University of Oxford

Department of Computer Science

Hardware-Centered Graph Rewiring

Marcel Heshmati Rød

Balliol College

supervised by Prof. Michael Bronstein

A dissertation submitted in partial completion of the

MSc in Advanced Computer Science

Trinity 2022

1 Introduction

With recents advances in deep learning, several domains have come "reasonably close" to being solved by various state-of-the-art architectures, in the sense that given a sufficient amount of training data the models are by far the best existing methods for most tasks on the domains. For instance, for images, deep convolutional neural architectures [12] have long been the most successful, only recently being overtaken by the more flexible and scalable vision transformers [8], arguably surpassing human ability in vision tasks. Similarly, transformers [18] have been wildly successful on natural language tasks [3]. The successes of these models have enabled applications anywhere they could be applicable. For instance, image models have enabled thousands of applications, like reliable methods for facial recognition and climate modeling. This success is not only due to the architectures being better suited to learning on their respective domains, but also due co-evolved hardware advancements.

Learning on graphs is in many ways a much harder problem than for images and text, the latter tasks being special cases of the former [2]. The state-of-the-art methods for learning on Many current neural architectures have issues performing on

1.1 Motivation

In current paradigms, standard approaches to constructing graph neural networks exhibit some properties that are exceptionally well evidenced, yet not entirely understood. The problems of underreaching [1], oversmoothing [5] and oversquashing [1, 15] show up repeatedly in literature about issues with Graph Neural Networks. A key solution to underreaching and oversquashing is rewiring, but as we will see in further background studies, rewiring is highly non-trivial for several reasons and does not come without losses in other areas. Arbitrary rewiring may contribute significantly to the performance of the neural network, yet it often introduces a substantial amount of computational cost.

1.2 Achievements

1. Definition and formalization Gradient Flow Framework (GRAFF) TODO: Is this different/new from the GRAFF paper?
2. Perform analyses of the behavior of graph neural networks that can be described in terms of GRAFF TODO: are there novel results?
3. Construct new extensions to GRAFF, allowing any generic energy function ε to govern the flow of a graph neural network.
4. Define a fuzzy clustering energy function that incentivizes graph rewiring that performs well on specific hardware architectures.
5. Defining a metric and showing the robustness of this method for finding optimal rewirings.

2 Background

2.1 Diffusion and Partial Differential Equations

A partial differential equation (PDE) is a set of constraints on the partial derivatives of an underlying function, often parameterized in terms of a different function. PDEs often take the form of expressions using their nth order partial derivatives, with some examples following in the table below. TODO: table of a few well known PDEs The process of solving a PDE is "what are the functions that satisfy these constraints?" The heat equation, also known as the diffusion equation, is a classic example of a PDE that shows up in many domains, and will be used in the we will study in this thesis. Developed by Joseph Fourier in the early 18th century [13], the purpose of the differential equation was to study the transfer of heat, but it has found uses in a broad spectrum of applications, e.g. studying particle dynamics, quantum mechanics and economics. Importantly,

A somewhat general yet intuitive description of diffusion would be the flow of "substance" from high levels to lower levels at a rate that is proportional to the gradient of the level of substance at a given area (for the heat equation, substitute substance for temperature, and flow of substance for heat). The basic equation that expresses this is somewhat non-obvious to those who haven't studied it, written as

$$\frac{\partial u}{\partial t} = \alpha \Delta u, \tag{2.1}$$

for our "substance" function $u(x, t)$, (thermal) diffusivity constant α, where $\Delta = \nabla^2$ is the Laplacian. To understand why the expression uses the Laplacian, an operator recovering the curvature of a function, consider the fact that a point at position x at time $t + \epsilon$ for sufficiently small $\epsilon > 0$ will be moved if its neighboring points are higher than itself is. If the gradient is positive in one direction, the flow of heat could be cancelled by a negative gradient in a different direction. We need to measure the differences in gradients around the point x at time t to find out the "substance" change per unit time at instant t.

