tensor itself contains unused values and can be simplified in earlier layers, or can at least be sliced to  $\sum_i C_i$  along the first dimension, and then the reshaping solution becomes applicable.



**Figure 5.8.** A visual explanation of the implicit broadcasting on a dimension 2-repeating example input tensor.

of shapes  $(l, \dots)$  and  $(n, \dots)$  respectively, that we are trying to perform broadcasted matrix multiplication on. Given the fact that both PyTorch and TensorFlow use row-major ordering (Fig. 3.1, p. 11), by reshaping both tensors to  $(1, l, \dots)$  and  $(n/l, l, \dots)$  respectively, and then applying matrix multiplication, both the frameworks operate as if both tensors had shape  $(n/l, l, \dots)$ , which for the former tensor is done by re-using it as-is for all the  $n/l$  computations along the first dimension. The matrix multiplication is then applied independently across all dimensions except the final two, in which the actual matrix multiplication occurs. Afterwards, we may reshape the resulting tensor back to  $(n, \dots)$ , which gives us the same result as the original operation. This is illustrated in Figure 5.7.

As reshaping is known from Chapter 3 to be a nearly free operation computationally, we were able to simplify the computation by removing tensor value copying by using reshaping+broadcasting instead, as no explicit memory copying of the smaller input tensor is done as part of the broadcasting, as opposed to the gather+repeat equivalent.

The same is applicable in the instance of dimension 2 repetition; the only difference being that the initial reshaping must be done  $(l, 1, \dots)$  and  $(l, n/l, \dots)$  respectively, as the implicit repetition is then done along the second dimension. When later reshaped back, the output tensor is then equivalent to the original variant, where both the gathers had the length of  $n$ , or when the dimension 2 repetition was used explicitly. Again, this is the case because of the row-major ordering of tensor values. To compare how this differs from the previous example, Figure 5.8 illustrates the reshaping for dimension 2 repetition.

When both original sequences are either dimension 1 repeatable, or dimension 2 repeatable (with the same repeatability type on both sides), only one of the two can be taken advantage of via broadcasting. It makes sense to choose that which produces a shorter gather operation (i.e. greater  $n/l$ ).

When one original gather sequence is dimension 1 repeatable with  $l_1$ , and the other is dimension 2 repeatable with  $l_2$ , and for both it holds that  $l_1 \cdot l_2 = n$ , then both techniques can be applied at once, i.e., both gathers can be simplified, and the resulting tensors may be reshaped to  $(1, l_1, \dots)$  and  $(l_2, 1, \dots)$ , respectively, resulting in a tensor of shape  $(l_2, l_1, \dots)$  after the matrix multiplication, which can then be reshaped back to  $(l_1 \cdot l_2, \dots) = (n, \dots)$ , giving us the correct result. Since such broadcasting replaces memory copying for both the input gathers, this results in even greater reduction in redundant, computationally expensive memory copy operations. Note that different gather sequences may be dimension 1 or dimension 2 repeatable with different lengths of  $l$ . E.g., the sequence  $I = (1, 2, 1, 2, \dots, 1, 2)$  is dimension 1 repeatable not only with

$l = 2$ , but, e.g., also with  $l = 4$  if  $|I|$  is divisible by 4, which increases the applicability of this paired technique, as the likelihood that a pair of gathers will be pairable as such is increased by this. Lastly, if a gather index sequence consists of the repeated use of a single value, i.e.,  $l = 1$ , then it is both dimension 1 and dimension 2 repeatable, allowing it to be utilized with an opposing side of either type of repeatability, in the exact same way, as it is reshaped to  $(1, 1, \dots)$  in either case.

#### ■ 5.2.4 Advanced Linear Layer Optimizations: Reordering and Padding

The initial conditions needed to be able to apply optimizations from Section 5.2.3 are quite rare. Nonetheless, it would be beneficial if we could use the broadcasting techniques as often as possible, in order to minimize the use of unnecessary value duplications in gathers, or the total number of gathers, even.

Ideally, we would like to be able to utilize similar optimizations in *every* instance of at least one of the two gathers containing a low number of unique values to be repeated.

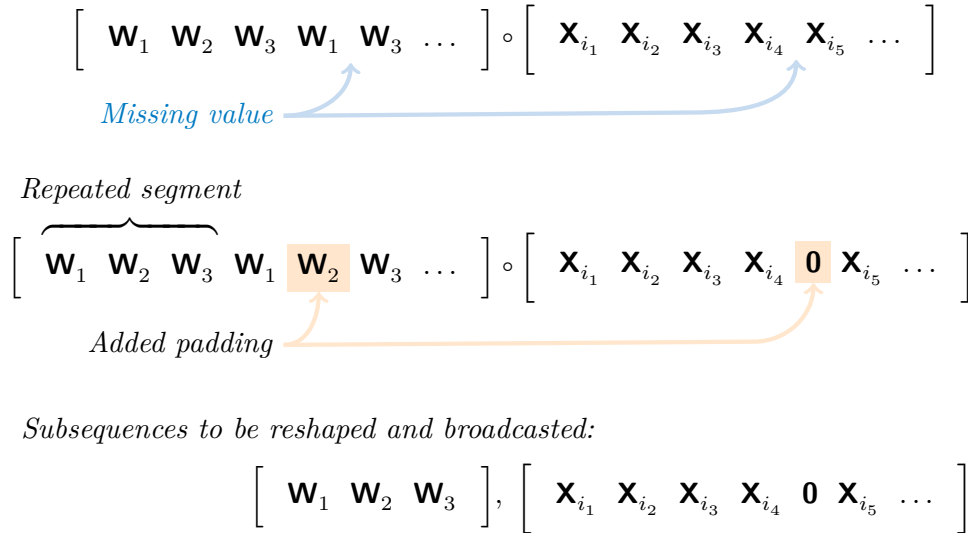
Thankfully, this is achievable.

Firstly, we may reorder the computations (i.e., the individual values) within the linear layer such that the gather operation for at least one side (the side with the small number of unique values specifically) is dimension 1 or dimension 2-repeatable. Such reordering is possible, since it only affects the output value order of the layer. It may initially seem that this will require an additional gather operation at the end of the layer; however, instead, we may simply update any subsequent layers' input gathers accordingly, so that the reordering is reflected in them. This means that an additional gather operation is typically not needed as a result of this.

Without loss of generality, let us assume that the left-hand side is the side that we want to make dimension 1 or dimension 2-repeatable. Often, the indices cannot be reordered as such due to the fact that some values from the left-hand side are used more times than others, making such reordering impossible. To solve this, we may add clever padding to the linear layer, i.e., we may add extra (pairs of) values to be matrix-multiplied, that are never used by any subsequent layers. This may be worth doing as long as the increase in the tensor sizes as a result of the padding is lower than the *decrease* in the tensor sizes as a result of the removal of the memory copy operation(s). This is illustrated in Figure 5.9. (Note that this also requires updating the input gathers of any subsequent layers accordingly, as the individual indices of this layer's output values shift as a result of this.)

In some cases, it is beneficial to add padding such that each unique value from the left-hand side is multiplied with each unique value from the right-hand side. Then, there exists a value pair ordering (specifically, the lexicographical ordering) such that one side becomes dimension 2-repeatable, and the opposite side becomes dimension 1-repeatable. This allows for the greatest memory copy reduction. However, this may often require too much padding, making the result computationally more expensive. Therefore, in some cases, it may be beneficial to only do it for one side, which requires less padding. Therefore, we must consider all three alternatives (both sides together, or either of the two sides), as well as a fourth alternative where no padding is applied at all.

To decide which of the four alternatives is the best one, we may simply measure the resulting lengths of the two gather operations (where if a gather operation is either



**Figure 5.9.** A visual explanation of the padding optimization in linear layers. Example of adding padding such that the left-hand side becomes dimension 1-repeatable.

dimension 1 or dimension 2-repeatable, we only count the length of the subsequence, since the explicit repeating will not be performed), and choose the variant where their sum is minimal.<sup>8</sup>

### ■ 5.2.5 Advanced Linear Layer Optimizations: Advanced Reordering/Padding

The padding variants from Section 5.2.4 are in fact not sufficient, and there are many more padding variants worth considering.

For example, let us consider a linear layer that uses two weights,  $\mathbf{W}_1$  and  $\mathbf{W}_2$ , in an arbitrary order, where  $\mathbf{W}_1$  is used *roughly* twice as often as  $\mathbf{W}_2$ . Then, the padding needed to achieve the weight gather subsequence  $(\mathbf{W}_1, \mathbf{W}_2)$  is significantly greater than the padding needed to achieve  $(\mathbf{W}_1, \mathbf{W}_1, \mathbf{W}_2)$ . Given the setting, the latter requires minimal padding, or even zero padding if  $\mathbf{W}_1$  is used *exactly* twice as much as  $\mathbf{W}_2$ .

What is more, an additional benefit of this is that if the resulting subsequence is sufficiently short and the preceding concatenate operation only concatenates single-value inputs (e.g., weights), we may remove the corresponding gather operation entirely, and simply concatenate the inputs. For example, concatenating  $(\mathbf{W}_1, \mathbf{W}_1, \mathbf{W}_2)$  directly (where the order of concatenation is typically determined on the CPU) is likely cheaper than the concatenation of  $(\mathbf{W}_1, \mathbf{W}_2)$  followed by a gather operation with index sequence  $I = (1, 1, 2)$  performed on, e.g., the GPU. This is purely because either of the two sequences to concatenate is very short, and the added overhead of performing the additional gather operation on different hardware is likely too big for such a short index sequence.

<sup>8</sup> This may not be the optimal decision in terms of the resulting performance, but it is likely a satisfactory proxy. Finding the actual optimal solution would require us to compare the total number of gather operations in the full computational graphs (including other layers) for all four variants, but this leads to an exponentially larger number of variants to test across all linear layers of the computational graph, and is thus infeasible.

To find the subsequence likely to produce the least amount of padding, we may do the following:

1. Find all unique values in the index sequence. Let the following be their identifiers:  
 $U = (I_1, \dots, I_{|U|})$ .
  - Let  $(n_{I_1}, \dots, n_{I_{|U|}})$  be the total counts of the uses of individual unique values in the index sequence.
2. Find the value  $n^{\min} := \min_{i \in \{I_1, \dots, I_{|U|}\}} n_i$  that is used the lowest number of times in the index sequence.
3. For all the unique values, compute how much more often they roughly appear in the index sequence:  $t_I := \lfloor \frac{n_I}{n^{\min}} + \frac{1}{2} \rfloor$
4. The subsequence is one in which each  $I_i$  is used  $t_{I_i}$  times, as long as its length does not exceed the predetermined upper limit.

All of the padding methods from Section 5.2.4 then can be used with the subsequence(s) found by the above method.

### ■ 5.2.6 Deduplication: Downward Propagation of Gathers

The situations which perform reordering of individual values in a vectorized node, and as a result require reindexing of gathers in subsequent layers, is what we refer to as *downward propagation of gathers*, as it in some instances may require the introduction of additional gather operations. We will discuss in a moment when additional gathers are needed, and when not. The reordering and padding discussed in Sections 5.2.4 and 5.2.5 is one example of the downward propagation of gathers.

A simple optimization that should precede any optimizations of linear layers discussed earlier (such as the introduction of padding) is the simple *removal* of unused values, as well as deduplication of values. The removal of unused values refers to the removal of any values which are not referred to by any subsequent layers' gathers. Deduplication refers to the removal of values that are redundant based on the fact that the same computation is already being performed in the layer at a different location.

The simpler version of deduplication simply checks for duplicate indices in input gathers, and ensures that gathers consist only of “unique” values. For transformation layers, this is trivial. Linear layers must keep matching *pairs* of indices in its two gather operations unique. Aggregation layers must keep matching *groups* of indices, based on how the inputs are grouped by its scatter operation. For fact layers this is done also (not for weights, though, as they are learnable), simply by comparing values. This ensures that every layer performs each computation exactly once and does not perform redundant computations.

It may seem counterintuitive that we first perform the removal of unused values and deduplication, to then do the exact opposite in later optimizations, such as by introducing padding in linear layers. Nonetheless, we must account for suboptimal computational graphs which perform redundant computations, and simplify them first, so that the subsequent optimizations truly operate properly.

The more complex deduplication method is *isomorphic compression* [87], discussed previously in the NeuraLogic chapter in Section 4.3.2 (p. 32). This method performs multiple forward passes of the full network, each with different weight value initializations. Any groups of values that are found to be consistently matching for all forward passes, are deemed “duplicate.” This form of compression is guaranteed to be lossless. The Optimizer reimplements the isomorphic compression algorithm to remove duplicate

values, but it does so independently for each vectorized node (i.e., only values within each vectorized node are compared, never across different layers), so as not to disrupt the computational graph structure.<sup>9</sup>

As discussed earlier, all of these methods must be reflected in subsequent layer gathers. The situations in which this may result in additional gather operations are twofold:

Firstly, if the layer in which this type of optimization is performed is the last layer, a final gather operation must be added at the end to obtain the output in expected order.

Secondly, if one of the subsequent layers using these values is an aggregation layer, it is likely that it uses at least some input values multiple times. It may be the case that if such form of deduplication had not been used, the aggregation layer would not have required a gather operation, as the necessary duplicates would have already existed in the prior layer's output. Therefore, by removing duplicates in the earlier layers, the aggregation layer must begin with a gather operation, in order to recover the proper segments to be aggregated.

Nonetheless, since it cannot be presumed that the computational graph is already in its optimal form on input, we first perform all the optimizations that deduplicate and simplify individual layers, i.e., all the optimizations that propagate gathers downwards, and then we perform upward propagation, to introduce (or reintroduce) any duplicities that help avoid redundant gather operations.

Lastly, it is worth noting that in some instances, it is not an easy task to determine prior to execution whether an additional gather operation is beneficial for performance or not. For example, for linear layers immediately followed by aggregation layers, it may be beneficial to simplify (deduplicate) the linear layer, and introduce an additional gather operation to the aggregation layer; however, the contrary may sometimes also be better for performance, i.e., to keep duplicate computations in the linear layer, and avoid an unnecessary additional gather operation.

### ■ 5.2.7 Upward Propagation of Gathers

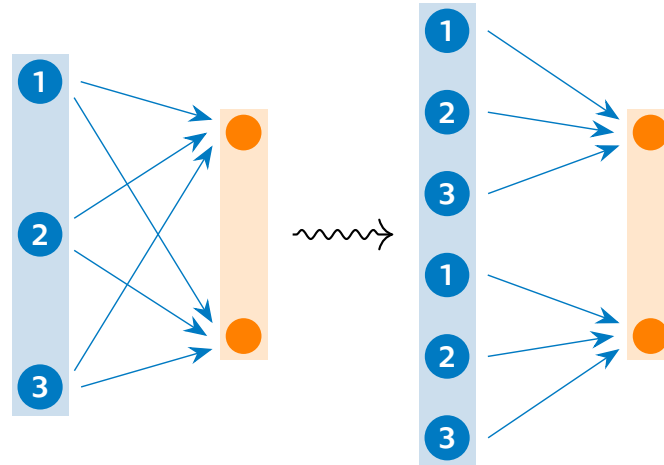
*Upward propagation of gathers* refers to the situation where we remove a gather operation from a layer's input by reordering/duplicating the values computed in the preceding layer. It must be noted that for simplicity, we only do so if the layer being modified is the only layer that uses the preceding layer on input (as the result would introduce – or reintroduce – gather operations into all the other layers that have the layer on input). Furthermore, we only do so if the vectorized node being modified has only a single vectorized node on its input (in other words, if the layer being modified has no Concat operation, as nearly all instances of Concat must be followed by a gather operation).

One motivation for upward propagation of gathers has already been discussed in Section 5.2.6. However, a perhaps even more motivating example is one where the upward propagation is taken to the extreme: Let us consider a computational graph where the structure of its vectorized variant is a simple sequence of layers.<sup>10</sup> If we perform upward propagation of gathers for every single layer (i.e., in the reverse topological ordering of

<sup>9</sup> I mention this because NeuraLogic has its own implementation of isomorphic compression, which compares individual neurons across the whole computational graph. However, as a result of this fact, when using the Compiler with NeuraLogic, it is recommended to *disable NeuraLogic's isomorphic compression algorithm*, as it may disrupt the network structure and, e.g., introduce unnecessary skip connections, which are then unresolvable by the Vectorizer. Therefore, the layerwise (vectorized) compression implementation, as part of the Optimizer, should be used instead.

<sup>10</sup> For example, many typical instances of graph neural networks have this structure.





**Figure 5.10.** Illustration of how the computational graph may expand into the form of a forest via the upward gather propagation optimization. The yellow layer is an aggregation layer. Note that as a result of the operation, the individual values are already in the expected order for the second layer, and so a gather operation is no longer needed.

the layers), irrespective of the amount of the (transitive) value duplication it causes, we obtain an equivalent computational graph *without any gather operations*. In other words, the non-vectorized variant of such computational graph is a forest (i.e. for each output node, its corresponding connected component is a tree, where if we flip the orientation of every edge, the output node becomes the root of the directed tree). What this may look like in practice is illustrated in Figure 5.10. What is even more interesting is that even though it may seem that the transitive upward node duplication will result in an exponential increase in the total number of nodes, *this is oftentimes not the case*, and what we will see later is that the resulting computational graph may sometimes in fact perform *the fastest*, due to the increase in memory usage being negligible in terms of performance compared to the reduction in the total number of vectorized operations performed.

Nevertheless, *some* degree of upward propagation of gathers is typically useful in any instance. The least aggressive configuration propagates gathers upwards only if such propagation does not increase the total number of values in the preceding vectorized operations at all. This is safe to do, as it may only lead to removals of gather operations, without any other side effects. However, it is typically useful to relax the strictness to further degree, as when the node duplication rate is sufficiently low, the increase in the total number of values is negligible compared to the reduction in the total number of (gather) operations performed.

What is also important to understand is that the upward propagation of gathers can start at any vectorized node in the computational graph, skip any gather along the way, and resume right after. This means that even for long chains of vectorized nodes, a well configured upward propagation optimization can, e.g., keep a single gather operation in the middle of a chain, and remove all gather operations that precede it, as well as all that follow it, simply because the single gather operation is found to be the only one worth keeping in the graph in terms of its effect on reducing the size of the full computational graph.

The implementation of the optimization in the Compiler offers multiple hyperparameters to configure the degree/strength to which it operates.

It must already be obvious to the reader that this optimization is able to propagate a gather not only *because* of an aggregation layer, but also *through* another (preceding) aggregation layer. In fact, the optimization is able to propagate a single gather through the full computational graph, if necessary. For aggregation layers, the reordering/duplication of its outputs requires the reordering/duplication of the whole *groups* of its inputs.<sup>11</sup>

### ■ 5.2.8 Unit Fact Processing

One important processing step worth mentioning is that the input data representation supports “unit” facts, which can be used either in place of weights, or in place of data inputs. They are understood as “tensors of arbitrary shapes, which, when matrix-multiplied with another tensor, simply act as the identity function for the opposite tensor.” Since this is only an abstract concept, processing steps must resolve this to actual operations. This is because matrix multiplication with a unit fact may not be removable always, and sometimes must actually be performed. For example, the same unit fact may be used as weight in two separate linear layers, alongside other weight values, where the weight values in the first layer have a different shape than the weight values in the second layer. To properly support this, the Optimizer must do two things:

- A) For all linear layers, where the gather operations interleave unit values with actual values, produce unit value tensors of appropriate shapes, to support the operation. The required form of a unit fact to properly correspond to the identity operation in linear layers may either be a diagonal matrix, or a vector of ones.<sup>12</sup>
- B) Simplify any linear layer where either of its sides consists *exclusively* of unit facts, by simply replacing it with the gather operation of the opposing side instead, skipping the matrix multiplication operation itself. This is needed for two reasons:
  - 1) The linear operation is completely redundant in such instances, and the identity matrix multiplication can thus be removed entirely, sparing computational resources.
  - 2) Step A) is impossible to do for computational graphs where there exists a linear layer with *both* sides consisting exclusively of unit facts, since the required shape of such layer cannot be inferred, and the unit fact tensors thus cannot be materialized.

### ■ 5.2.9 Further Optimizations for More Complex Architectures

In vectorized computational graphs with a lot of branching, it may be beneficial to put a greater focus on layers which perform the concatenation of multiple inputs:<sup>13</sup>

Firstly, if possible, the individual values in the layer should be reordered such that the initial gather operation’s index sequence  $I$  can be represented as the concatenation of index sequences  $I = I_1, \dots, I_n$ , where  $I_i$  only accesses values from the  $i$ -th input. In aggregation operations, such reordering can be done even if values are aggregated across inputs, either by using scatter operations instead of segmentation operations, or, in the instances where aggregation is done using reshaping and aggregation along a dimension, by using aggregation along dimension 1 instead of dimension 2.

