High Performance Computing on Intelligence Processing Units

Accelerator Performance Analysis on Spatio-Temporal Graph Convolutional Networks

Johannes Ø. Moe

Thesis submitted for the degree of
Master in Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2021

High Performance Computing on Intelligence Processing Units

Accelerator Performance Analysis on Spatio-Temporal Graph Convolutional Networks

Johannes Ø. Moe

Acknowledgements

I want to extend my deepest gratitude to the people and institutions that made this possible.

First I want to thank Johannes Langguth, Konstantin Pogerelov, and Xing Cai for being my supervisors for this project, and the input and guidance they have extended to me throughout.

I would like to thank the Institute of Informatics at the University of Oslo and Simula Research Laboratories for the opportunity to work on such an exiting project.

I want to thank Graphcore for the support they made available to me when working on their technology[1].

My parents, Hilde Moe and Jens Erik Østenby, also need to be thanked for the love and support I have continuously received from them.

Finally I would like to show immense appreciation to Benedikte Wallace, Emma Stensby Norstein, and Frank Veenstra. What sanity survived through writing my thesis during the lockdown I can attribute to them, and for that I am eternally grateful.

[1]I would also like to thank Graphcore for the job.

Abstract

The feed-forward Artificial Neural Network has been used for a multitude of regression tasks, and its descendants have expanded the domain to (amongst others) image [28] and speech[4] [5] recognition, filtering social networks[13], and machine translation[37][51].

While conventional artificial neural networks (ANNs) and their variations[2] work well on data represented in euclidean space [3] (images, vectors, matrices, etc.), they struggle to work on data in non-euclidean space. Graph Neural Networks (GNNs) expand Recurrent Neural Networks (RNN) [52] [47] to directly process graphs in the form of adjacency matrices.

The computational complexity of GNN's makes them ill fitted for conventional Graph Processing Units (GPUs), the purpose of this thesis is to investigate the viability of a new Artificial Intelligence Accelerator: The Intelligence Processing Unit.

[2]f.ex.: Convolution [31], Recurrent [21], etc.

[3]Standard N-dimensional space.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

1.1 Background and Motivation

As Artificial Neural Networks (ANNs) and Graph Neural Networks (GNNs) grow in complexity, the duration of training networks increases. As the processing time increases, the necessity for better High Performance Computing (HPC) accelerators becomes increasingly important.

Modern machine learning algorithms trained by Reinforcement Learning use Graphical Processing Units (GPUs) to perform mass throughput data processing.

The GPU architecture is particularly suited for image processing and dense matrix operations on conventionally structured data sets. However, it is not particularly suited for graph data, or sparse data sets.

The Intelligence Processing Unit (IPU) is a state-of-the-art High Performance Computing processor for AI acceleration produced by Graphcore. Mounted in sets of four in the m2000 Machine, the IPU is set to promise built-in scalability for processing extremely large parallel processing workloads.

While GPUs are now matured as a product, the IPU is still very new, the first machine released in 2018 and its second in 2020.

To analyse the IPU and its supporting libraries, I intend to implement an existing Graph Neural Network: The Spatio-Temporal Graph Convolutional Network [56], in the coding language C++ on the IPU. With the results from this project, I hope to measure the performance of the IPU Mk. 2 as an AI processor, discern the maturity of the Graphcore libraries and the IPU's supporting software, and ultimately analyse the IPU's capabilities as an accelerator.

1.2 Problem Statement

The requirements of high performance accelerators for AI have increased continuously as machine learning algorithms become more complex. Where historically machine learning algorithms have been employed Graph Processing Units, the budding frontier of Graph Neural Networks and their unpredictable core-to-core behaviour requires increased computational flexibility.

The Intelligence Processing Unit (IPU) promises much higher computational flexibility due to its design. Does the IPU outperform the GPU for these tasks?

1.3 Research Question

It is the intention of this thesis to investigate the promise of Graphcore's Intelligence Processing Unit (IPU) as an alternative machine learning accelerator for the Graph Neural Network Domain.

The approach is to implement the Spatio-Temporal Graph Neural Network (STGCN) on the IPU in Graphcore's Poplar API and comparing it to the original STGCN trained on a GPU.

1.4 Contributions

The contributions of this thesis entail the completed and verified forward implementation of the Spatio-Temporal Graph Neural Network, a customized implementation of the backward pass of the same algorithm, and a comparison between the historic STGCN on the GPU and the produced STGCN for the IPU. It will be shown that the IPU delivers on the promised computational power.

1.5 Outline

This thesis has 11 chapters divided into 3 parts:

Part I: Introduction After the introductory chapter follows chapter 2: delving into the background of graph neural networks and their deeper motivations, the most main models, and the different challenges inherent to them; chapter 3: Inspecting the Intelligence Processing Unit, its own motivations and challenges; and chapter 4: the related works.