The description in words above is formulated in Proposition 2.1.1:

Proposition 2.1.1. *In the diffusion equation, the function value $u(x,t)$ at position x will change over time t toward the values of its surroundings.*

Equation 2.1 belies another intuition for what the heat equation means:

Proposition 2.1.2. *In the diffusion equation, function values change to minimize their spatial curvature.*

Proposition 2.1.1 follows closely the standard interpretation of the heat equation. This can be thought of heat or particles flowing from positions where there is more to fill positions which have less. In the absence of sources and drains of heat or particles (referred to as homogeneous systems), we have

Instead, Proposition 2.1.2 can be thought of as minimization of curvature. In this interpretation we are moving each point on the function with a rate proportional to its curvature. A property of the solutions to this differential equation is much more obvious here than in the other interpretation. Namely: when the curvature is zero, ergo the function is locally linear, the function value will stay constant.

2.1.1 Numerical Solutions to Partial Differential Equations

There are several ways of solving PDEs, but central to this thesis is Runge-Kutta (RK) methods, the simplest of which being Euler's method.

Given an initial value problem of the form

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(t,y), \quad y(t_0) = y_0, \tag{2.2}$$

we

Simple Solvers

TODO: Write up how simple PDEs can be solved

Stiff Equations

The methods described above work well for simple functions, but may fail spectacularly when solving PDEs that are expressed in complex functions f, for instance when parameterized by a neural network, a case we will come back to in subsection 2.1.2 as well as in section 3.1 and section 3.2.

Adaptive Methods and Tricks for Stiff Equations

The implementation (see TODO: link section on solving the solver), the inner workings of these methods is beyond the scope of this thesis.

In particular use of higher order adaptive [14] RK methods like Tsitouras 5(4) [17] and Dormand Prince 5(4) [7]

2.1.2 Neural Differential Equations

In his thesis on the matter [11], Patrick Kidger concisely defines a neural differential equation as "a differential equation using a neural network to parameterize the vector field". The canonical example of this is the standard neural ODE

$$y(0) = y_0 \quad \frac{\mathrm{d}y}{\mathrm{d}t}(t) = f_\theta(t, y(t)). \tag{2.3}$$

In this equation, θ represents the weights of the flow field neural network $f_\theta : \mathbb{R} \times \mathbb{R}^{d_1 \times \cdots \times d_k} \to \mathbb{R}^{d_1 \times \cdots \times d_k}$.

2.2 Graph Neural Networks

Graph Neural Network (GNN), is a collective term for neural network architectures that run on graphs as their input data.

2.2.1 Graph Learning Tasks

Using graph input, one may consider several prediction tasks. Often, these are separated into which aspects of the graph they are trying to make predictions about. We have

- Node-level tasks – Predict some features per node on the input graph. Often this would be classification of the node features, for instance classifying fake accounts in a social media graph. If we reorder the nodes (the order in which they are organized in the input representation), we expect reordered outputs.
- Edge-level tasks – Predict features on existing or potential edges on the input graph. This might involve which edges might be missing, or the strength of a relationship. Again, reordering the input graph should appropriately reorder the output of these predictions.
- Graph-level tasks – Predict features on the entire graph. If our graph is a molecule, we could for instance try to predict material properties. Reordering the nodes and edges should in this instance have no effect on our prediction.

2.3 Message Passing Neural Networks

Message passing neural networks [9] (MPNNs) are a subclass of graph neural networks that conform to the following equations

$$m_v^{t+1} = \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw}) \tag{2.4}$$

$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1}), \tag{2.5}$$

$$\tag{2.6}$$

where m_v^{t+1} is the message computed for timestep $t+1$ at node v, $N(v)$ is the set of neighbors of v, h_x^t is the state of node x at timestep t and e_{vw} is the edge weight from node v to node w. M_t and U_t are the message function and the node update

function at time t respectively, and this time variable is usually used as an index for the current layer of a feed-forward neural network.

One could also consider global tasks on

2.3.1 Graph Data and Datasets

Unlike images or time series data, connections and thus (first order) interactions between features in an input graph are not fixed between graphs, nor are they arbitrary.

3 Method

3.1 Graph Neural Diffusion

Graph Neural Diffusion (GRAND) [4] is a method that sees graph neural networks as a discretization of an underlying PDE.