<sup>11</sup> This is nontrivial, and so it deserves an explicit mention.

<sup>12</sup> Why do we need unit values in linear layers? Why cannot we simply skip the matrix multiplication for corresponding indices entirely? We can; however, in some instances, it is beneficial to vectorize nodes that perform matrix multiplication with nodes that do not. This is because it may prevent unnecessary skip connections, which mainly prevents an extra concatenation operation in subsequent layers, at the expense of the linear layer being wider.

<sup>13</sup> Almost none of this is implemented at the time of writing.

Secondly, as long as the first step is possible to do, the  $n$  individual gather operations can then be performed first, and the result then can be aggregated. The benefit of this is twofold: The individual gather operations can potentially be upwards-propagated through, or at least simplified. This is because for many such concatenate + gather operations, the original gather index sequence  $I$  may not be optimizable using methods from Section 5.2.1, whereas the individual index sequences  $I_i$  may be. It is worth studying when exactly it is beneficial to perform multiple separate (simpler) gather operations as opposed to running a single, more complex operation. It is worth noting that this will likely differ depending on the hardware on which the computational graphs are to be executed. Nonetheless, when upward propagation of gathers is enabled to the strongest degree, using this approach will allow to propagate even through such layers, dropping their gather operations as a result entirely.<sup>14</sup>

### ■ 5.2.10 The Full Implementation

Additional operations are performed along the whole process, such as the removal of unused layers (operating iteratively in reverse topological order of vectorized nodes), as well as various conversions between intermediate data representations, depending on the individual needs of the various optimizations, for the sake the implementation being simpler.

For the sake of the completeness of this chapter, the next few pages explain in detail the full optimization pipeline, explaining the Optimizer's current implementation, mainly the order in which the individual optimizations are performed. The full optimization pipeline performs the optimizations in the following order:

1. Unit facts are merged into a single initial fact layer of a single value. All references to the unit fact across all layers are replaced with this single reference.
2. Fact layers are deduplicated.
3. Individual layer counts are computed for future reference.
4. Layers are condensed. Since the layer representation combines optional matrix multiplication with subsequent optional aggregation and transformation, then if there are layers that can be condensed, they are condensed (e.g., matrix multiplication + identity + identity, and identity + aggregation + transformation are merged into a single layer of matrix multiplication + aggregation + transformation, if there is no branching between the two former layers).
5. Individual layer shapes are computed for future reference.
6. Transposition of layers containing dimension 2 aggregations is done, so that they become dimension 1 aggregations (refer back to Section 5.2.9.)
7. Linear layers that are found to be gathering values of incompatible shapes are split into separate layers, and concatenated back together after their respective matrix multiplications are performed.<sup>15</sup>
8. Individual layer counts are recomputed for future reference.
9. Linear layers consisting exclusively of unit facts are simplified (refer back to Section 5.2.8).

<sup>14</sup> As discussed in Section 5.2.7, upward propagation of gathers is currently only implemented such that it only propagates through vectorized nodes that have no more than one vectorized node on input.

<sup>15</sup> As discussed at the very beginning of this chapter, the lines between the Vectorizer and the Optimizer are slightly blurry in the actual implementation. This is one of those instances, as this is clearly a problem pertaining to the Vectorizer. For the purposes of this text, this can be considered an implementation detail.

10. Isomorphic compression is performed (refer back to Section 5.2.6). This requires the conversion of the current intermediate representation of the vectorized computational graph into an executable one, so that the forward pass can be computed using the appropriate backend. A simplified pipeline (without optimizations) is used for this, so that the isomorphic compression can conclude before other optimizations follow. The simplified pipeline is explained later. After the isomorphic compression is performed, additional required updates are done:
  - (i) All multi-input gather operations are updated to properly reflect the changes made by isomorphic compression in their input layers.
  - (ii) Individual layer counts are recomputed for future reference.
11. Multiple downward gather propagation optimizations are performed, for each layer, in the following order:
  - (i) Multi-input gather operations are updated to reflect any changes made by this optimization in preceding layers.
  - (ii) Index-based deduplication is done in linear layers (refer back to Section 5.2.6). An additional gather operation is added temporarily after each updated linear layer, unless there already is a gather operation immediately after, in which case it is updated with the results of this optimization.
  - (iii) Padding for linear layers is done, to be simplified later (refer back to Sections 5.2.3, 5.2.4, and 5.2.5). An additional gather operation is added temporarily after each updated linear layer, unless there already is a gather operation immediately after, in which case it is updated with the results of this optimization.
  - (iv) Index-based deduplication is done in non-linear layers, together with the downward propagation of the intermediate gather operations, if possible. Downward propagation of these gather operations may not be possible because they often immediately precede aggregation operations, which may require the gathers to duplicate some values. If the additional gather operations are not desirable, upward propagation optimization will remove them in a moment. Refer to Sections 5.2.6 and 5.2.7 for more details.
  - (v) Layer counts are recomputed for future reference.
12. Layer representations, where a linear operation, itself always beginning with a gather operation, is followed by another gather operation prior to the optional aggregation + transformation, are simplified. This is done by splitting such complex layers into chains of simpler layers: The linear layer by itself (with no aggregation nor transformation), followed by a layer with the second gather operation, and the optional aggregation + transformation from the original layer. This is merely a change in the intermediate data representation, so that the subsequent optimizations are simpler.
13. The multi-input gather representations are converted to concatenation + gather representations. Please refer to Section 5.1.4 for more details. The decision to do this now is because the preceding optimizations are simpler with the former representation, whereas the optimizations that follow are simpler with the latter.
14. The basic linear layer optimizations (refer to Section 5.2.3) are performed, utilizing the padding made in preceding optimizations.
15. Upward propagation of gathers is performed (refer to Section 5.2.7), after which the layer counts are recomputed for future reference. This optimization is sometimes optionally performed before the basic linear layer optimizations, as the actual implementation of upward gather propagation does not support propagating through linear layers with both sides utilizing broadcasting. When this optimization runs after the basic linear layer optimization, it propagates through single-side broadcasted linear

layers such that it honors the broadcasting. When this optimization runs before, it disrupts the padding.

16. Remaining gather operations are simplified in line with Section 5.2.1. Any concatenate + gather operations that only concatenate single-value inputs (e.g., weights) are simplified to just concatenate, as long as the number of inputs is sufficiently low. This is discussed in greater detail in Section 5.2.5.
17. Single-input layers where all operations are identity operations, are removed. Any references to such layers are updated accordingly, to instead refer to preceding layers.
18. Unit facts are materialized (refer to Section 5.2.8).
19. If any facts or weights are only ever referenced together, using the same consecutive order of concatenation, then the facts/weights are pre-concatenated together, to remove the need for the concatenation operations. Note that learnable weights must never be pre-concatenated with non-learnable facts. What is more, note that a learnable weight must never be concatenated with itself (i.e., the concatenation sequences affected by this optimization must never contain duplicates of learnable weights), as it would lead to learnable parameter duplication.
20. Unused layers (i.e., layers that are not the output of the computational graphs, nor are the input of any subsequent layers) are removed. (This is done in reverse topological order, so that transitively unused layers are also removed.)
21. The identifiers of individual facts, weights, and other layers are prepended with the respective prefixes (`f_`, `w_`, and `l_`), so that references to them can from this point onward co-exist without any additional type identifiers.
22. The layer representation of the computational graph is converted to a representation where individual nodes are mere sequences of vectorized operations. In other words, the strict representation of what a “layer” is, is relaxed.
23. Sequences of nodes (without branching) are merged into single operation sequences, so that the total number of nodes is minimized.
24. Any pairs of reshape operations found in immediate succession are modified so that only the final reshape operation remains, as the preceding ones are redundant.
25. The vectorized network is now finished.

For the sake of completeness, let us discuss the simplified tail end of the pipeline for the isomorphic compression (i.e. to obtain an executable computational graph from the representation found in step 10 above, using as few steps as possible):

1. Individual layer counts are computed for future reference.
2. Individual layer shapes are computed for future reference.
3. The multi-input gather representations are converted to concatenate + gather representations.
4. Unit facts are materialized.
5. Unused layers are removed.
6. The identifiers of individual facts, weights, and other layers are prepended with the corresponding `f_`, `w_`, and `l_` prefixes.
7. The layer representation is relaxed to the operation sequence representation.
8. This is now an executable computational graph.

# Chapter 6

## Results

Let us show that the Compiler is able to optimize the performance of computational graphs significantly. We will be comparing the performance of graph neural networks against the baseline of PyTorch Geometric on different hardware. We will also measure the performance of more complex relational networks; however, the only baseline we will be using in such case is the (not vectorized) CPU implementation of NeuraLogic, as PyTorch Geometric does not support such networks.

### 6.1 Datasets

We will be using the following datasets:

- “Mutagenesis”: A dataset of 188 molecules, each having 17.9 nodes and 39.6 edges on average. 3371 nodes and 7442 edges in total. Classification task into 2 classes.
  - The version we will be using to compare against PyTorch Geometric has 7 features for each node. It has been sourced by the TU Dortmund University [89] and is in this form available directly on the PyTorch Geometric website [90].
  - The version we will be using in more advanced (non-GNN) examples has a node type of 8 possible values, and 6 different bond strengths. It is available in the form of a *relational database* at [91].
- “Enzymes”: A dataset of 600 molecules (746 connected components, though), each connected component having 26.2 nodes and 100.0 edges on average. 3 features for each node, 19 580 nodes and 74 564 edges in total. Available from [89] and the PyTorch Geometric website [90].
- “Proteins”: A dataset of 1113 molecules, each having 36.2 nodes and 135.1 edges on average. 3 features for each node, 34 471 nodes and 162 088 edges in total. Available from [89] and the PyTorch Geometric website [90].

For everything discussed in this chapter, each dataset will be fully batched, unless mentioned otherwise.

### 6.2 Graph Neural Networks

Since graph neural networks give us a good baseline to compare the Compiler against, let us begin with them. We will be comparing the performance of example GNNs built in NeuraLogic, vectorized using the Compiler from Chapter 5, against equivalent GNNs built with PyTorch Geometric.

#### 6.2.1 Performance

Let us measure the performance of various GNNs. The compared GNN architectures are two-layer GCN and GraphSAGE, on all three datasets mentioned at the beginning of this chapter. Figure 6.1 (p. 60) shows the performance comparison of the forward passes

of the Compiler-produced (optimized) output against PyTorch Geometric, executed on an Nvidia Tesla V100 GPU. Figure 6.2 (p. 61) is a zoomed-in view of the same.

The Compiler is parameterized only in terms of the strength of the upward gather propagation optimization (Section 5.2.7). All other optimizations are enabled and configured to their default settings. The “strength” is only an approximate representation of the actual configuration of the upward gather propagation optimization, as the actual parametrization consists of multiple different hyperparameters, which together loosely translate to the strength with which the optimization pushes gathers upwards. “Minimal strength” performs only propagation that does not duplicate any values (i.e., propagates only gathers that merely reorder values, and are as such truly redundant). “Maximal strength,” on the other hand, propagates (removes) *all* gathers, as explained in Chapter 5, or as shown by example later in Section 6.2.3. There are other variants in between; 18 in total. All 18 variants are available to the users of the Compiler as presets.

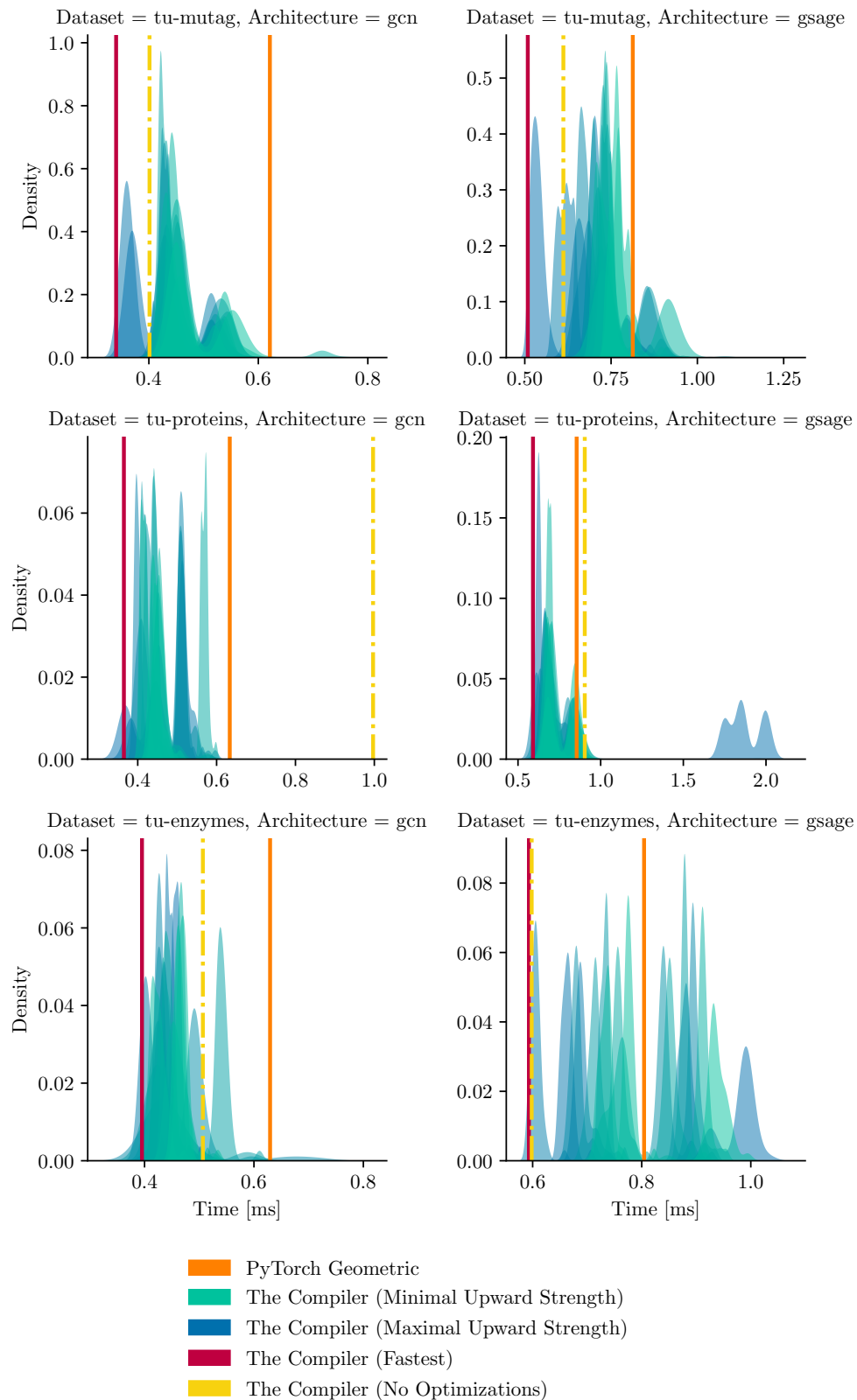
Unoptimized Compiler output (i.e., with all optimizations disabled, corresponding to the immediate output of the Vectorizer) is also added for comparison.

*As you can see, the Compiler beats PyTorch Geometric across all architectures and datasets of all sizes by a significant amount, especially on the GPU, sometimes even with its unoptimized variant.* This is likely because the Compiler is able to use segment CSR operations instead of scatter operations immediately, even without applying other optimizations. Nonetheless, the optimizations improve the results even further.

In each instance, at least half of the 18 configurations surpasses PyTorch Geometric in terms of performance. For the GraphSAGE architecture on the Mutagenesis dataset, the *best performing* configuration is the one with *the most aggressive* upward propagation of gathers, i.e., the computational graph with the lowest number of gather operations possible (zero), at the expense of the largest memory utilization. Conversely, for the Proteins dataset, the largest of the datasets, this configuration performs *the worst*, as the memory utilization is simply too high as a result. This is because for the Proteins dataset, the unrestricted upward gather propagation results in 948 017 values on the input, whereas the computational graph resulting from the 2nd strongest upward propagation configuration has only 2 additional gather operations in total, and the largest number of values (i.e., the size of the first tensor dimension) that it has in a single tensor is 240 271, corresponding to a significantly lower memory utilization.

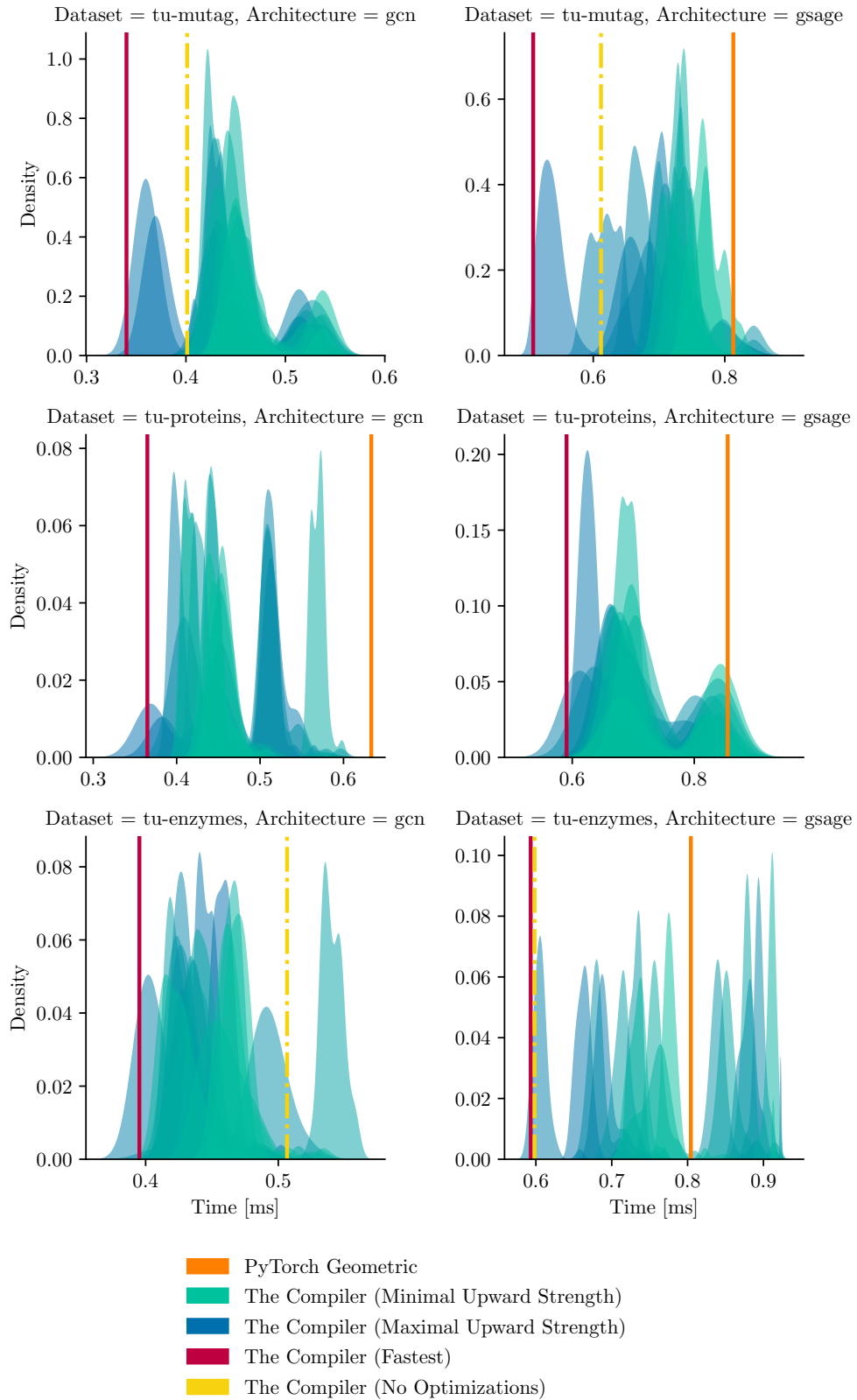
Figure 6.3 (p. 62) compares the best performing Compiler configuration with PyTorch Geometric, for multiple consecutive iterations in each instance. As you can see, the first few iterations are typically outliers, which is due to the GPU typically performing additional operations during the first few iterations, such as freeing old memory.