Part II: The Project This part details the project itself. In chapter 5 there is a thorough walk-through of the Spatio-Temporal Graph Neural Network; chapter 6 details the projected planning of the project, details on occurring issues, and motivation for the approach decided upon; chapter 7 describes the implemented model, details on the code, and the implemented structures; and finally in chapter 8 the results are represented.

Part III: Discussion Chapter 9 details observations in regards to the implementation, detailing four of the main takeaways from the work itself. Chapter 10 focuses on comments on the technology (software and hardware). Finally chapter 11 is the conclusion of the work.

Chapter 2

Background I: Graph Neural Networks

2.1 Motivation

While tasks dependent on euclidean data (f.ex. image processing) have been the focus of academic research, in no small part due to accelerators focusing optimization for dense operations, the field of graph representation models has not been given the same attention. Graph data is inherently non-euclidean in representation, and operations on them open an entire new domain of AI tasks that conventional euclidean based ANN's have failed on.

2.2 Preliminary

For the purpose of making the rest of this chapter comprehensible, some introduction into graphs, terminology, and adjacency matrix representation is required:

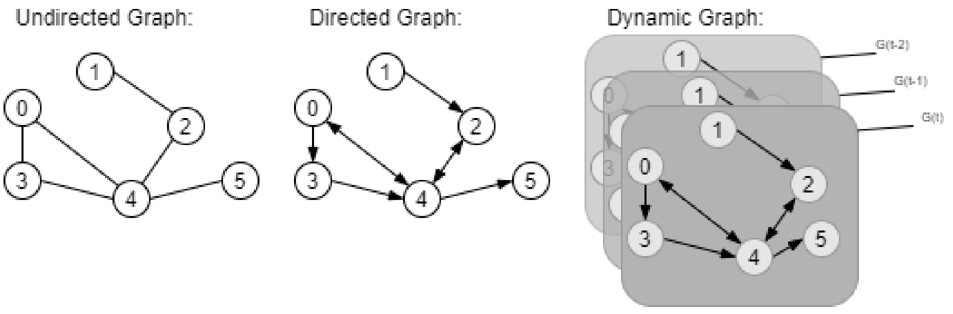

Figure 2.1: Graphs

The Directed graph is denoted as $G = (V, E)$ where V is the vertices and E is the edges. A vertex is represented as $v_i \in V$ and an edge as $e_{ij} = (v_i, v_j) \in E$. A directed graph is illustrated in figure 2.1.

The adjacency matrix $\mathbf{A}$ is derived as an $n \times n$ matrix where

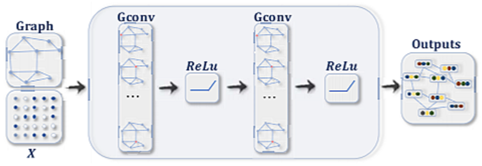

Figure 2.2: Recurrent Graph Neural Network

This figure is taken from "A Comprehensive Survey on Graph Neural Networks" by Zonghan Wu et al.[52].

$$A_{ij} = \begin{cases} 0 \text{ when } e_{ij} \notin E \\ \hline 1 \text{ when } e_{ij} \in E \end{cases} \tag{2.1}$$

A graph might have an attributed graph **X** which holds information of its vertex attributes, $\mathbf{X} \in \mathbf{R}^{n \times d}$ which is a vertex feature matrix with $x_v \in \mathbf{R}^d$ representing the feature vector of a vertex v. The graph may have edge attributes $\mathbf{X}^e$, where $\mathbf{X}^e \in \mathbf{R}^n \times c$ is an edge feature matrix with $x^e_{u,v} \in \mathbf{R}^c$ representing the feature vector on an edge *(v, u)*.

An undirected graph is a special case of the directed graph where $\forall v_{ij} \exists v_{ji}$, i.e., a graph where edges always go both ways. Note that the adjacency matrix of an undirected graph is symmetric. An undirected graph is illustrated in figure 2.1.

The Spatial-Temporal Graph (STG) exists in the context of the Spatio-Temporal Graph Neural Network. It has permanent vertices but edges exist in spans of time. In addition, the STG has vertex inputs that change dynamically over time. It is used to map dynamic systems.

Table 2.1 displays some common notations that are used throughout this document.

Notation	**Description**
$\mathbf{X} \in \mathbf{R}^{n \times d}$	The feature matrix of a graph.
$\mathbf{x}_v \in \mathbf{R}^n$	The feature vector of the node v.
$\mathbf{X}^e \in \mathbf{R}^{m \times c}$	The edge feature matrix of a graph.
$\mathbf{x}^e_{(u,v)} \in \mathbf{R}^c$	The edge feature vector of the edge (v, u).