The diffusion equation applied to graphs can be written as

$$\frac{\partial u(x,t)}{\partial t} = \div[\boldsymbol{G}(u(x,t),t)\nabla_x u(x,t)] \tag{3.1}$$

TODO: Some detail here, not too much to cover.

3.2 The Gradient Flow Framework

The Gradient Flow Framework (GRAFF), was introduced in [6] as a way to study and modify the properties of MPNNs. TODO: This will be rather detailed

3.3 Machine Learning Hardware Architectures

A major factor to the success of the great machine learning algorithms that are currently in use is their computational efficiency. Operations like the convolution and matrix multiplies are highly compatible with hardware architectures that are already in use for other reasons. Graphics processing units (GPUs) have been used as hardware accelerators for decades, and their massively parallel nature means that with few tweaks they are very capable of computing the transformations necessary for these machine learning processes. The static relationship between input and output leads to a linear relationship between parallel compute capability and compute performance

3.3.1 GNNs on Hardware

Graph neural networks are different, however, as the interactions between nodes are sparse and sporadic, defined by the edges of the graph between arbitrary node indices. While

this *can* be expressed in the form of matrix multiplication on dense layers, this leads to incredibly sparse tensors and wasted computation while adding zeros. In the message passing framework defined in section 2.3, we can deal with the sparsity of operations by *scattering*. Recall the MPNN equation defined in Equation 2.4 and Equation 2.5.

$$m_v^{t+1} = \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw}) \qquad \text{(2.4 revisited)}$$

$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1}), \qquad \text{(2.5 revisited)}$$

First, organize the data in a timestep into dense representations: Assume any edge in the graph must have edge features $e_{vw} \neq 0$. We can then define our adjacency matrix

We can isolate the sparse operation in the message passing operation happening in Equation 2.5 by factoring the

Representing the edge matrix as a sparse tensor using sparse operations This shows up as the `scatter` operation in several frameworks [16, 10], which is the key operation to implementing most graph neural network architectures. Using the code in section A.1

Now that

The accessible

3.4 Inference Runtime Optimization

Using GRAFF we can optimize for features other than those that can be defined in a loss function

3.4.1 Restricting Rewiring With an Energy Function

3.4.2 Fuzzy Clustering

In the algoritm described above we rely on a method of clustering the points defined by our positional encodings . A further discussion of the runtime complexity and technical challenges of the fuzzy clustering algorithm can be found in subsection 4.2.1.

Annealing to the Result

TODO: Add analysis of the annealing process (making the communication boundaries harder and harder during training time). TODO: Explain the scheduling of this parameter Jax somewhere else, similar to LR scheduling

3.4.3 Reordering

As mentioned in subsection 3.3.1, arbitrary node ordering can lead to sparse matrices with poor access patterns for both reading and writing during compute. Therefore, we apply a permutation matrix P to the input, and since the graph neural network is node permutation equivariant, we can invert the permutation afterward by applying its transpose P^T (since $P^\mathsf{T}P = I, P^{-1} = P^\mathsf{T}$).

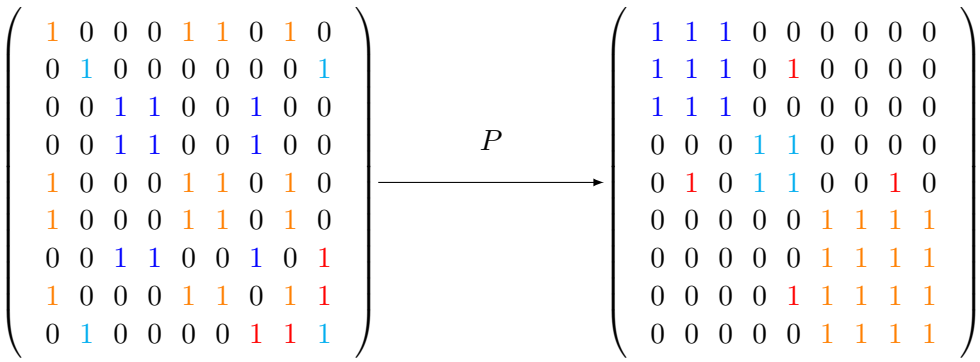

Figure 3.1: Even after clustering has finished, the adjacency matrix is unpredictable. Traversing the neighbors of a node in the first matrix would require looking at values that are sparsely distributed either when reading or when writing. Reordering the nodes results in an almost block diagonal adjacency structure, enabling local reads and writes. Connections between blocks will still require sparse compute, but the list of them is substantially shorter, and so the interactions can be computed in parallel at no extra time cost.