Ultimately, as expected, it seems that the best configuration of this optimization depends on the dataset size and architecture complexity. Based on this, performance tuning requires finding the sweet spot for this optimization. This can be done, e.g., by utilizing the intuition from Section 6.2.3 for finding the best configuration, or simply by trying all of the 18 presets prior to the actual training. This should not be difficult in any case because the Compiler typically does not require tuning for any other of its optimizations.

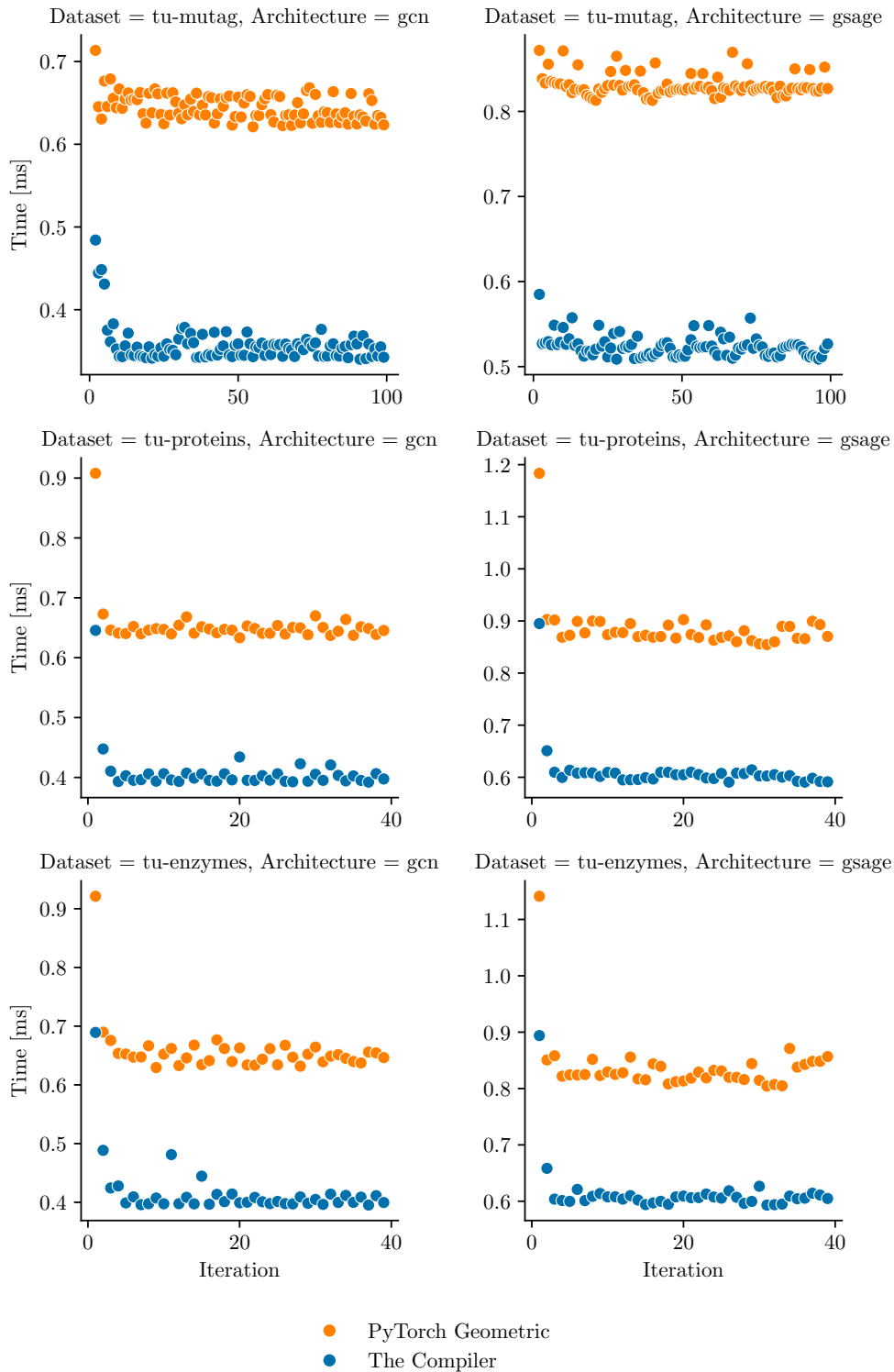


**Figure 6.1.** Forward pass performance comparison of GNN architectures on a GPU.





**Figure 6.2.** Forward pass performance comparison of GNN architectures on a GPU, close-up view (with outliers excluded).



**Figure 6.3.** Forward pass performance comparison of GNN architectures on a GPU, comparing the best performing Compiler configurations with the baseline, for multiple consecutive iterations.

### ■ 6.2.2 CPU Performance

Figures 6.4 (p. 64) and 6.5 (p. 65) show the same forward pass comparison on the CPU, where the NeuraLogic CPU implementation in Java is added as another baseline. It must be noted that since the NeuraLogic Java implementation is fully sequential, it does not utilize batching.

Here, the Compiler-produced computational graphs still have the best performance. Figure 6.6 (p. 66) compares the best performing Compiler configuration with the two baselines for multiple consecutive iterations.

However, as opposed to GPUs, there is a clear performance benefit in keeping the upward propagation strength *low* for CPUs. This is obvious, as the CPU operates more-less sequentially (unlike, e.g., the GPU), so the less values to process, the better. Furthermore, CPUs perform operations requiring random memory access very well, and as such, having more gather operations barely makes a difference in and of itself in terms of performance.

Lastly, both the Compiler and PyTorch Geometric perform better than the NeuraLogic CPU backend, even on the CPU. This is because CPUs are able to perform vectorized operations to a small degree, and both the former solutions utilize the PyTorch CPU backend, which is implemented in a low-level language, and is able to vectorize individual operations to the degree that CPUs support. NeuraLogic’s original CPU backend, however, is implemented in Java in a way that likely does not utilize CPU vectorization at all.

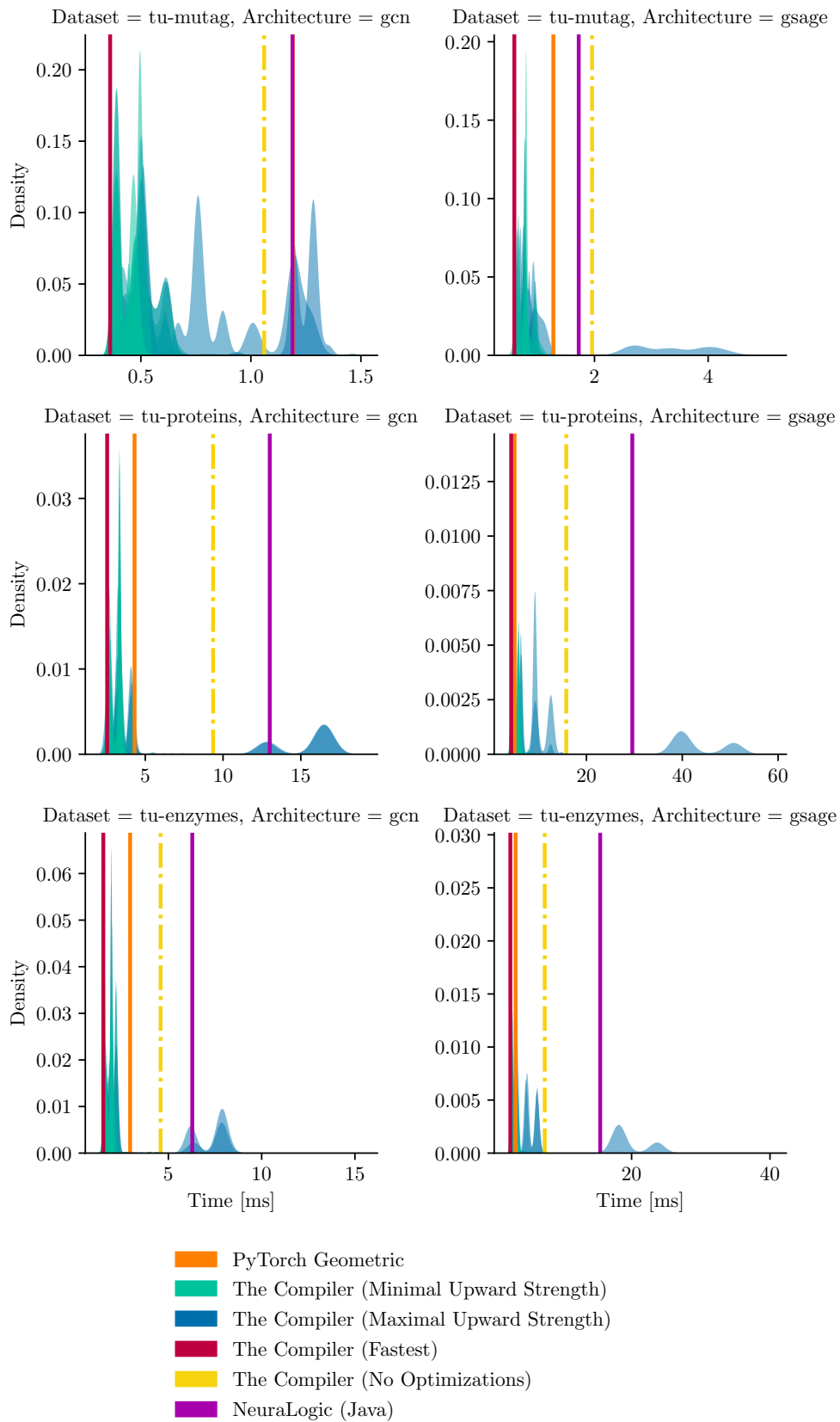
### ■ 6.2.3 Computational Graphs – GCN Example

Let us now take a momentary break from direct performance comparisons, to compare select computational graphs themselves, in order to better understand the effect of the Optimizer, especially that of the upward optimization of gathers. We will be looking at a very simple GNN architecture, which is a two-layer GCN convolution without normalization, on the Mutagenesis dataset. We will be using Python-like pseudo-code to represent the (vectorized) computational graphs of the forward passes of the respective solutions.

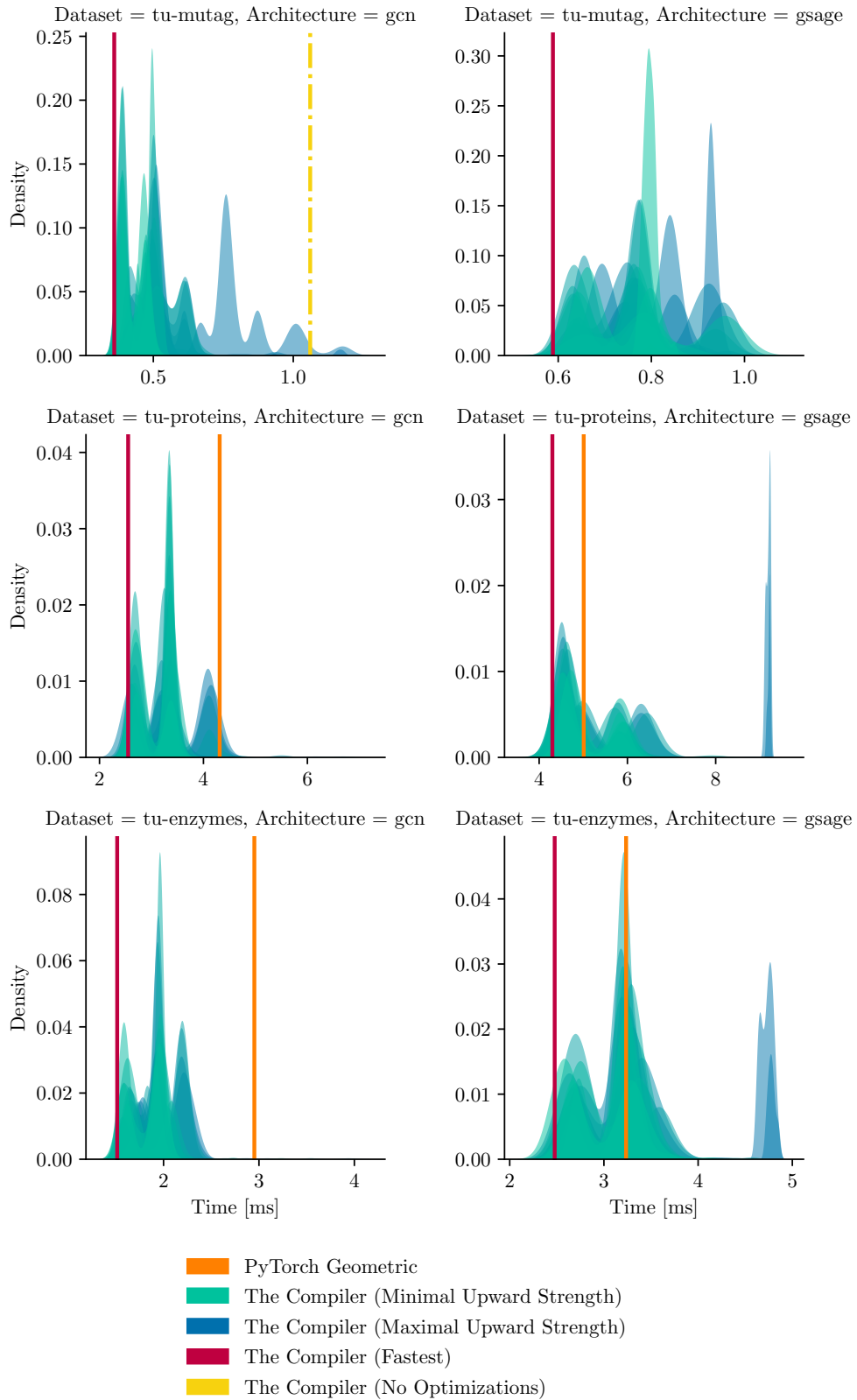
Figure 6.7 (p. 67) shows the pseudo-code equivalent to the PyTorch Geometric forward pass, and Figure 6.8 (p. 68) shows the pseudo-code equivalent to the *unoptimized*, vectorized NeuraLogic computational graph (i.e., the output of the Compiler on NeuraLogic input, with all optimizations disabled).

As you can see, the computational graphs are not too different. There are three main differences:

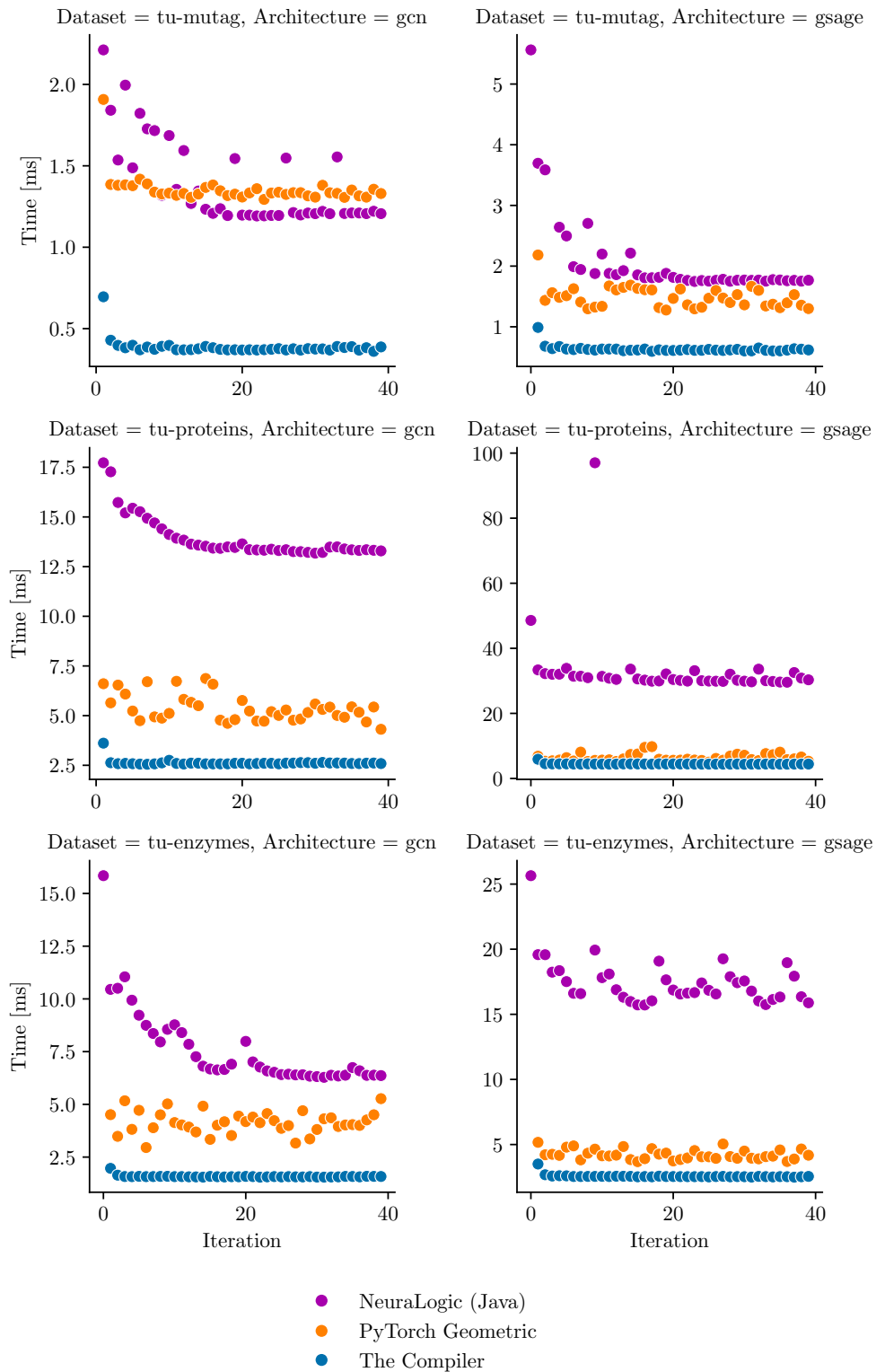
1. The unoptimized, general representation of a Compiler-produced graph contains unnecessary gathers for weights. In this example, consisting of GCN convolutions, all the weights are single-valued, and so each corresponding gather operation merely repeats each weight 7442 times. The broadcasting optimization (Section 5.2.3) will solve this once enabled. The PyTorch Geometric computational graph, on the other hand, is essentially hand-crafted as-is for the GCN convolution, and as a result does not contain this redundancy.
2. The PyTorch Geometric example performs linear operations first, as the GCN convolution is defined like this, and its computational graph is hand-crafted for the GCN convolution specifically. This is not the case in the Compiler-produced result, which was built without any such domain knowledge.



**Figure 6.4.** Forward pass performance comparison of GNN architectures on a CPU.



**Figure 6.5.** Forward pass performance comparison of GNN architectures on a CPU, close-up view (with outliers excluded).



**Figure 6.6.** Forward pass performance comparison of GNN architectures on a CPU, comparing the best-performing Compiler configurations with the baselines, for multiple consecutive iterations.

```

def forward(x): # x is a tensor of shape (3371, 7)
    x = linear(w1, x)
    x = gather(x)      # 3371 -> 7442
    x = scatter(x)     # 7442 -> 3371
    x = relu(x)
    x = linear(w2, x)
    x = gather(x)      # 3371 -> 7442
    x = scatter(x)     # 7442 -> 3371
    x = scatter(x)     # 3371 -> 188
    x = linear(w3, x)
    x = sigmoid(x)
    return x

```

**Figure 6.7.** PyTorch Geometric forward pass computational graph for an example GNN with two GCN convolutions

```

def forward(x): # x is a tensor of shape (3371, 7)
    x = gather(x)      # 3371 -> 7442
    w1_f = gather(w1)  # 1 -> 7442
    x = linear(w1_f, x)
    x = segment_csr(x) # 7442 -> 3371
    x = relu(x)
    x = gather(x)      # 3371 -> 7442
    w2_f = gather(w2)  # 1 -> 7442
    x = linear(w2_f, x)
    x = segment_csr(x) # 7442 -> 3371
    x = segment_csr(x) # 3371 -> 188
    w3_f = gather(w3)  # 1 -> 188
    x = linear(w3_f, x)
    x = sigmoid(x)
    return x

```

**Figure 6.8.** Unoptimized Vectorized NeuraLogic forward pass computational graph for an example GNN with two GCN convolutions

3. The Compiler was able to use the computationally less expensive segment CSR operations instead of scatter operations immediately, even with all optimizations disabled. This is done at the Vectorizer stage already, as every (unoptimized) aggregation layer in this stage is defined as a sequence of a gather operation immediately followed by a scatter operation (done for the sake of the computational graph following a strict scheme before optimizations are applied; for details see Chapter 5). For this reason, any scatter operation can be a segment CSR operation, as its preceding gather operation can ensure the necessary order of the individual elements.<sup>1</sup>

<sup>1</sup> Given that all optimizations are disabled, there should be additional gather operations before each of the two final segment CSR operations, to match the unoptimized vectorized graph definition from Chapter 5. In this case, however, the gather operations are both exactly equivalent to the identity functions, and so

```

def forward(x): # x is a tensor of shape (7, 7)
    x = linear(w1, x)
    x = gather(x)      # 7 -> 67
    x = segment_csr(x) # 67 -> 25
    x = relu(x)
    x = linear(w2, x)
    x = gather(x)      # 25 -> 524
    x = segment_csr(x) # 524 -> 207
    x = gather(x)      # 207 -> 3371
    x = segment_csr(x) # 3371 -> 188
    x = linear(w2, x)
    x = sigmoid(x)
    return x

```

**Figure 6.9.** NeuraLogic forward pass computational graph, optimized using the Vectorizer + the Optimizer with the default upward propagation configuration, built for an example GNN with two GCN convolutions.