Table 2.1: Commonly used notations

2.3 Main Variations

2.3.1 The Recurrent Graph Neural Network (RecGNNs)

Early research into GNNs mostly focused on directed acyclic graphs [47], these GNNs were adaptions of the Recurrent Neural Networks (RNNs) that apply a static set of parameters over nodes recurrently to extract high-level node representations. Zonghan Wu et al. labeled these Recurrent Graph Neural Networks [52] in their taxonomy.

Important works:

- Graph Neural Network (GNN*)[1] [47]: To handle general types of graphs (e.g., acyclic, cyclic, directed, undirected), Scarcelli et al. extends prior recurrent models in their GNN*. Using the information diffusion mechanism the GNN* updates node states by exchanging neighbourhood node states in a recurrent fashion until a stable equilibrium is met. The node's hidden state is updated by

  $$\mathbf{h}_v^{(t)} = \sum_{u \in N(v)} f(\mathbf{x}_v, \mathbf{x}^e_{(v,u)}, \mathbf{x}_u, \mathbf{h}_u^{(t-1)})$$

  where $f(\cdot)$ is a recurrent parametric function, and $\mathbf{h}_v^{(0)}$ is initialized randomly. The GNN* is applicable to all nodes, regardless of differing number of neighbours, unknown neighbourhood ordering, etc, due to the sum operation. The function f should be a contraction mapping to ensure convergence, this means points mapped to a latent plane have their distance shrunk[2]. To enable the GNN* to operate on cyclic graphs, the GNN* alternates the stage of node state propagation and the stage of parameter gradient computation.

- Graph Echo State Network (GraphESN) [17]: The GraphESN increases the training efficiency of the GNN* by employing an echo state network. It has an encoder and an output layer. The encoder is not trained and does not require training. A contractive state transition function recurrently updates node states until the graph state reaches convergence. The output function is a simple feed-forward linear unit.

- Gated Graph Neural Network (GGNN) [34]: GGNN use Gated Recurrent Units (GRUs) [10], the GRUs ensuring convergence and eliminates the need to constrain parameters to achieve this. The employment of a GRU also means the recurrence is reduced to a fixed number of steps. Its update function is

  $$\mathbf{h}_v^t = GRU(\mathbf{h}_v^{t-1}, \sum_{u \in N(v)} \mathbf{W}\mathbf{h}_u^{(t-1)}),$$

  where $u,\ v\in\ E$ is the node to be updates and its neighbour(s) respectively, and $\mathbf{h}_v^{(0)} = \mathbf{x}_v$. The GGNN uses back-propagation through time (BPTT) algorithm, which is a problem when working on large graphs, as the recurrent functions need to be run multiple times over all nodes, which forces the GGNN to store the intermediate state of all nodes in memory.

2.3.2 The Convolutional graph Neural Network (ConvGNNs)

The ConvGNN is closely related to the RecGNN, where the RecGNN iterates node states by contractive constraints, the ConvGNN is designed to

[1]This is to mean the GNN as proposed by Scarcelli et al. in [47], to avoid ambiguity labeled GNN*.

[2]In the case where f is a neural network, it is suggested that a penalty term be applied to the Jacobian Matrix.

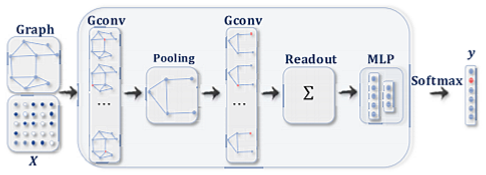

Figure 2.3: Convolutional Graph Neural Network

This figure is taken from "A Comprehensive Survey on Graph Neural Networks" by Zonghan Wu et al. [52].

address cyclic mutual architectural dependencies by using a fixed number of *different* weights in each.

The spatial ConvGNN is heavily based on its non-graph variant (the Convolutional Neural Network) but applied to a nonuniform non-Euclidean space. The first work towards a spatial ConvGNN is the Neural Network for Graphs [39] (NN4G).

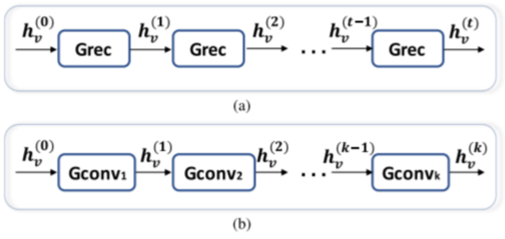

Figure 2.4: A comparison of RecGNNs and ConvGNNs.

This figure is taken from "A Comprehensive Survey on Graph Neural Networks" by Zonghan Wu et al. [52].