3.5 Evaluation

Following the description of

3.5.1 Hardware Modelling

4 Implementation

When implementing the

4.1 Frameworks

4.1.1 Choosing a framework

There are several possible choices for a method for

4.1.2 Jax

Using Jax presents several advantages for the implementation of the neural architecture. While in some aspects the tooling is immature, the extensibility and raw speed of Jax has allowed for well optimized methods that are not easily available even in the more mature PyTorch.

Equinox

TODO: Explain why Kidger's way of doing it works and what the consequences are of chosing Equinox vs something Flax/Haiku/Objax

Diffrax

TODO: Go through some key features of Diffrax and why the solvers are good and fast. I have some understanding of the method of implementation as well, so might bring that up?

4.2 Energy Functions

As described in our method

4.2.1 Fuzzy Clustering

The fuzzy clustering algorithm defined in subsection 3.4.2 is implemented as an energy function...

Clustering on highly parallel hardware like a GPU or TPU TODO: explain the runtime and space complexity of the method in use, and why KDTrees are theoretically optimal, but not in use

4.3 Gradient Flow

TODO: Explain how the gradient flow works and how the solver is in use

4.4 Datasets

4.4.1 Standard Datasets

In order to get some idea of how this method performs on real datasets

TODO: Run through all the datasets and what they contain, expected results, challenges, heterophily/homophily

4.4.2 Synthetic Datasets

In addition, we construct some ideal and some adverserial datasets that show best and worst-case situations. In chapter 6 there's some in-depth discussion on results on these datasets and what they mean for the model as a whole.

The HomophilicAberration dataset ... TODO: fill in details on how each dataset is constructed and motivate the construction

TODO: Should probably also quickly implement other GNN architectures to be sure that I'm comparing apples to apples

TODO: Maybe move this to method?

5 Results

5.1 Optimizing for Runtime Costs

We train using the method with

5.1.1 Reducing Scattered Compute

As argued in
TODO: Present the results from changing from scattered compute to locally dense (block diagonal structure).

5.1.2 Trade-off Analysis

Table 5.1: Comparing compute time on CPU and GPU using

6 Discussion

6.1 Synthetic Datasets

A Implementation Snippets

While the code is available in the repository on GitHub, I have included some snippets of impelmentation in this appendix for reference in the rest of the thesis. Some sections of code have been simplified in order to improve understandability, and for the full implementation please see the GitHub repository.

A.1 Benchmarking Scattered and Dense Operations

```python
@jax.jit
def graph_cnn_operation(x_in, A, S, edge_list):
    # x_in: (n_nodes, n_features)
    # edge_list: (n_edges, 2)
    messages = x_in @ A
    self_messages = x_in @ S

    x_out = self_messages.at[edge_list[1], :].add(messages[edge_list[0], :])
    return x_out
```

Listing 1: First listing

TODO: There is more code than just that for the benchmarking...

Bibliography

[1] U. Alon and E. Yahav. *On the Bottleneck of Graph Neural Networks and Its Practical Implications*. 9th Mar. 2021. DOI: 10.48550/arXiv.2006.05205. arXiv: 2006.05205 [cs, stat]. URL: http://arxiv.org/abs/2006.05205 (visited on 22/09/2022).

[2] M. M. Bronstein et al. *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges*. 2nd May 2021. DOI: 10.48550/arXiv.2104.13478. arXiv: 2104.13478 [cs, stat]. URL: http://arxiv.org/abs/2104.13478 (visited on 22/09/2022).

[3] T. B. Brown et al. *Language Models Are Few-Shot Learners*. 22nd July 2020. DOI: 10.48550/arXiv.2005.14165. arXiv: 2005.14165 [cs]. URL: http://arxiv.org/abs/2005.14165 (visited on 22/09/2022).