Figure 6.9 shows the equivalent computational graph produced by the Compiler with *all optimizations enabled* and the upward propagation of gathers set to a modest configuration. These are the key differences:

- The redundant gathers of weights are no longer present.
- Linear operations are now performed before gather operations, which in this case is the better option. This is the result of the downward gather propagation optimization (Section 5.2.6).
- The numbers of values in the individual layers are significantly lower, including also the input (which is actually a fact layer, optimized together with the rest of the computational graph, as the computational graph itself is input-dependent). This is the result of, e.g., the isomorphic compression optimization (Section 5.2.6). Note that this reduction is *completely lossless*.
- There is an additional gather operation between the two final segment CSR operations. This is because the gather operation itself is a downwards-propagated one, and the upward propagation optimization (Section 5.2.7) decided not to remove it, as it found the operation to significantly reduce memory usage.

Figure 6.10 shows the computational graph that the Compiler produces when its upward gather propagation optimization is configured to the limitless (aggressive) setting. As you can see, there are no longer any gather operations, but the duplication of values is significantly higher as a result, starting with 18 298 values on the input.

Note that there are many possible additional equivalents of these computational graphs, including ones with *more* gather operations than in Fig. 6.9, or, conversely, with *less* gather operations than in Fig. 6.9. In fact, any gather operation can be upwards-propagated (i.e., removed), allowing for any possible balance between the total number of operations and additional memory utilization. As shown earlier, different configurations may have different impact on performance depending on the size of the dataset as well as batch size.

---

they are removable trivially. In the implementation of the Compiler, removing identity operations is not considered to be an “optimization,” and as such is done always (i.e., cannot be disabled).



```

def forward(x): # x is a tensor of shape (18298, 7)
    x = linear(w1, x)
    x = segment_csr(x) # 18298 -> 7441
    x = relu(x)
    x = linear(w2, x)
    x = segment_csr(x) # 7441 -> 3371
    x = segment_csr(x) # 3371 -> 188
    x = linear(w2, x)
    x = sigmoid(x)
    return x

```

**Figure 6.10.** NeuraLogic forward pass computational graph, optimized using the Vectorizer + the Optimizer, built for an example GNN with two GCN convolutions. The upward gather propagation optimization is set to the extreme (absolute) configuration.

## 6.3 Relational Architectures

We shall also study more complex relational architectures, going beyond the typical graph neural networks, albeit without having any baselines on the GPU to compare ourselves against.

As the dataset, we will be using the relational representation of the Mutagenesis dataset, discussed at the beginning of this chapter.

We will be comparing two different network architectures, not a lot more complicated than the GNNs we worked with previously, although with a few notable differences, making the resulting computational graphs more complex.

Let us first explain the two architectures using NeuraLogic template rules. The following is the simpler one:

$$\begin{aligned}
 \mathbf{W}_i[3, ] \text{ AtomEmbed}(A) &:- \text{AtomType}_i(A). \quad \forall i \in \{C, O, Br, I, F, H, N, Cl\} \\
 \mathbf{W}_j[3, ] \text{ BondEmbed}(B) &:- \text{BondStrength}_j(B). \quad \forall j \in \{B_1, B_2, B_3, B_4, B_5, B_7\} \\
 \text{Layer1}(X) &:- \mathbf{W}_3[3, 3] \text{ AtomEmbed}(X), \mathbf{W}_4[3, 3] \text{ AtomEmbed}(Y), \\
 &\quad \_Bond(X, Y, B), \text{BondEmbed}(B). \\
 \text{Layer2}(X) &:- \mathbf{W}_5[3, 3] \text{ Layer1}(X), \mathbf{W}_6[3, 3] \text{ Layer1}(Y), \\
 &\quad \_Bond(X, Y, B), \text{BondEmbed}(B). \\
 \text{Layer3}(X) &:- \mathbf{W}_7[3, 3] \text{ Layer2}(X), \mathbf{W}_8[3, 3] \text{ Layer2}(Y), \\
 &\quad \_Bond(X, Y, B), \text{BondEmbed}(B). \\
 \mathbf{W}_9[1, 3] \text{ Out} &:- \text{Layer3}(X)
 \end{aligned}$$

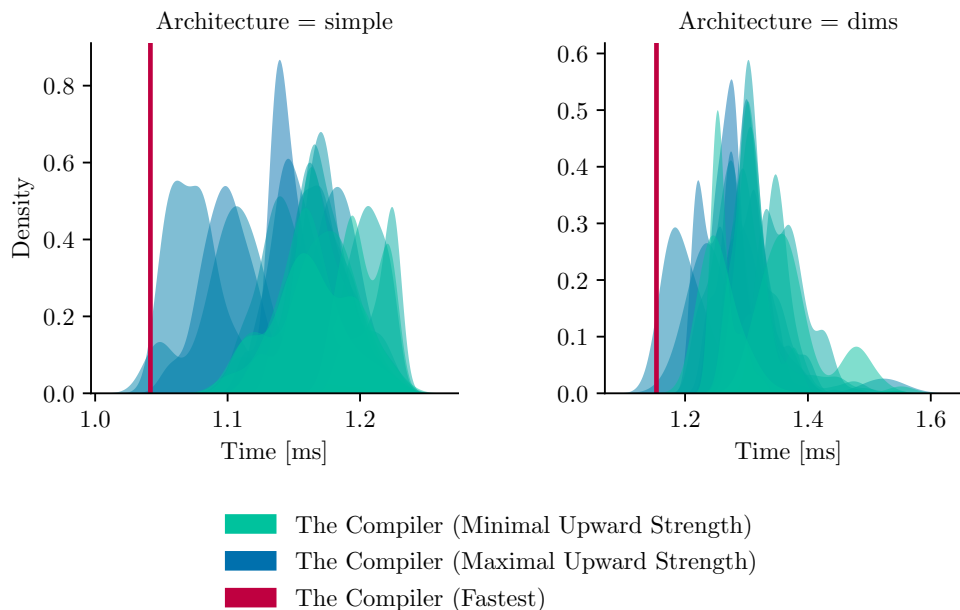
This network template uses learnable embeddings for nodes and edges. Since bonds (edges) now carry values, the bond predicate is now ternary (as opposed to graphs' edge predicates typically being binary). First, the network creates learnable embeddings for the 8 different atom (node) types, and separate learnable embeddings for the 6 different bond (edge) types. Then, it creates three consecutive “layers,” which follow the molecule structure (aggregating based on atom neighbors using the “\_Bond” predicate) and

combine the corresponding atom pair embeddings with the corresponding bond strength embeddings as well, which are then aggregated back to the individual source nodes  $X$ . The following is a slightly more complex version:

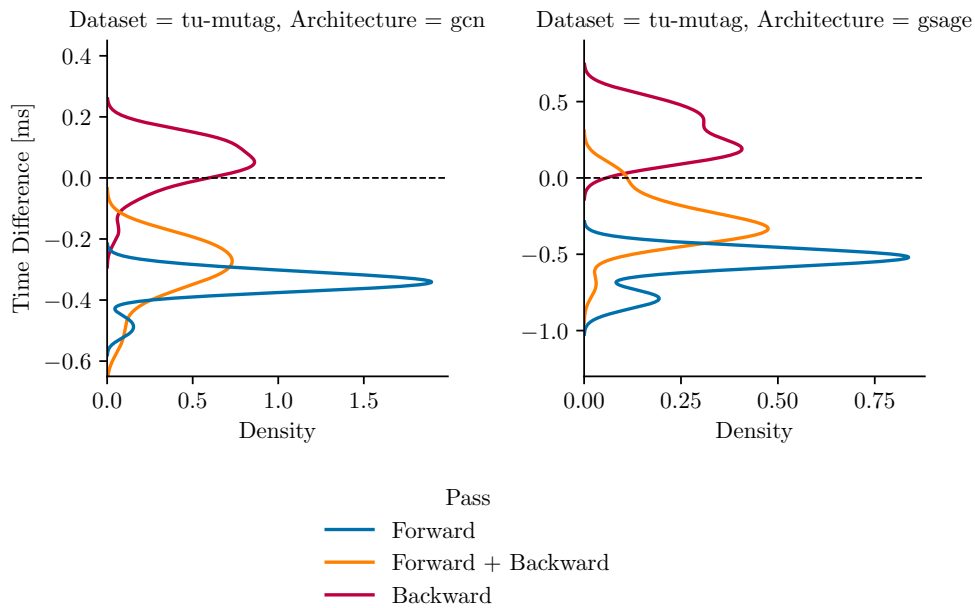
$$\begin{aligned}
\mathbf{W}_i[3, ] \text{ AtomEmbed}(A) &:- \text{AtomType}_i(A). \quad \forall i \in \{\text{C, O, Br, I, F, H, N, Cl}\} \\
\mathbf{W}_j[1, ] \text{ BondEmbed}(B) &:- \text{BondStrength}_j(B). \quad \forall j \in \{B_1, B_2, B_3, B_4, B_5, B_7\} \\
\text{Layer1}(X) &:- \mathbf{W}_3[3, 3] \text{ AtomEmbed}(X), \mathbf{W}_4[3, 3] \text{ AtomEmbed}(Y), \\
&\quad \text{Bond}(X, Y, B), \mathbf{W}_5[3, 1] \text{ BondEmbed}(B). \\
\text{Layer2}(X) &:- \mathbf{W}_6[3, 3] \text{ Layer1}(X), \mathbf{W}_7[3, 3] \text{ Layer1}(Y), \\
&\quad \text{Bond}(X, Y, B), \mathbf{W}_8[3, 1] \text{ BondEmbed}(B). \\
\text{Layer3}(X) &:- \mathbf{W}_9[3, 3] \text{ Layer2}(X), \mathbf{W}_{10}[3, 3] \text{ Layer2}(Y), \\
&\quad \text{Bond}(X, Y, B), \mathbf{W}_{11}[3, 1] \text{ BondEmbed}(B). \\
\mathbf{W}_{12}[1, 3] \text{ Out} &:- \text{Layer3}(X)
\end{aligned}$$

The main difference of this architecture from the former is that the bond embedding vectors have different dimensionalities than the atom embedding vectors. This also means that the bond embeddings must be weighted in the individual layers as well (which they previously were not), otherwise the dimensions would not match. Also, the “Bond” predicate is no longer used implicitly (see Chapter 4 for details).

Figure 6.11 shows the resulting performance on the GPU. The corresponding computational graph is significantly more complex, and it is shown in Appendix A for the latter (“dms”) example, with maximal strength of the upwards propagation optimization.



**Figure 6.11.** Forward pass performance comparison of more complex architectures on a GPU, with outliers excluded.



**Figure 6.12.** Performance comparison between Compiler-built computational graphs utilizing scatter operations, when compared to segment CSR. The Y axis shows the time difference between scatter and segment CSR runtimes (for otherwise equivalent computational graphs): negative values correspond to segment CSR being faster, whereas positive values correspond to scatter being faster.

## 6.4 The Backward Pass

For the backward pass, we use the “autograd” feature of PyTorch, which automatically infers the backward pass computational graph from any forward pass computational graph. A notable downgrade in performance can be observed compared to the forward pass. This is mainly caused by PyTorch autograd, which is explained below.

A comparison for overall best-performing models (with fastest combined forward + backward pass) is shown on Figures 6.13 (p. 73) and 6.14 (p. 74). (Additional plots are available in Appendix C.) As you can see, when compared against PyTorch Geometric, even though the forward pass is typically faster for the Compiler, the backward pass is in fact quite substantially slower. The overall performance of the forward and the backward pass combined is tied between the two, though.

The slow backward pass is in fact caused entirely by the differences between the backward pass implementations of the segment CSR operation, used by the Compiler, and the scatter operation, used by PyTorch Geometric. Figure 6.12 shows a performance comparison between the forward and backward passes when using scatter operations in the Compiler in place of segment CSR operations. The computational graphs are otherwise identical. As expected based on Chapter 3, Section 3.4.3, the forward pass is significantly faster for the segment CSR operation compared to the scatter equivalent. However, the opposite is the case for the backward pass, where the segment CSR back-propagation computation is *noticeably slower*, thus negating its performance advantage on the forward pass.

However, since the segment CSR (forward) operation performs *the exact same computation* as the scatter operation, there is no reason for its backward pass to be slower, since it is fully exchangeable with the backward pass implementation of the scatter

operation. The segment CSR operation is a special case of the scatter operation, but not the other way around, and so we can use the backward pass implementation of the scatter operation for either of the two operations. Therefore, we can easily fix this by simply using the backward pass implementation of the scatter operation in place of the segment CSR backward pass.

The reason why the backward pass implementations for the two operations differ in PyTorch in the first place, is that the scatter operation is in the PyTorch base library, whereas the segment CSR operation is not. The segment CSR operation comes from an external source, namely the `pytorch_scatter` library [80], which has its own low-level implementation for both its forward and its backward pass, for both CPU and GPU hardware.

The backward pass of the scatter operation is *the gather operation*. For the segment CSR operation, the backward pass is implemented using a “gather CSR” operation, which is a gather operation that uses the index sequence in the compressed (CSR) representation (discussed in Chapter 3, Section 3.4.3).

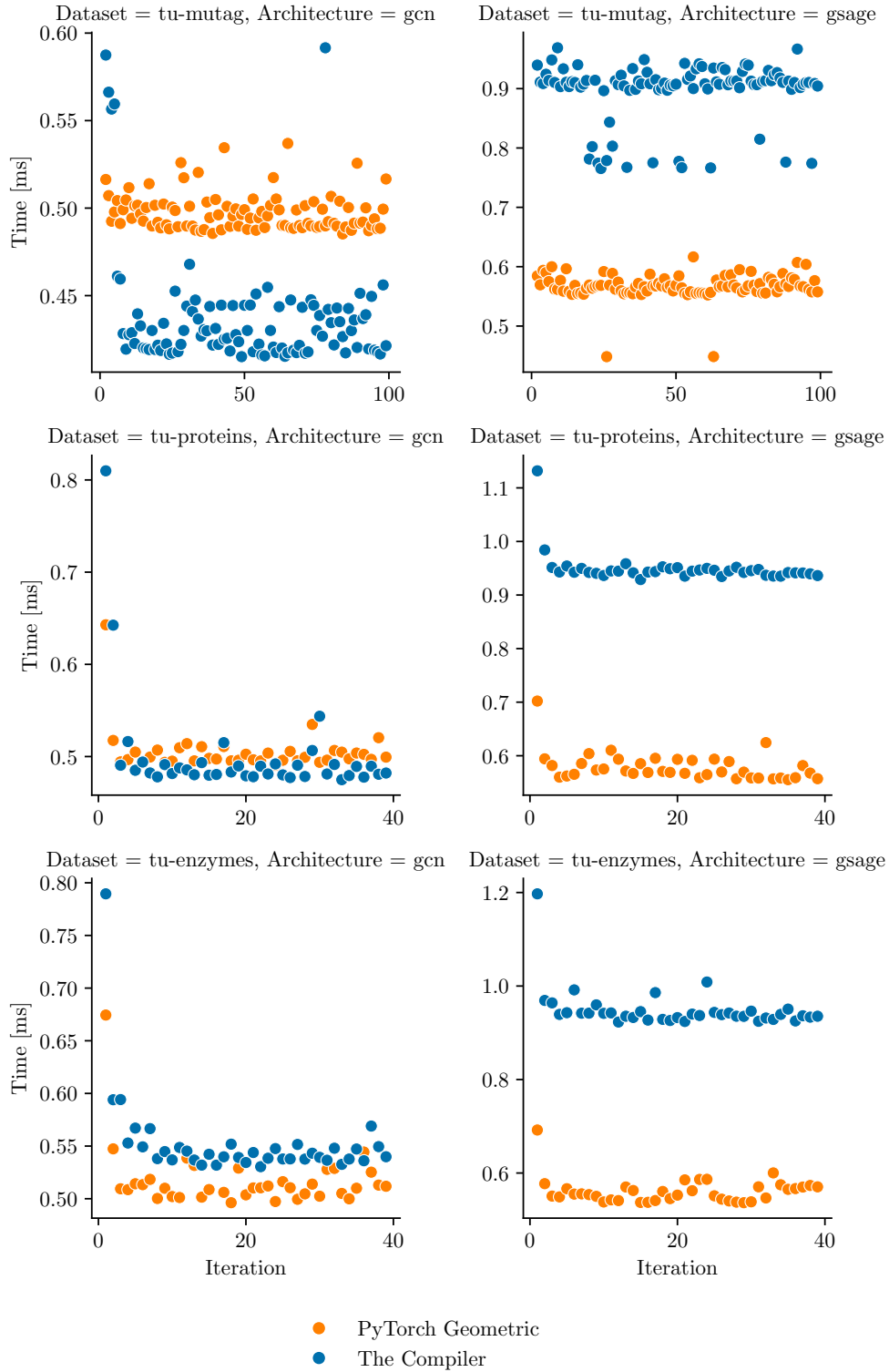
One possible explanation for the performance difference is that the “gather CSR” operation in the `pytorch_scatter` library is less optimized than the gather operation in the base PyTorch library. Another possibility is that the “gather CSR” operation is simply a more difficult operation to perform. If the latter is the case, then there is no reason for us not to use segment CSR on the forward pass, and the standard gather implementation on the backward pass, each utilizing different index sequence representations. Either way, this is a resolvable issue.

*Ultimately, it appears that the slow backward passes that we measured are not indicative of any shortcomings of the Compiler, but are rather the result of a shortcoming in the `pytorch_scatter` library. The performance difference measured on the forward pass is thus the most indicative of the performance benefits that the Compiler offers.*

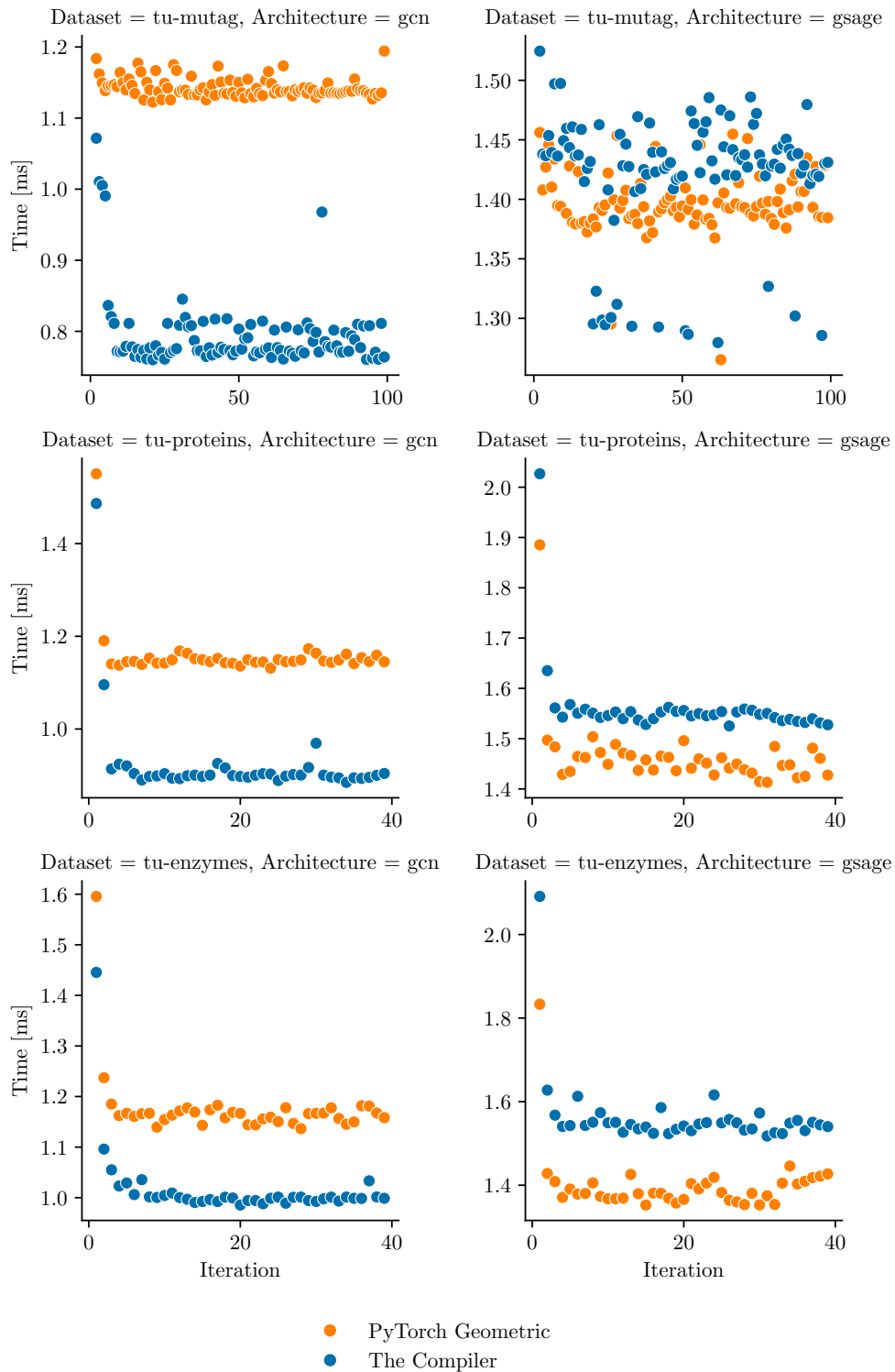
## 6.5 Graphcore Intelligence Processing Units (IPUs)

Unlike GPUs, since Graphcore IPUs are not SIMD, and as such are able to parallelize different operations at once (Chapter 2, Section 2.1.3, p. 7), earliest experiments performed on the IPU involved attempts at skipping the vectorization/compilation process entirely. This idea comes naturally, as the batching/vectorization of dynamic computational graphs is only needed as a result of the design of typically available hardware, and the IPU appears to be designed to overcome these hardware shortcomings. Furthermore, the Graphcore IPU uses its own computational graph compiler, which takes the input computational graph and transforms it for the IPU such that different computations are “batched” together, i.e., executed on the IPU in parallel, even when the operations are themselves different.

However, the early experiments showed that custom preprocessing of computational graph, i.e., vectorization, is still necessary, even for execution on the IPU, as the IPU expects vectorized computational graphs on its input. In fact, essentially all of the custom libraries available for the IPU provide APIs that are more-less similar to, e.g., PyTorch or TensorFlow, which means that they are designed to build computational graphs that are already vectorized. Therefore, the focus of this work shifted to the task of the computational graph vectorization early on, as the outcome of the initial IPU experiments was that the vectorization is necessary nonetheless, irrespective of the use of specialized hardware.



**Figure 6.13.** Backward pass of the best-performing (for the forward+backward pass) configurations on the GPU, on GNN architectures.



**Figure 6.14.** Forward + backward pass of the best-performing configurations on the GPU, on GNN architectures.

The aforementioned IPU experiments were done as follows: An example proof-of-concept forward pass was implemented using IPU-exclusive APIs, first using PopXL (a high-level computation graph-building library in Python for the Graphcore IPU), and later using Poplar (the underlying computation graph-building library with a C++ API). The proof-of-concept was written simply to see how the Graphcore graph compiler operates on graphs of thousands of operations on small tensors, as opposed to a few tens to a few hundreds of operations on large tensors.

The algorithm involved running one forward pass on a single batch of a fixed number of data “samples,” where each data sample was run through a simple feed forward neural network transforming a vector of 2 scalars to a scalar, with a hidden layer dimension of 8 units, with weights shared across the individual small modules.<sup>2</sup> Even though all the small neural networks were identical, and transforming them such that they operate on an arbitrarily large batch with a single forward pass is trivial in this particular case, the runtime performance on the IPU hardware should be equivalent even when not explicitly vectorized beforehand.

As expected, the runtime performance itself of a single forward pass through an arbitrarily large batch was fast. However, unfortunately, the *compilation time*, i.e., the preprocessing of the highly granular input computational graph by the Graphcore compiler, was slow.

The original aim was to test batch sizes that would reach the memory limit of the hardware, as a proof of concept for large datasets. Then, the subsequent goal was to proceed to finding a good solution to the IPU-specific batching problem (Section 2.1.3). However, the compilation time reached highly impractical runtimes too early on, where a computational graph of mere 8192 independent connected components of nodes (the example described above) already took over three minutes of compilation. For any practical real-life examples of non-vectorized computational graphs, this approach was entirely infeasible, as the compilation did not finish.

Therefore, the simple conclusion for the IPU was that computational graph vectorization is needed nonetheless. The testing of the finished Vectorizer/Optimizer from Chapter 5 was unfortunately not done on the IPU due to a lack of time. However, it can be reasonably expected that the performance will be better than on the GPU, as long as the input is not batched (as the IPU-specific batching problem does not have a good solution for the arbitrary computational graph structures of ours). The reason why the single-batch performance is expected to be better than on the GPU is that the IPU compiler will take the vectorized input (of up to hundreds of nodes, not thousands) and vectorize it further, as part of its compilation of the program to low-level IPU instructions, as long as there are independent nodes in the computational graph to be found in the first place. Furthermore, the IPU implementations for gather/scatter operations in and of themselves are likely significantly faster than their GPU equivalents. Therefore, the IPU might still be useful for future testing.

<sup>2</sup> This resembles the computational graph structures we keep seeing throughout this entire thesis. Even though this example is trivially vectorizable, the goal was to see whether the explicit vectorization step is necessary in the first place, and this example network, itself highly granular, was a mere representation of the full class of possible computational graphs in the non-vectorized, highly granular form.

## Chapter 7

### Conclusion


The training of deep neural networks comes with a plethora of unique challenges when highly structural data is involved, which the classical, non-structural deep neural networks typically are not burdened by. These challenges mainly pertain achieving good performance and scalability with respect to large input datasets, as the lack of either of these properties limits effective training only to simple neural networks. Existing solutions either reduce the data to simpler (i.e., non-structural) representations, in order to train on top of such data effectively, which in turn limits the inferential and representational capabilities of the models, or they simply do not have sufficient performance to scale well altogether.

Deep relational learning, itself operating on top of structural data, is affected by these challenges as well. Despite the fact that it pertains deep learning on arguably the most widespread data representations of today – the relational representations, deep relational learning itself is not widespread, likely because effectively scaling deep learning on such data is still an open problem. On the other hand, graph neural networks – a subset of deep relational learning architectures, able to process a subset of relational data representations – found a reasonable middle ground between restricting the data and the architectures, and having good computational performance and scalability.

To offer a direction towards solving the problem of scaling deep *relational* networks for arbitrary *relational* data, a solution was presented in this thesis, based on extending the scalability principles used in graph neural networks onto the relational superset of these networks. A compiler for differentiable relational (dynamic) computational graphs was developed as a result, based on the generalization of the computational concepts promoted by GNNs, together with precompilation of the networks. It was demonstrated that on the subset of data and computational graphs equivalent to GNNs, it achieves performance and scalability *equivalent or better* to that of commonly available GNN solutions. Furthermore, it was shown that the capabilities of the resulting system, to ensure scalability of NNs, *extend beyond GNNs*, to NN architectures that no longer fit the GNN paradigm. As such, the capabilities of the compiler were demonstrated both in terms of *the raw performance* on arbitrary hardware (compared against the GNN baseline), as well as in terms of its capability of effectively and efficiently *covering the full range of relational data representations* and deep relational networks. This was shown by demonstrating how the compiler is able to cover the full range of the computational graphs constructible in NeuraLogic – a framework for building and training deep relational networks, on top of relational data, *including directly from relational databases*.

*The initial results were shown to be highly promising for the scalability of deep relational learning as a whole.* As an immediately usable result, the compiler can be used together with NeuraLogic to train arbitrary deep relational networks, e.g., on the GPU, using arbitrary backends/frameworks (not just PyTorch, on top of which the outputs of the





compiler were mainly tested). Previously, training on the GPU, the IPU, or other hardware was not possible using just NeuraLogic by itself.

Of course, there is plenty of room for follow-up work. There are additional ways to improve the performance even further, and the research in this regard can proceed in parallel with the research into the relational neural network architectures themselves.

Firstly, the simplest improvements can be made in the low-level gather/scatter implementations themselves; namely in the backward pass implementation of the segment CSR operation. Similarly, the gather operation can be studied for whether a faster equivalent can be implemented for the GPU, e.g., by also utilizing data locality to a performance advantage, much like the segment CSR operation is able to utilize data locality to achieve better performance over the scatter operation.

Secondly, there is space for additional, more advanced optimizations in the resulting computational graph compiler. The more complex the network architectures, the more opportunities for more advanced optimizations arise based on the resulting computational graphs, which is to be expected.

Lastly, the usability of specialized hardware, such as the Graphcore IPUs, should be studied further. While the compiler is perfectly usable on the IPU, it has not been tested to see the results in terms of the actual resulting performance on this hardware.

Ultimately, there is reason to believe that similar pre-compilation and optimization of the computational graphs is the way forward for training neural networks on top of structural data in general. This is because in a way, it can itself be understood as a mere extension of the ideas already successfully employed in the training of GNNs, where instead of pre-compilation, the individual computational graph optimizations, from which the performance and scalability stems, are merely hand-made. As the results of this thesis show, it is not only unnecessary for these optimizations to be hand-made, but the added compilation step is even able to outdo them at times. The question hence becomes: how much further can we get with this?



# Appendix A

## Complex Computational Graph Example

This is the computational graph for the second example from Section 6.3 (p. 69).

```
# input: unit tensor, with shape (3,1)
def forward(unit31):
    # bond embeddings:

    # w_bond: shape (6,1,1)
    b = tanh(w_bond)

    # atom embeddings:

    # w_a: shape (16,1,1)
    w_a = concat([wB1, wB6, wB7, wB2,
                  wB8, wB5, wB4, wB3,
                  wB6, wB1, wB7, wB2,
                  wB8, wB5, wB4, wB3])
    a = tanh(w_a)

    # layer 1:

    # w_3_4: shape (2,3,3)
    l1 = linear(reshape(w_3_4, (2,1,3,3)), reshape(a, (2,8,3,1)))
    l1 = reshape(a, (16,3,1))

    # w_5: shape (1,3,1)
    b1 = gather(b) # 6 -> 6
    b1 = linear(w_5, b1) # (6,3,1)

    l1 = concat([l1, b1, unit31]) # (23,3,1)
    l1 = gather(l1) # 23 -> 2000
    l1 = sum(reshape(l1, (4,500,3,1)), dim=0) # -> (500,3,1)
    l1 = tanh(l1)
    l1 = segment_csr(l1) # 500 -> 174
    l1 = tanh(l1)
```

```

# layer 2:

# w_6_7: shape (2,3,3)
l2 = linear(reshape(w_6_7, (2,1,3,3)), reshape(l1, (2,87,3,1)))
l2 = reshape(l2, (174,3,1))

# w_8:  shape (1,3,1)
b2 = gather(b)           # 6 -> 6
b2 = linear(w_8, b2)     # (6,3,1)

l2 = concat([l2, b2, unit31]) # (180,3,1)
l2 = gather(l2)          # 174 -> 9440
l2 = sum(reshape(l2, (4,2360,3,1)), dim=0) # -> (2360,3,1)
l2 = tanh(l2)
l2 = segment_csr(l2)     # 2360 -> 834
l2 = tanh(l2)

# layer 3:

# w_9_10: shape (2,3,3)
l3 = linear(reshape(w_9_10, (2,1,3,3)), reshape(l2, (2,417,3,1)))
l3 = reshape(l3, (834,3,1))

# w_11:  shape (1,3,1)
b3 = gather(b)           # 6 -> 6
b3 = linear(w_11, b3)     # (6,3,1)

l3 = concat([l3, b3, unit31]) # (840,3,1)
l3 = gather(l3)          # 840 -> 41944
l3 = sum(reshape(l3, (4,10486,3,1)), dim=0) # -> (10486,3,1)
l3 = tanh(l3)
l3 = segment_csr(l3)     # 10486 -> 4893
l3 = tanh(l3)

# predict:
predict = segment_csr(l3) # 4893 -> 188

# w_12: (1,1,3)
predict = linear(w_12, l3)
predict = tanh(predict)

# result shape: (188,1,1)
return predict

```

# Appendix B

## Code Guide

The code is available mainly on GitHub.<sup>1</sup> The following is its high-level structure:<sup>2</sup>

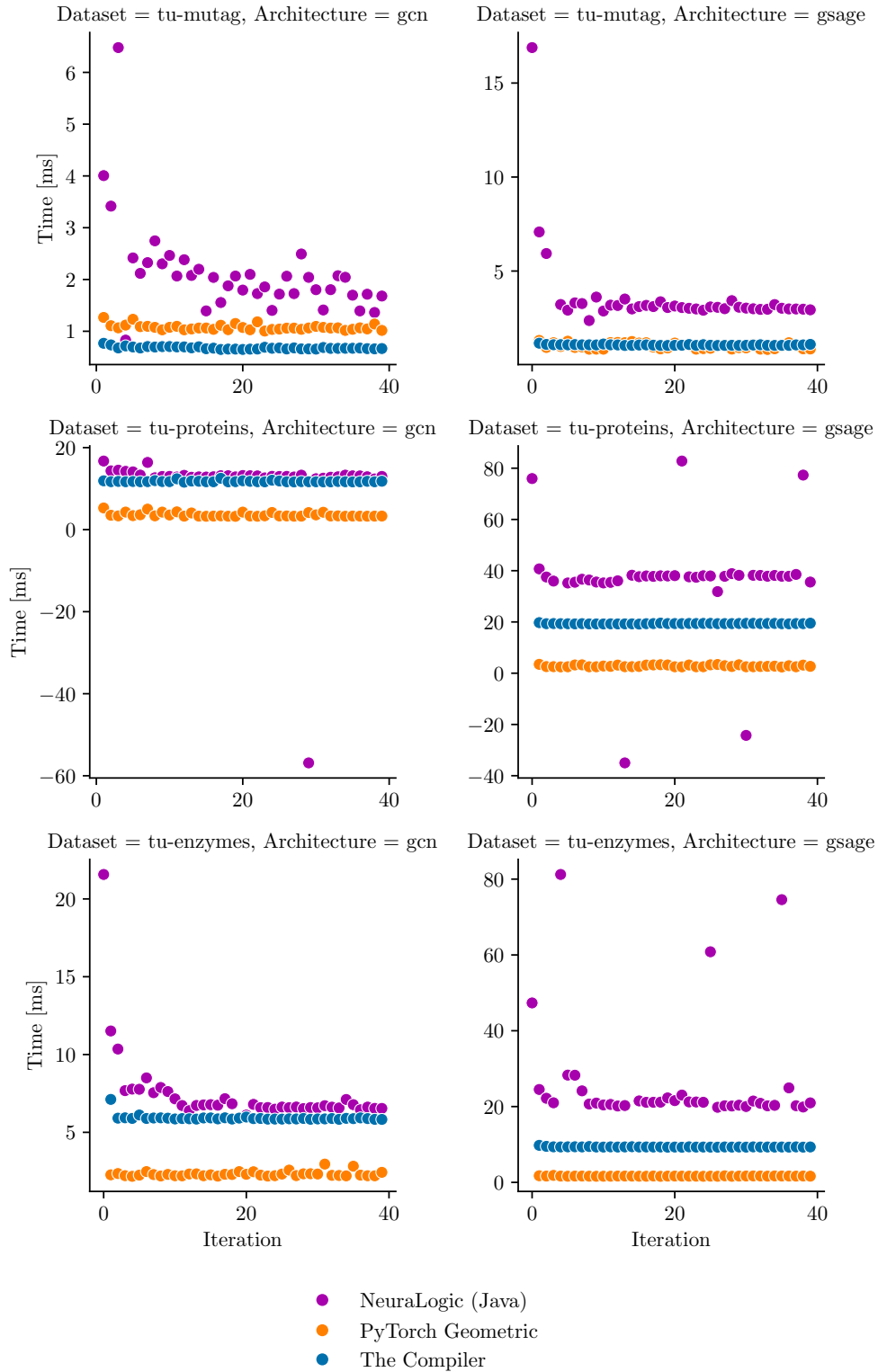
- `benchmark.py` - executable script for running benchmarks, such as those done in Chapter 6
- `lib/` - directory containing the main code of the library
  - `benchmarks/` - utilities for running benchmarks using different technologies (the Compiler, Java NeuraLogic, PyTorch Geometric)
  - `datasets/` - Contains a unified API for the datasets, for both the Compiler, as well as PyTorch Geometric, where applicable. Contains also the individual NN architecture templates.
  - `engines/torch/` - Contains the final step of the Compiler - the conversion of the symbolic computational graph to a PyTorch module
  - `facts/` - utilities for parsing and writing datasets/rules in the format recognized by NeuraLogic
  - `model/` - type definitions for individual aggregation and transformation operations
  - `sources/` - the API for the input computational graphs for the Compiler
    - `minimal_api/` - the minimal API interface, and example implementations, namely for the input from NeuraLogic (including all the NeuraLogic-specific code).
    - `base.py`, `base_impl.py`, `builders.py` - the full (easy to use, “Pythonic”) input API interface and implementation
    - `minimal_api_bridge.py` - utilities for converting minimal API implementations to the full API
  - `vectorize/` - the Compiler itself, including all the optimizations
    - `model/` - the symbolic data structures that together represent intermediate vectorized computational graphs, as well as the resulting computational graphs
      - `op_network.py` - the model for the final data representation of the resulting computational graphs
    - `pipeline/` - the individual optimizations
      - `pipeline.py` - The main entry point, containing the composition of the final optimization sequence, as described in detail in Chapter 5, Section 5.2.10.
    - `settings.py` - settings structure for the configuration of the Compiler
    - `settings_presets.py` - iterators over various Compiler settings presets (configurations)

<sup>1</sup> <https://github.com/neumannjan/nn-structural-graph-vectorizer-compiler>

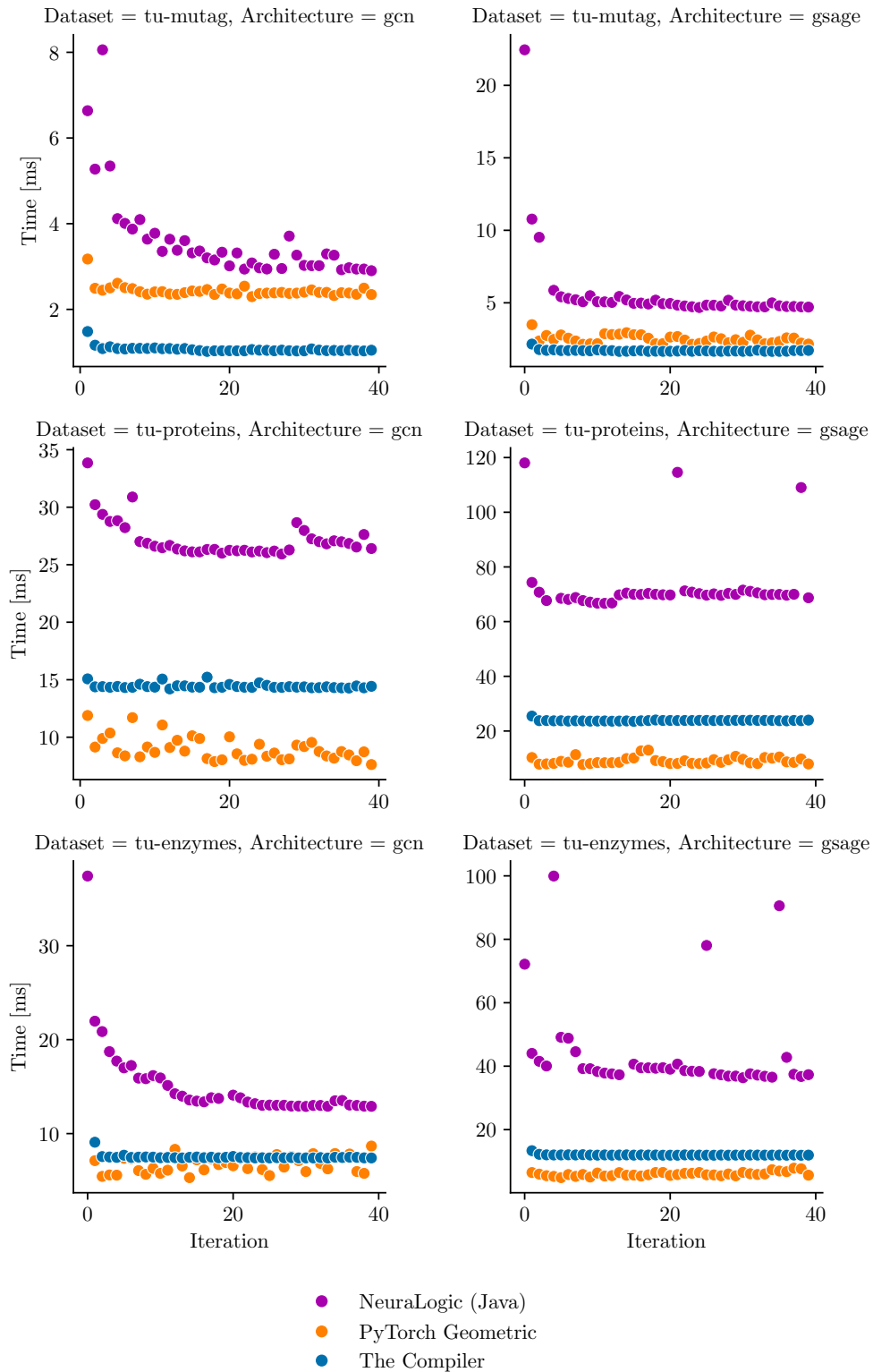
<sup>2</sup> As of May 24, 2024.