[4] B. P. Chamberlain et al. 'GRAND: Graph Neural Diffusion'. 22nd Sept. 2021. arXiv: 2106.10934 [cs, stat]. URL: http://arxiv.org/abs/2106.10934 (visited on 24/04/2022).

[5] D. Chen et al. *Measuring and Relieving the Over-smoothing Problem for Graph Neural Networks from the Topological View*. 18th Nov. 2019. arXiv: 1909.03211 [cs, stat]. URL: http://arxiv.org/abs/1909.03211 (visited on 22/09/2022).

[6] F. Di Giovanni et al. *Graph Neural Networks as Gradient Flows*. 15th Aug. 2022. DOI: 10.48550/arXiv.2206.10991. arXiv: 2206.10991 [cs, stat]. URL: http://arxiv.org/abs/2206.10991 (visited on 18/08/2022).

[7] J. R. Dormand and P. J. Prince. 'A Family of Embedded Runge-Kutta Formulae'. In: *Journal of Computational and Applied Mathematics* 6.1 (1st Mar. 1980), pp. 19–26. ISSN: 0377-0427. DOI: 10.1016/0771-050X(80)90013-3. URL: https://www.sciencedirect.com/science/article/pii/0771050X80900133 (visited on 11/09/2022).

[8] A. Dosovitskiy et al. *An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 3rd June 2021. DOI: 10.48550/arXiv.2010.11929. arXiv: 2010.11929 [cs]. URL: http://arxiv.org/abs/2010.11929 (visited on 22/09/2022).

[9] J. Gilmer et al. *Neural Message Passing for Quantum Chemistry*. 12th June 2017. DOI: 10.48550/arXiv.1704.01212. arXiv: 1704.01212 [cs]. URL: http://arxiv.org/abs/1704.01212 (visited on 22/09/2022).

[10] *Jax.Lax.Scatter — JAX Documentation*. URL: https://jax.readthedocs.io/en/latest/_autosummary/jax.lax.scatter.html (visited on 23/09/2022).

[11] P. Kidger. 'On Neural Differential Equations'. 4th Feb. 2022. arXiv: 2202.02435 [cs, math, stat]. URL: http://arxiv.org/abs/2202.02435 (visited on 31/03/2022).

[12] Y. Lecun et al. 'Gradient-Based Learning Applied to Document Recognition'. In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 1558-2256. DOI: 10.1109/5.726791.

[13] T. N. Narasimhan. 'Fourier's Heat Conduction Equation: History, Influence, and Connections'. In: *Proceedings of the Indian Academy of Sciences - Earth and Planetary Sciences* 108.3 (1st Sept. 1999), pp. 117–148. ISSN: 0973-774X. DOI: 10.1007/BF02842327. URL: https://doi.org/10.1007/BF02842327 (visited on 26/09/2022).

[14] W. H. Press and S. A. Teukolsky. 'Adaptive Stepsize Runge-Kutta Integration'. In: *Computers in Physics* 6.2 (1992), p. 188. ISSN: 08941866. DOI: 10.1063/1.4823060. URL: http://scitation.aip.org/content/aip/journal/cip/6/2/10.1063/1.4823060 (visited on 11/09/2022).

[15] J. Topping et al. 'Understanding Over-Squashing and Bottlenecks on Graphs via Curvature'. 16th Mar. 2022. arXiv: 2111.14522 [cs, stat]. URL: http://arxiv.org/abs/2111.14522 (visited on 04/04/2022).

[16] *Torch.Tensor.Scatter_ — PyTorch 1.12 Documentation*. URL: https://pytorch.org/docs/stable/generated/torch.Tensor.scatter_.html#torch.Tensor.scatter_ (visited on 23/09/2022).

[17] C. Tsitouras. 'Runge–Kutta Pairs of Order 5(4) Satisfying Only the First Column Simplifying Assumption'. In: *Computers & Mathematics with Applications* 62 (1st July 2011), pp. 770–775. DOI: 10.1016/j.camwa.2011.06.002.

[18] A. Vaswani et al. *Attention Is All You Need*. 5th Dec. 2017. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762 [cs]. URL: http://arxiv.org/abs/1706.03762 (visited on 22/09/2022).