**Appendix C**  
**Additional Figures**

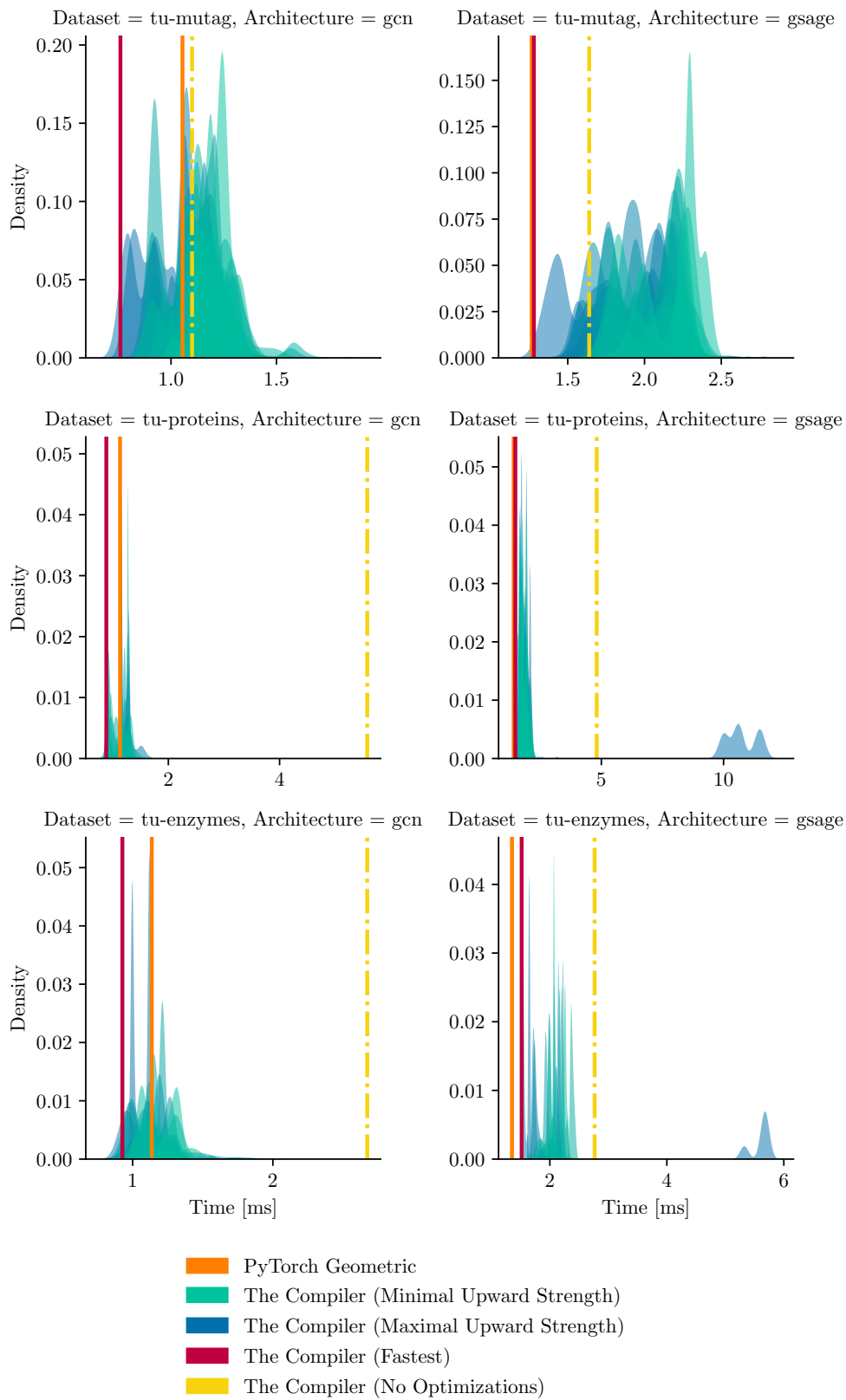


**Figure C.1.** Backward pass of the best-performing (for the forward+backward pass) configurations on the CPU, on GNN architectures.

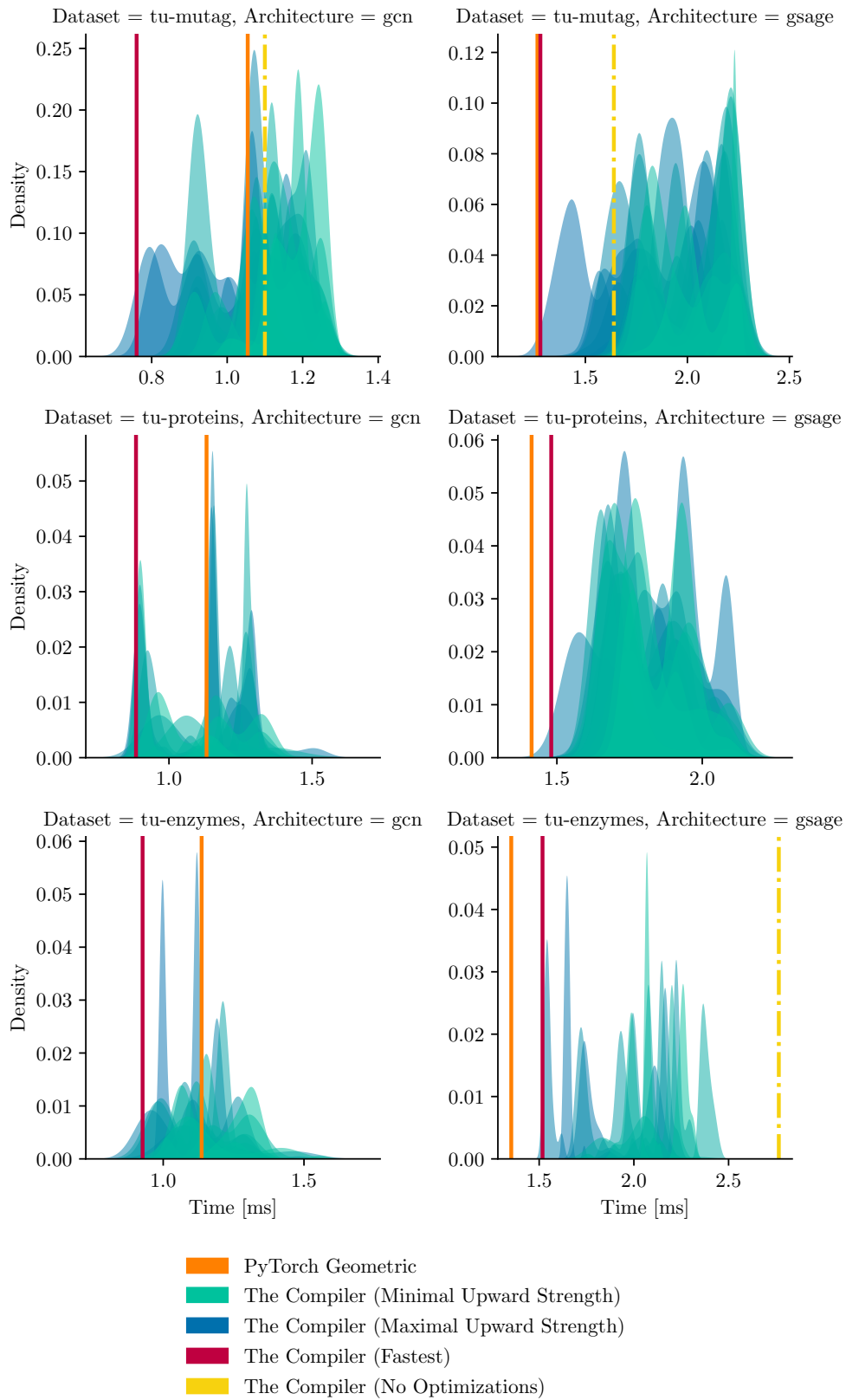


**Figure C.2.** Forward + backward pass of the best-performing configurations on the CPU, on GNN architectures.

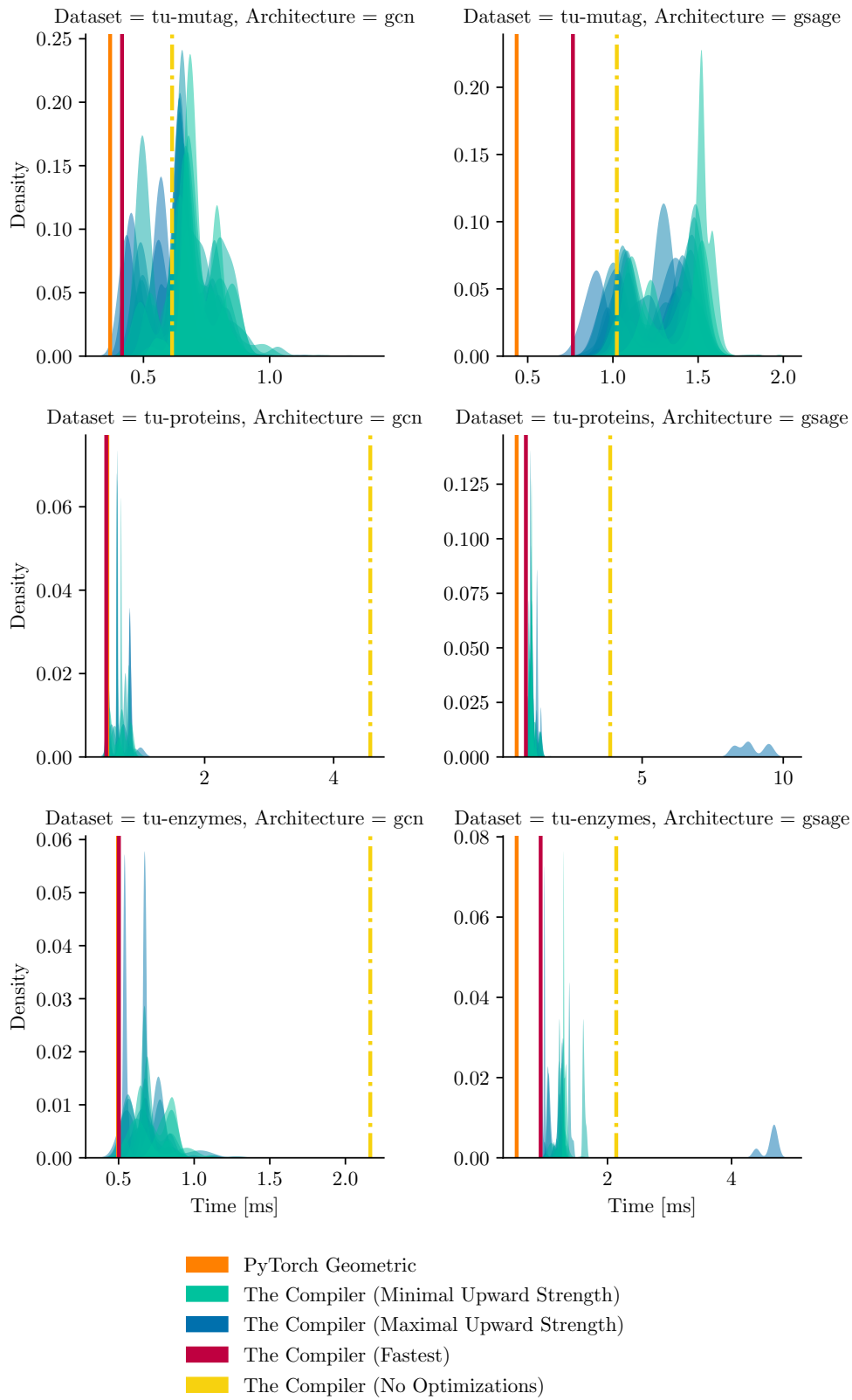




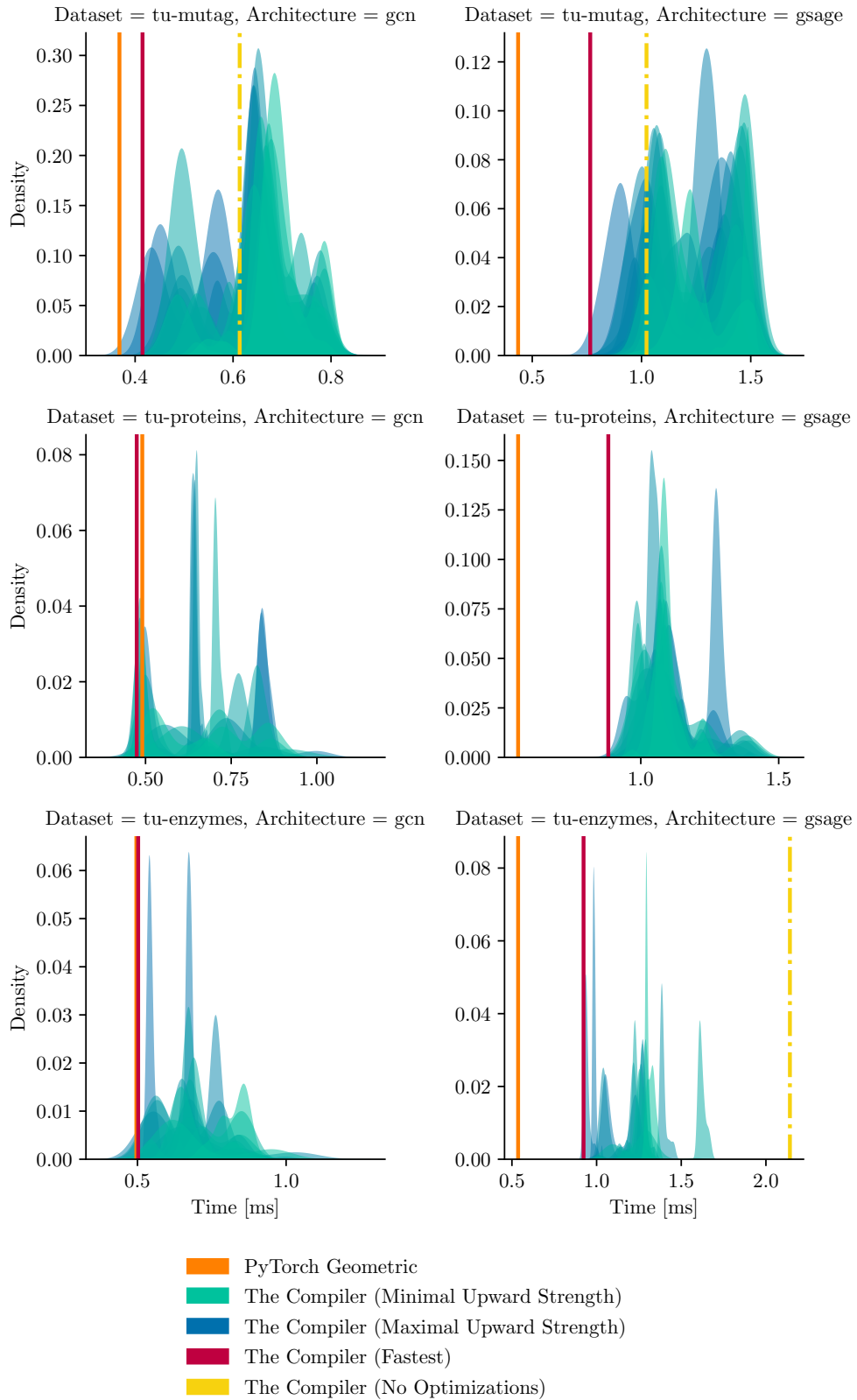
**Figure C.3.** Forward + Backward pass performance comparison of GNN architectures on a GPU.



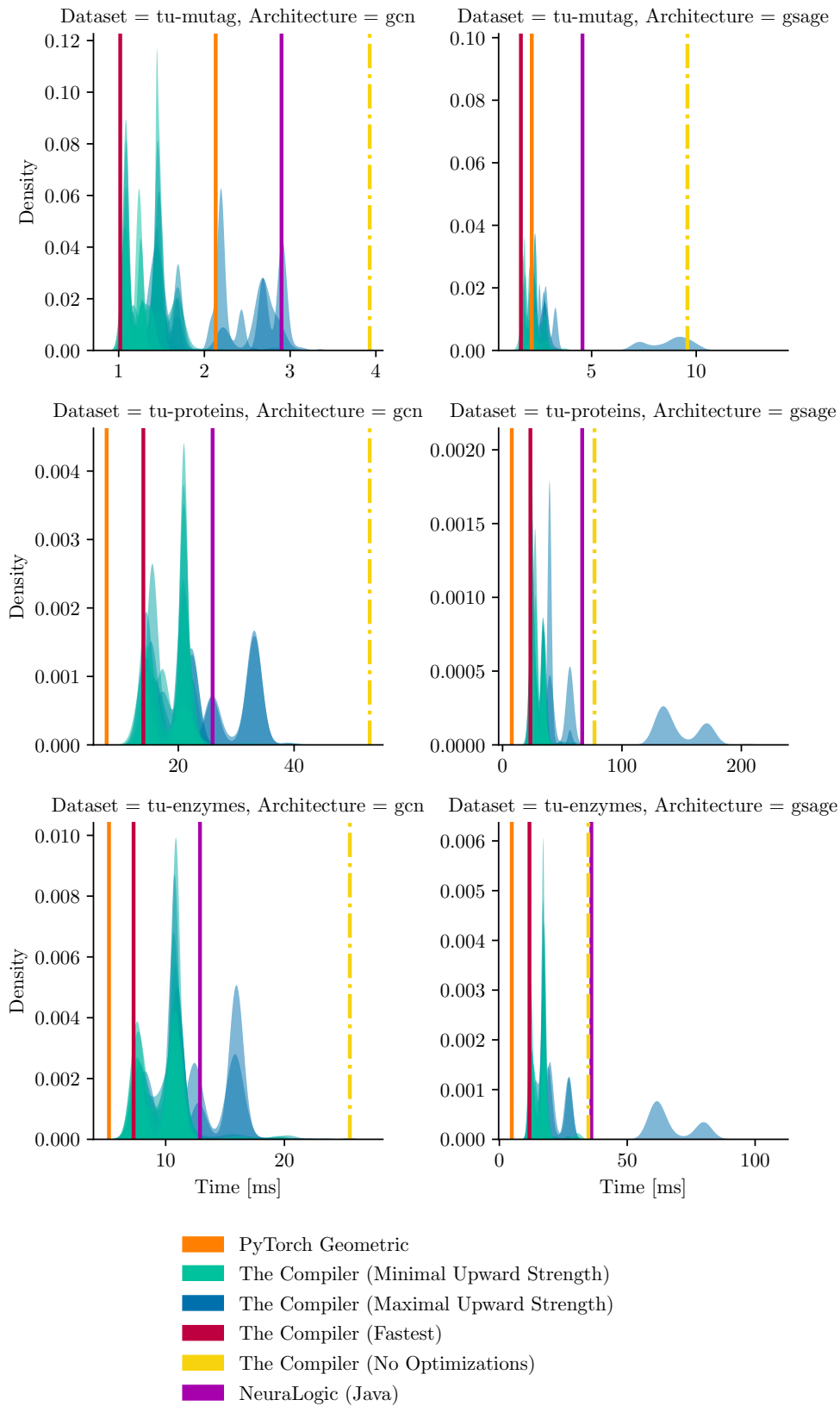
**Figure C.4.** Forward + Backward pass performance comparison of GNN architectures on a GPU, close-up view (with outliers excluded).



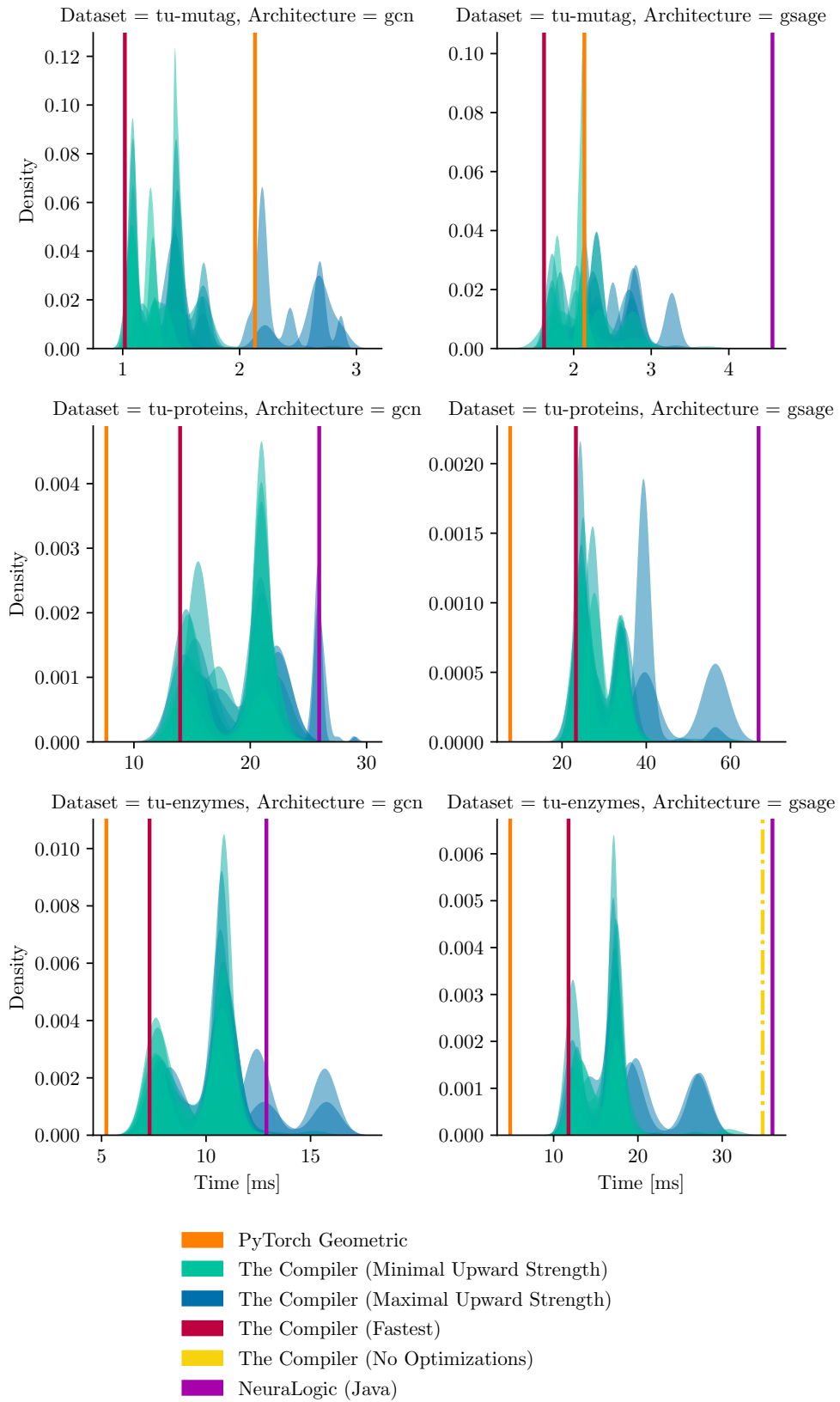
**Figure C.5.** Backward pass performance comparison of GNN architectures on a GPU.



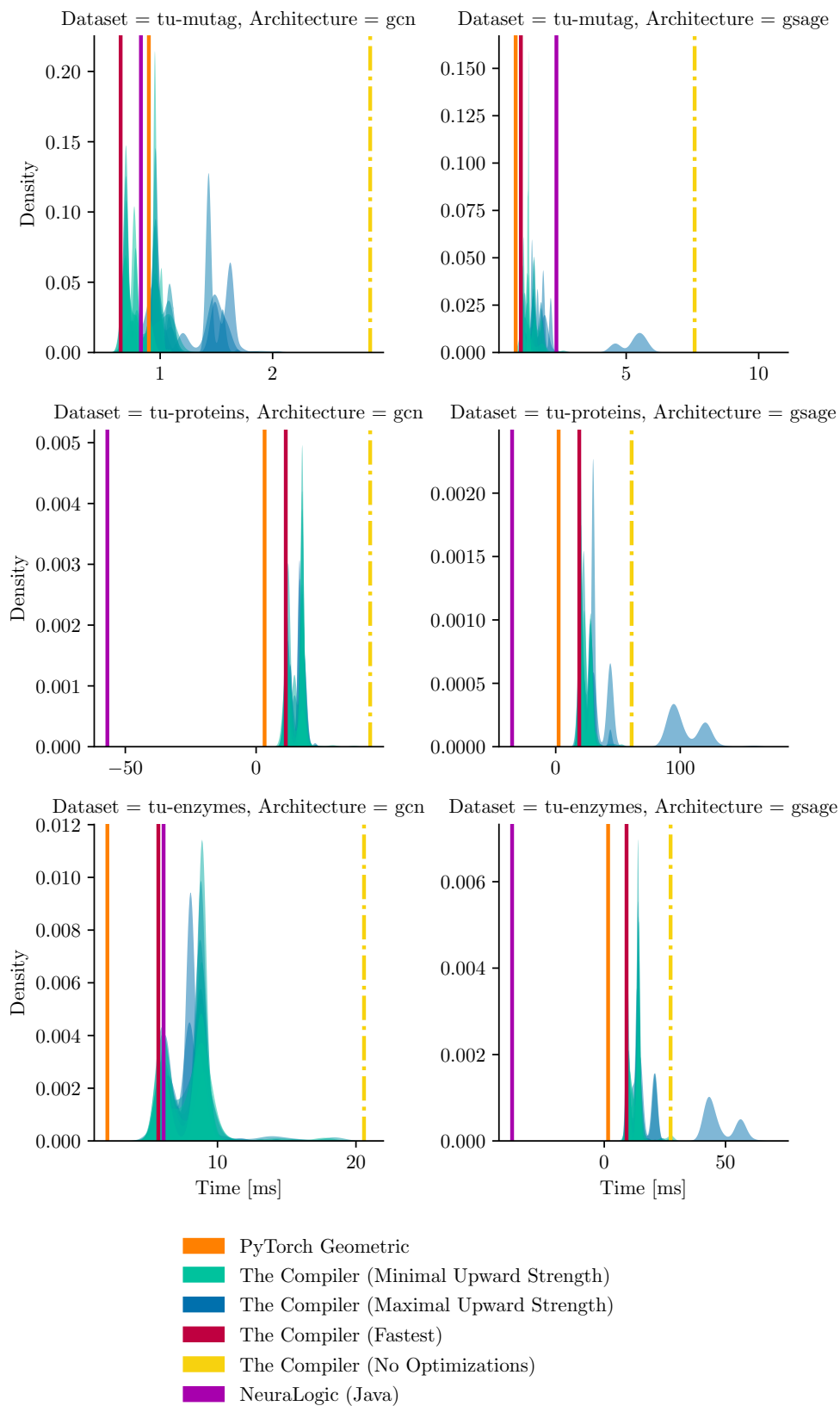
**Figure C.6.** Backward pass performance comparison of GNN architectures on a GPU, close-up view (with outliers excluded).



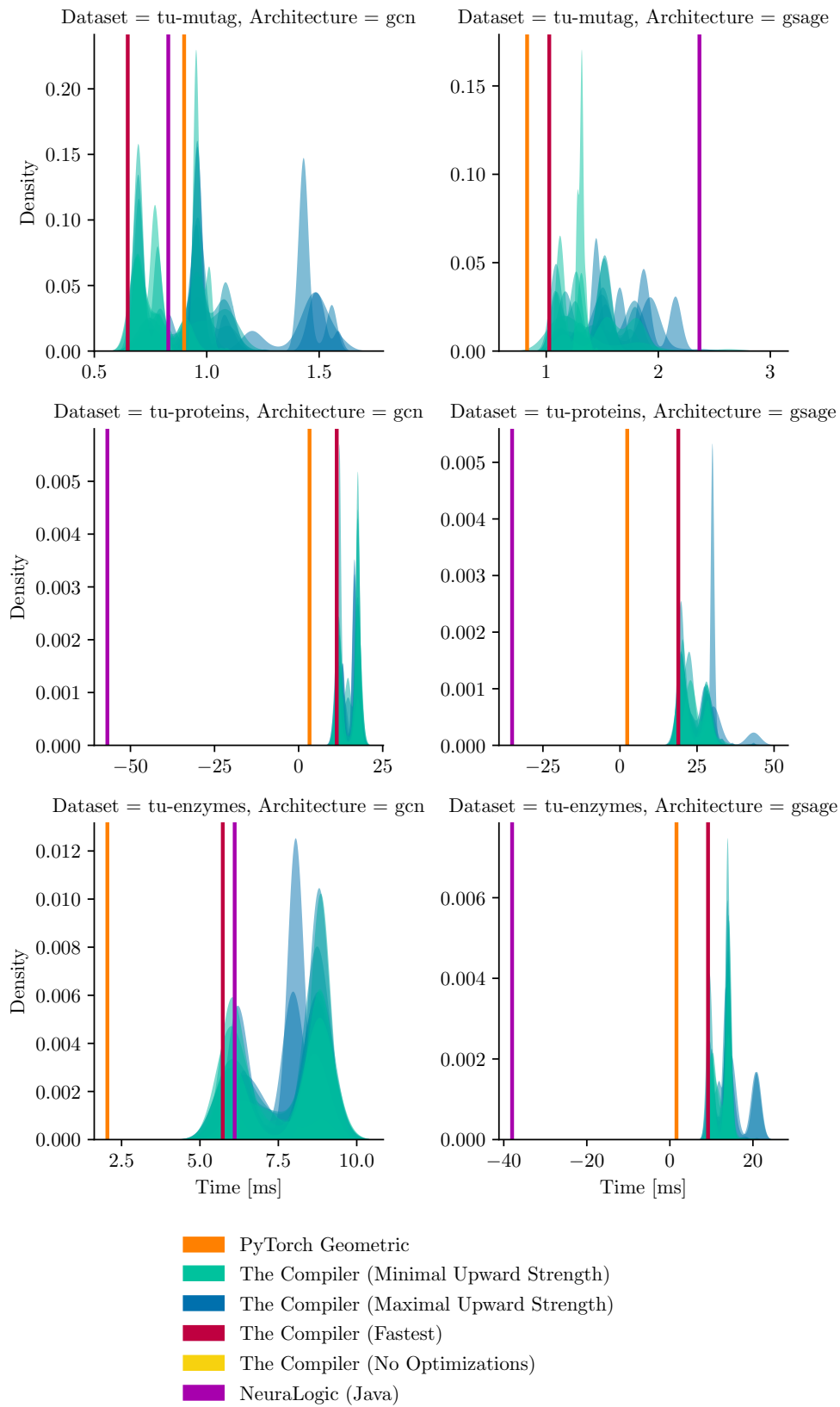
**Figure C.7.** Forward + Backward pass performance comparison of GNN architectures on a CPU.



**Figure C.8.** Forward + Backward pass performance comparison of GNN architectures on a CPU, close-up view (with outliers excluded).



**Figure C.9.** Backward pass performance comparison of GNN architectures on a CPU.



**Figure C.10.** Backward pass performance comparison of GNN architectures on a CPU, close-up view (with outliers excluded).



## Appendix D

### Glossary

AOT	■	Ahead-of-time compilation
CNN	■	Convolutional neural network
COO	■	Coordinate list (sparse matrix representation format)
CPU	■	Central processing unit
CSC	■	Compressed sparse column (sparse matrix representation format)
CSR	■	Compressed sparse row (sparse matrix representation format)
DAG	■	Directed acyclic graph
FOL	■	First-order logic
GCN	■	Graph convolutional network
GNN	■	Graph neural network
GPU	■	Graphics processing unit
ILP	■	Inductive logic programming
IPU	■	Intelligence processing unit
JIT	■	Just-in-time compilation
MIMD	■	Multiple instruction, multiple data
NN	■	Neural network
RDBMS	■	Relational database management system
SIMD	■	Single instruction, multiple data
SQL	■	Structured query language
SRL	■	Statistical relational learning

## References

- [1] BREIMAN, Leo, Jerome FRIEDMAN, R. A. OLSHEN, and Charles J. STONE. *Classification and Regression Trees*. New York: Chapman and Hall/CRC, 2017. ISBN 978-1-315-13947-0. Available from DOI 10.1201/9781315139470.
- [2] HEARST, M.A., S.T. DUMAIS, E. OSUNA, J. PLATT, and B. SCHOLKOPF. Support Vector Machines. *IEEE Intelligent Systems and their Applications*. 1998, Vol. 13, No. 4, pp. 18–28. ISSN 2374-9423. Available from DOI 10.1109/5254.708428.
- [3] GOODFELLOW, Ian, Yoshua BENGIO, and Aaron COURVILLE. *Deep Learning*. MIT Press, 2016 . Adaptive Computation and Machine Learning Series. ISBN 978-0-262-03561-3. Available from <https://books.google.cz/books?id=Np9SDQAAQBAJ>.
- [4] SHARIFANI, Koosha, and Mahyar AMINI. Machine Learning and Deep Learning: A Review of Methods and Applications. *World Information Technology and Engineering Journal*. 2023, Vol. 10, No. 07, pp. 3897–3904. Available from <https://papers.ssrn.com/abstract=4458723>.
- [5] BRANTS, Thorsten, Ashok C. POPAT, Peng XU, Franz J. OCH, and Jeffrey DEAN. Large Language Models in Machine Translation. In: Jason EISNER, ed. *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*. Association for Computational Linguistics, 2007. pp. 858–867. Available from <https://aclanthology.org/D07-1090>.
- [6] HALPIN, Terry, and Tony MORGAN. *Information Modeling and Relational Databases*. Morgan Kaufmann, 2010 . ISBN 978-0-08-056873-7.
- [7] CODD, E. F.. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*. 1970, Vol. 13, No. 6, pp. 377–387. ISSN 0001-0782. Available from DOI 10.1145/362384.362685.
- [8] CODD, E. F.. *Relational Completeness of Data Base Sublanguages*. San Jose, California: IBM Corporation, 1972.
- [9] DEAN, Jeffrey, Greg CORRADO, Rajat MONGA, Kai CHEN, Matthieu DEVIN, Mark MAO, Marc' aurelio RANZATO, Andrew SENIOR, Paul TUCKER, Ke YANG, Quoc LE, and Andrew NG. Large Scale Distributed Deep Networks. In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2012. Available from [https://proceedings.neurips.cc/paper\\_files/paper/2012/hash/6aca97005c68f1206823815f66102863-Abstract.html](https://proceedings.neurips.cc/paper_files/paper/2012/hash/6aca97005c68f1206823815f66102863-Abstract.html).
- [10] HO, Qirong, James CIPAR, Henggang CUI, Seunghak LEE, Jin Kyu KIM, Phillip B. GIBBONS, Garth A GIBSON, Greg GANGER, and Eric P XING. More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server. In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2013. Available from [https://proceedings.neurips.cc/paper\\_files/paper/2013/hash/b7bb35b9c6ca2aee2df08cf09d7016c2-Abstract.html](https://proceedings.neurips.cc/paper_files/paper/2013/hash/b7bb35b9c6ca2aee2df08cf09d7016c2-Abstract.html).

- [11] CHILIMBI, Trishul, Yutaka SUZUE, Johnson APACIBLE, and Karthik KALYANARAMAN. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2014 . pp. 571–582. OSDI'14. ISBN 978-1-931971-16-4.
- [12] LI, Mu, David G. ANDERSEN, Jun Woo PARK, Alexander J. SMOLA, Amr AHMED, Vanja JOSIFOVSKI, James LONG, Eugene J. SHEKITA, and Bor-Yiing SU. Scaling Distributed Machine Learning with the Parameter Server. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, 2014 . pp. 583–598. ISBN 978-1-931971-16-4. Available from [https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li\\_mu](https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu).
- [13] IANDOLA, Forrest N., Khalid ASHRAF, Matthew W. MOSKEWICZ, and Kurt KEUTZER. FireCaffe: Near-Linear Acceleration of Deep Neural Network Training on Compute Clusters. *arXiv.org* [online]. 2016. Available from DOI 10.48550/arXiv.1511.00175.
- [14] MORITZ, Philipp, Robert NISHIHARA, Ion STOICA, and Michael I. JORDAN. SparkNet: Training Deep Networks in Spark. *arXiv.org* [online]. 2016. Available from DOI 10.48550/arXiv.1511.06051.
- [15] YOU, Yang, Zhao ZHANG, Cho-Jui HSIEH, James DEMMEL, and Kurt KEUTZER. ImageNet Training in Minutes. *arXiv.org* [online]. 2018. Available from DOI 10.48550/arXiv.1709.05011.
- [16] GOYAL, Priya, Piotr DOLLÁR, Ross GIRSHICK, Pieter NOORDHUIS, Lukasz WESOŁOWSKI, Aapo KYROLA, Andrew TULLOCH, Yangqing JIA, and Kaiming HE. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv.org* [online]. 2018. Available from DOI 10.48550/arXiv.1706.02677.
- [17] SMITH, Samuel L., Pieter-Jan KINDERMANS, Chris YING, and Quoc V. LE. Don't Decay the Learning Rate, Increase the Batch Size. *arXiv.org* [online]. 2018. Available from DOI 10.48550/arXiv.1711.00489.
- [18] POLLACK, Jordan B.. Recursive Distributed Representations. *Artificial Intelligence*. 1990, Vol. 46, No. 1, pp. 77–105. ISSN 0004-3702. Available from DOI 10.1016/0004-3702(90)90005-K.
- [19] GOLLER, Christoph, and Andreas KÜCHLER. Learning Task-Dependent Distributed Representations by Backpropagation through Structure. In: 1996 . pp. 347-352 vol.1. ISBN 978-0-7803-3210-2. Available from DOI 10.1109/ICNN.1996.548916.
- [20] BIANUCCI, Anna Maria, Alessio MICHELI, Alessandro SPERDUTI, and Antonina STARITA. Application of Cascade Correlation Networks for Structures to Chemistry. *Applied Intelligence*. 2000, Vol. 12, No. 1, pp. 117–147. ISSN 1573-7497. Available from DOI 10.1023/A:1008368105614.
- [21] O'SHEA, Keiron, and Ryan NASH. An Introduction to Convolutional Neural Networks. *arXiv.org* [online]. 2015. Available from DOI 10.48550/arXiv.1511.08458.
- [22] BRONSTEIN, Michael M., Joan BRUNA, Yann LECUN, Arthur SZLAM, and Pierre VANDERGHEYNST. Geometric Deep Learning: Going beyond Euclidean Data. *IEEE Signal Processing Magazine*. 2017, Vol. 34, No. 4, pp. 18–42. ISSN 1558-0792. Available from DOI 10.1109/MSP.2017.2693418.

- [23] FEY, Matthias, and Jan Eric LENSSEN. Fast Graph Representation Learning with PyTorch Geometric. *arXiv.org* [online]. 2019. Available from DOI 10.48550/arXiv.1903.02428.
- [24] SOUREK, Gustav, Vojtech ASCHENBRENNER, Filip ZELEZNY, and Ondrej KUZELKA. Lifted Relational Neural Networks. *arXiv.org* [online]. 2015. Available from DOI 10.48550/arXiv.1508.05128.
- [25] ŠÍR, Gustav. Deep Learning with Relational Logic Representations. In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 2019. pp. 6462–6463. Available from DOI 10.24963/ijcai.2019/920.
- [26] KRIZHEVSKY, Alex, Ilya SUTSKEVER, and Geoffrey E HINTON. ImageNet Classification with Deep Convolutional Neural Networks. In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2012. Available from [https://papers.nips.cc/paper\\_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html](https://papers.nips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html).
- [27] ABADI, Martín, Paul BARHAM, Jianmin CHEN, Zhifeng CHEN, Andy DAVIS, Jeffrey DEAN, Matthieu DEVIN, Sanjay GHEMAWAT, Geoffrey IRVING, Michael ISARD, Manjunath KUDLUR, Josh LEVENBERG, Rajat MONGA, Sherry MOORE, Derek G. MURRAY, Benoit STEINER, Paul TUCKER, Vijay VASUDEVAN, Pete WARDEN, Martin WICKE, Yuan YU, and Xiaoqiang ZHENG. TensorFlow: A System for Large-Scale Machine Learning. *arXiv.org* [online]. 2016. Available from DOI 10.48550/arXiv.1605.08695.
- [28] PASZKE, Adam, Sam GROSS, Francisco MASSA, Adam LERER, James BRADBURY, Gregory CHANAN, Trevor KILLEEN, Zeming LIN, Natalia GIMELSHEIN, Luca ANTIGA, Alban DESMAISON, Andreas KÖPF, Edward YANG, Zach DEVITO, Martin RAISON, Alykhan TEJANI, Sasank CHILAMKURTHY, Benoit STEINER, Lu FANG, Junjie BAI, and Soumith CHINTALA. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv.org* [online]. 2019. Available from DOI 10.48550/arXiv.1912.01703.
- [29] NAVEED, Humza, Asad Ullah KHAN, Shi QIU, Muhammad SAQIB, Saeed ANWAR, Muhammad USMAN, Naveed AKHTAR, Nick BARNES, and Ajmal MIAN. A Comprehensive Overview of Large Language Models. *arXiv.org* [online]. 2024. Available from DOI 10.48550/arXiv.2307.06435.
- [30] LIAO, Xia, Shengguo LI, Wei YU, and Yutong LU. Parallel Matrix Multiplication Algorithms in Supercomputing. In: *2021 6th International Conference on Intelligent Computing and Signal Processing (ICSP)*. 2021. pp. 1–4. Available from DOI 10.1109/ICSP51882.2021.9409013.
- [31] CHEN, Siyuan, Pratik Pramod FEGADE, Tianqi CHEN, Phillip GIBBONS, and Todd MOWRY. ED-Batch: Efficient Automatic Batching of Dynamic Neural Networks via Learned Finite State Machines. In: *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 2023. pp. 4514–4528. ISSN 2640-3498. Available from <https://proceedings.mlr.press/v202/chen23g.html>.
- [32] HELAL, Hatem, Jesun FIROZ, Jenna BILBREY, Mario Michael KRELL, Tom MURRAY, Ang LI, Sotiris XANTHEAS, and Sutanay CHOUDHURY. Extreme Acceleration of Graph Neural Network-based Prediction Models for Quantum Chemistry. *arXiv.org* [online]. 2022. Available from DOI 10.48550/arXiv.2211.13853.
- [33] BILBREY, Jenna A., Kristina M. HERMAN, Henry SPRUEILL, Soritis S. XANTHEAS, Payel DAS, Manuel Lopez ROLDAN, Mike KRAUS, Hatem HELAL, and Sutanay

- CHOUDHURY. Reducing Down(Stream)Time: Pretraining Molecular GNNs Using Heterogeneous AI Accelerators. *arXiv.org* [online]. 2022. Available from DOI 10.48550/arXiv.2211.04598.
- [34] CATTANEO, Alberto, Daniel JUSTUS, Harry MELLOR, Douglas ORR, Jerome MALOBERTI, Zhenying LIU, Thorin FARNSWORTH, Andrew FITZGIBBON, Blazej BANASZEWSKI, and Carlo LUSCHI. BESS: Balanced Entity Sampling and Sharing for Large-Scale Knowledge Graph Completion. *arXiv.org* [online]. 2022. Available from DOI 10.48550/arXiv.2211.12281.
- [35] MUGGLETON, Stephen. Inductive Logic Programming. *New Generation Computing*. 1991, Vol. 8, No. 4, pp. 295–318. ISSN 0288-3635. Available from DOI 10.1007/BF03037089.
- [36] MUGGLETON, Stephen, and Luc de RAEDT. Inductive Logic Programming: Theory and Methods. *The Journal of Logic Programming*. 1994, Vol. 19–20, pp. 629–679. ISSN 0743-1066. Available from DOI 10.1016/0743-1066(94)90035-3.
- [37] GETOOR, Lise, and Ben TASKAR. *Introduction to Statistical Relational Learning*. Cambridge, MA, US: MIT Press, 2007. ISBN 978-0-262-07288-5.
- [38] KIMMIG, Angelika, Lilyana MIHALKOVA, and Lise GETOOR. Lifted Graphical Models: A Survey. *Machine Learning*. 2015, Vol. 99, No. 1, pp. 1–45. ISSN 1573-0565. Available from DOI 10.1007/s10994-014-5443-2.
- [39] KRAMER, Stefan, Nada LAVRAČ, and Peter FLACH. *Propositionalization Approaches to Relational Data Mining*. Available from DOI 10.1007/978-3-662-04599-2\_11.
- [40] FRIEDMAN, Jerome H.. Greedy Function Approximation: A Gradient Boosting Machine.. *The Annals of Statistics*. Institute of Mathematical Statistics, 2001, Vol. 29, No. 5, pp. 1189–1232. ISSN 0090-5364, 2168-8966. Available from DOI 10.1214/aos/1013203451.
- [41] SHWARTZ-ZIV, Ravid, and Amitai ARMON. Tabular Data: Deep Learning Is Not All You Need. *Information Fusion*. 2022, Vol. 81, pp. 84–90. ISSN 1566-2535. Available from DOI 10.1016/j.inffus.2021.11.011.
- [42] THE SQLNET COMPANY GMBH. *getML*. [cit. 2024-05-05]. Available from <https://www.getml.com/>.
- [43] THE ALTERYX. *Deep Feature Synthesis: How Automated Feature Engineering Works – Alteryx | Innovation*. [cit. 2024-05-05]. Available from <https://innovation.alteryx.com/deep-feature-synthesis/>.
- [44] THE ALTERYX. *Featuretools*. [cit. 2024-05-05]. Available from <https://www.featuretools.com/>.
- [45] CRABBE, Jonathan, Zhaozhi QIAN, Fergus IMRIE, and Mihaela van der SCHAAR. Explaining Latent Representations with a Corpus of Examples. In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2021. pp. 12154–12166. Available from [https://proceedings.neurips.cc/paper\\_files/paper/2021/hash/65658fde58ab3c2b6e5132a39fae7cb9-Abstract.html](https://proceedings.neurips.cc/paper_files/paper/2021/hash/65658fde58ab3c2b6e5132a39fae7cb9-Abstract.html).
- [46] ZANIOLO, Carlo, Peter C. LOCKEMANN, Marc H. SCHOLL, and Torsten GRUST. *Advances in Database Technology - EDBT 2000: 7th International Conference on Extending Database Technology Konstanz, Germany, March 27-31, 2000 Proceedings*. Springer Science & Business Media, 2000. ISBN 978-3-540-67227-2.

- [47] KADRA, Arlind, Marius LINDAUER, Frank HUTTER, and Josif GRABOČKA. Well-Tuned Simple Nets Excel on Tabular Datasets. In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2021. pp. 23928–23941. Available from <https://proceedings.neurips.cc/paper/2021/hash/c902b497eb972281fb5b4e206db38ee6-Abstract.html>.
- [48] PYTORCH CONTRIBUTORS. *TorchScript — PyTorch 2.3 Documentation*. [cit. 2024-05-05]. Available from <https://pytorch.org/docs/stable/jit.html>.
- [49] SHANKAR, Asim, and Wolff DOBSON. *Eager Execution: An Imperative, Define-by-Run Interface to TensorFlow*. 2018-04-03. [cit. 2024-05-11]. Available from <http://research.google/blog/eager-execution-an-imperative-define-by-run-interface-to-tensorflow/>.
- [50] LOOKS, Moshe, Marcello HERRESHOFF, DeLesley HUTCHINS, and Peter NORVIG. Deep Learning with Dynamic Computation Graphs. *arXiv.org* [online]. 2017. Available from DOI 10.48550/arXiv.1702.02181.
- [51] NEUBIG, Graham, Chris DYER, Yoav GOLDBERG, Austin MATTHEWS, Waleed AMMAR, Antonios ANASTASOPOULOS, Miguel BALLESTEROS, David CHIANG, Daniel CLOTHIAUX, Trevor COHN, Kevin DUH, Manaal FARUQUI, Cynthia GAN, Dan GARRETTE, Yangfeng Ji, Lingpeng KONG, Adhiguna KUNCORO, Gaurav KUMAR, Chaitanya MALAVIYA, Paul MICHEL, Yusuke ODA, Matthew RICHARDSON, Naomi SAPHRA, Swabha SWAYAMDIPTA, and Pengcheng YIN. DyNet: The Dynamic Neural Network Toolkit. *arXiv.org* [online]. 2017. Available from DOI 10.48550/arXiv.1701.03980.
- [52] NEUBIG, Graham, Yoav GOLDBERG, and Chris DYER. On-the-Fly Operation Batching in Dynamic Computation Graphs. *arXiv.org* [online]. 2017. Available from DOI 10.48550/arXiv.1705.07860.
- [53] XU, Shizhen, Hao ZHANG, Graham NEUBIG, Wei DAI, Jin Kyu KIM, Zhijie DENG, Qirong HO, Guangwen YANG, and Eric P. XING. Cavs: An Efficient Runtime System for Dynamic Neural Networks. In: 2018. pp. 937–950. ISBN 978-1-939133-01-4. Available from <https://www.usenix.org/conference/atc18/presentation/xu-shizhen>.
- [54] ZHA, Sheng, Ziheng JIANG, Haibin LIN, and Zhi ZHANG. Just-in-Time Dynamic-Batching. *arXiv.org* [online]. 2019. Available from DOI 10.48550/arXiv.1904.07421.
- [55] FEGADE, Pratik. *Auto-Batching Techniques for Dynamic Deep Learning Computation*. 2023. Dissertation. Available from [https://kilthub.cmu.edu/articles/thesis/Auto-batching\\_Techniques\\_for\\_Dynamic\\_Deep\\_Learning\\_Computation/21859902/1](https://kilthub.cmu.edu/articles/thesis/Auto-batching_Techniques_for_Dynamic_Deep_Learning_Computation/21859902/1).
- [56] KIPF, Thomas N., and Max WELLING. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv.org* [online]. 2017. Available from DOI 10.48550/arXiv.1609.02907.
- [57] HAMILTON, William L., Rex YING, and Jure LESKOVEC. Inductive Representation Learning on Large Graphs. *arXiv.org* [online]. 2018. Available from DOI 10.48550/arXiv.1706.02216.
- [58] VELIČKOVIĆ, Petar, Guillem CUCURULL, Arantxa CASANOVA, Adriana ROMERO, Pietro LIÒ, and Yoshua BENGIO. Graph Attention Networks. *arXiv.org* [online]. 2018. Available from DOI 10.48550/arXiv.1710.10903.

- [59] DU, Jian, Shanghang ZHANG, Guanhang WU, Jose M. F. MOURA, and Soumya KAR. Topology Adaptive Graph Convolutional Networks. *arXiv.org* [online]. 2018. Available from DOI 10.48550/arXiv.1710.10370.
- [60] XU, Keyulu, Weihua HU, Jure LESKOVEC, and Stefanie JEGELKA. How Powerful Are Graph Neural Networks?. *arXiv.org* [online]. 2019. Available from DOI 10.48550/arXiv.1810.00826.
- [61] WANG, Yue, Yongbin SUN, Ziwei LIU, Sanjay E. SARMA, Michael M. BRONSTEIN, and Justin M. SOLOMON. Dynamic Graph CNN for Learning on Point Clouds. *arXiv.org* [online]. 2019. Available from DOI 10.48550/arXiv.1801.07829.
- [62] CORSO, Gabriele, Luca CAVALLERI, Dominique BEAINI, Pietro LIÒ, and Petar VELIČKOVIĆ. Principal Neighbourhood Aggregation for Graph Nets. *arXiv.org* [online]. 2020. Available from DOI 10.48550/arXiv.2004.05718.
- [63] GRAPHCORE LTD. *IPU Hardware Overview — IPU Programmer’s Guide*. [cit. 2024-05-09]. Available from [https://docs.graphcore.ai/projects/ipu-programmers-guide/en/latest/about\\_ipu.html](https://docs.graphcore.ai/projects/ipu-programmers-guide/en/latest/about_ipu.html).
- [64] SCHLICHTKRULL, Michael, Thomas N. KIPF, Peter BLOEM, Rianne van den BERG, Ivan TITOV, and Max WELLING. Modeling Relational Data with Graph Convolutional Networks. In: Aldo GANGEMI, Roberto NAVIGLI, Maria-Esther VIDAL, Pascal HITZLER, Raphaël TRONCY, Laura HOLLINK, Anna TORDAI, and Mehwish ALAM, eds. *The Semantic Web*. Springer International Publishing, 2018 . pp. 593–607. ISBN 978-3-319-93417-4. Available from DOI 10.1007/978-3-319-93417-4\_38.
- [65] CVITKOVIC, Milan. Supervised Learning on Relational Databases with Graph Neural Networks. *arXiv.org* [online]. 2020. Available from DOI 10.48550/arXiv.2002.02046.
- [66] ZAHRADNÍK, Lukáš, Jan NEUMANN, and Gustav ŠÍR. A Deep Learning Blueprint for Relational Databases. In: *NeurIPS 2023 Second Table Representation Learning Workshop*. 2023. Available from <https://openreview.net/forum?id=b4GEmjshAB>.
- [67] FEY, Matthias, Weihua HU, Kexin HUANG, Jan Eric LENSSEN, Rishabh RANJAN, Joshua ROBINSON, Rex YING, Jiaxuan YOU, and Jure LESKOVEC. Relational Deep Learning: Graph Representation Learning on Relational Databases. *arXiv.org* [online]. 2023. Available from DOI 10.48550/arXiv.2312.04615.
- [68] ŠÍR, Gustav. *Towards Deep Learning for Relational Databases*. [cit. 2024-05-06]. Available from <https://towardsdatascience.com/towards-deep-learning-for-relational-databases-de9adce5bb00>.
- [69] VASWANI, Ashish, Noam SHAZEER, Niki PARMAR, Jakob USZKOREIT, Llion JONES, Aidan N GOMEZ, Łukasz KAISER, and Illia POLOSUKHIN. Attention Is All You Need. In: *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017. Available from [https://papers.nips.cc/paper\\_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html](https://papers.nips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html).
- [70] SOUREK, Gustav, Filip ZELEZNY, and Ondrej KUZELKA. Beyond Graph Neural Networks with Lifted Relational Neural Networks. *Machine Learning*. 2021, Vol. 110, No. 7, pp. 1695–1738. ISSN 0885-6125, 1573-0565. Available from DOI 10.1007/s10994-021-06017-3.
- [71] ZAHRADNÍK, Lukáš. *LukasZahradnik/PyNeuraLogic*. [cit. 2024-05-06]. Available from <https://github.com/LukasZahradnik/PyNeuraLogic>.

- [72] WILSON, D.Randall, and Tony R. MARTINEZ. The General Inefficiency of Batch Training for Gradient Descent Learning. *Neural Networks*. 2003, Vol. 16, No. 10, pp. 1429–1451. ISSN 08936080. Available from DOI 10.1016/S0893-6080(03)00138-2.
- [73] MASTERS, Dominic, and Carlo LUSCHI. Revisiting Small Batch Training for Deep Neural Networks. *arXiv.org* [online]. 2018. Available from DOI 10.48550/arXiv.1804.07612.
- [74] ZAHRADNÍK, Lukáš. *PyNeuraLogic — PyNeuraLogic Documentation*. [cit. 2024-05-14]. Available from <https://pyneuralogic.readthedocs.io/en/latest/index.html>.
- [75] HARRIS, Charles R., K. Jarrod MILLMAN, Stéfan J. van der WALT, Ralf GOMMERS, Pauli VIRTANEN, David COURNAPEAU, Eric WIESER, Julian TAYLOR, Sebastian BERG, Nathaniel J. SMITH, Robert KERN, Matti PICUS, Stephan HOYER, Marten H. van KERKWIJK, Matthew BRETT, Allan HALDANE, Jaime Fernández del RÍO, Mark WIEBE, Pearu PETERSON, Pierre GÉRARD-MARCHANT, Kevin SHEPPARD, Tyler REDDY, Warren WECKESSER, Hameer ABBASI, Christoph GOHLKE, and Travis E. OLIPHANT. Array Programming with NumPy. *Nature*. Nature Publishing Group, 2020, Vol. 585, No. 7825, pp. 357–362. ISSN 1476-4687. Available from DOI 10.1038/s41586-020-2649-2.
- [76] *NumPy*. [cit. 2024-05-07]. Available from <https://numpy.org/>.
- [77] PYTORCH CONTRIBUTORS. *Torch.Sparse — PyTorch 2.3 Documentation*. [cit. 2024-05-14]. Available from <https://pytorch.org/docs/stable/sparse.html>.
- [78] YAMAGUCHI, Takuma, and Federico BUSATO. *Accelerating Matrix Multiplication with Block Sparse Format and NVIDIA Tensor Cores*. 2021-03-19. [cit. 2024-05-14]. Available from <https://developer.nvidia.com/blog/accelerating-matrix-multiplication-with-block-sparse-format-and-nvidia-tensor-cores/>.
- [79] TENSORFLOW DEVELOPERS. *TensorFlow*. [cit. 2024-05-14]. Available from DOI 10.5281/zenodo.10798587.
- [80] FEY, Matthias. *rusty1s/pytorch\_scatter*. [cit. 2024-05-14]. Available from [https://github.com/rusty1s/pytorch\\_scatter](https://github.com/rusty1s/pytorch_scatter).
- [81] BUSA-FEKETE, Róbert, András KOCSOR, and Sándor PONGOR. *Tree-Based Algorithms for Protein Classification*. Available from DOI 10.1007/978-3-540-76803-6\_6.
- [82] ULLMAN, Jeffrey D.. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1989 . ISBN 0-7167-8162-X.
- [83] BRATKO, Ivan. *Prolog Programming for Artificial Intelligence*. Pearson Education, 2001 . ISBN 978-0-201-40375-6.
- [84] CHAMBERLIN, Donald D, and Raymond F BOYCE. SEQUEL: A Structured English Query Language. In: *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. 1974 . pp. 249–264.
- [85] MERRIAM WEBSTER, INCORPORATED. *The Verbing Adventures of 'Weigh' and 'Weight'*. 2019-01-16. [cit. 2024-05-16]. Available from <https://www.merriam-webster.com/wordplay/when-to-use-weigh-and-weight-as-a-verb>.
- [86] ŠÍR, Gustav. *GustikS/NeuraLogic*. [cit. 2024-05-16]. Available from <https://github.com/GustikS/NeuraLogic>.



- 
- [87] SOUREK, Gustav, Filip ZELEDNY, and Ondrej KUZELKA. Lossless Compression of Structured Convolutional Models via Lifting. In: 2020. Available from <https://openreview.net/forum?id=oxnp2q-PGL4>.
- [88] SIEK, Jeremy G.. *Essentials of Compilation: An Incremental Approach in Racket*. MIT Press, 2023 . ISBN 978-0-262-04776-0.
- [89] MORRIS, Christopher, Nils M. KRIEGE, Franka BAUSE, Kristian KERSTING, Petra MUTZEL, and Marion NEUMANN. TUDataset: A Collection of Benchmark Datasets for Learning with Graphs. *arXiv.org* [online]. 2020. Available from DOI 10.48550/arXiv.2007.08663.
- [90] PYG TEAM. *Datasets — PyTorch Geometric Documentation*. [cit. 2024-05-23]. Available from <https://pytorch-geometric.readthedocs.io/en/latest/modules/datasets.html>.
- [91] MOTL, Jan, and Oliver SCHULTE. The CTU Prague Relational Learning Repository. *arXiv.org* [online]. 2024. Available from DOI 10.48550/arXiv.1511.03086.