Important works:

- Neural Network for Graphs (NN46) [39].
- Contextual Graph Markov Model (CGMM) [3]: proposed a probabilistic model that maintains spatial locality and has the strength of probabilistic interpretability.
- Diffusion Convolution Neural Network (DCNN) [2]: The DCNN uses diffusion-convolution operations to learn diffusion based representations.
- Message Passing Neural Network (MPNN) [19]: MPNN outlined a general framework for spatial-based Convolutional Graph Neural Networks. MPNN is very flexible, able to assume many existing frameworks by assuming different learnable and readout functions. The readout function generates representations based of the full graph using the hidden node representation. MPNN manages this by considering graphs as a message passing process, and iterating passing to propagate information.

- Graph Isomorphism Network (GIN) [54]: Keyulu Xu et al. showed that the MPNN and similar implementations were unable to distinguish graph structures using the embedding product. To remedy this, GIN introduces a learnable parameter to adjust the weight of the central node.

- Graph Attention Network (GAT) [49]: GAT implements attention mechanisms to learn the relative weights between connected nodes (pairwise), this allows GAT to treat neighbourhood contributions to central nodes as non-identical and not predetermined. The GAT uses a LeakyReLU activation function.

2.3.3 The Spectral Convolutional Graph Neural Network (SCGNN)

The SCGNN is derived from a background in signal processing, allowing for the extraction of statistical patterns in large-scale high-dimensional data sets. They extract local features from the graph, but are limited by their learned filters being domain dependent, meaning the filters can't be applied to another graph [52].

Important works:

- Chebyshev Spectral CNN (ChebNet) [14]: The ChebNet would reduce the computational complexity of eigen-decomposition from $O(n^3)$ to $O(m)$ by using multiple simplifications and approximations. The ChebNet in specific uses Chebyshev polynomials. Due to its localized filters, the ChebNet can extract local features independent of the graph's total size. .

- CayleyNet [32]: Building on the ChebNet, the CayleyNet uses Cayley polynomials instead. The Cayley Polynomials are parametric rational functions that allows the CayleyNet to capture narrow frequency bands.

- Graph Convolutional Network (GCN) [25]: The GCN introduces an approximation upon the ChebNet, the GCN is from a spatially based perspective aggregating features from a node's neighbourhood.

- Adaptive Graph Convolutional Network (AGCN)[26]: The AGCN uses a residual graph adjacency matrix though a learnable distance function that takes two node features as inputs to learn hidden structural relations unspecified by just just the conventional adjacency matrix.

- Dual Graph Convolutional Network (DGCN) [58]: As its name suggests, the DGCN introduces dual graph architecture (i.e. using two graph convolutional layers in parallel). While the two layers are otherwise similar, the normalized adjacency matrix $\bar{A}$and the **P**ositive **P**ointwise **Mutual I**n-formation (PPMI) allows the DGCN to encde both local and global structure information without stacking multiple

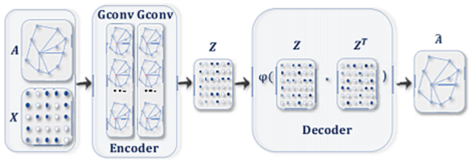

Figure 2.5: Graph Auto-Encode

This figure is taken from "A Comprehensive Survey on Graph Neural Networks" by Zonghan Wu et al.[52].

layers. The PPMI is defined such:

$$\mathbf{PPMI}_{v_1,v_2} = max(\log\frac{count(v_1,v_2)\cdot|D|)}{count(v_1)count(v_2)}),0),$$

where $v_1, v_2 \in V$, $|D| = \sum_{v_1,v_2} count(v_1, v_2)$, and $count(\cdot)$ returns the number of a times a given node (or nodes) occur (or co-occur) in sampled random walks.

Comparing Spatial and Spectral models.

Spatial models have been preferred over spectral models due to efficiency, generality, and flexibility[52]. ConvGNNs can run operations directly on the graph representation via information propagation, the Spectral models in contrast must handle the entire graph or perform eigenvector computations. Spectral models (particularly those heavily dependent on Fourier transformations) assume fixed graphs, and thus are not generalizeable to new graphs. Lastly, the the spectral model assumes nondirected graphs as input, and unlike the spatial model can not be implemented on un-directed graphs.

2.3.4 The Graph Autoencoder (GAE)

Graph autoencoders (GAEs) have two main types: Network embedders and graph generators. The GAE deep neural architecture maps nodes into latent feature spaces, and from latent representations decodes graph information.

Different GAE use different neural networks as encoders, amongst them multi-layer perceptrons, ConvGNNs, RNN, Long short-term memory (LSTM) networks (i.e. RNN), and RecGNNs. The GAE decoder is often a neural network (multi-layer perceptron, LSTM, or deconvolutional network, among others), but can also be a similarity measure, an identity function or a decision process [52].

Network Embedding

A network embedding is a vector representing a node while preserving the node's topological information. To learn network embeddings the

GAE use a decoder to enforce network embeddings to preserve topological information via reconstructing the adjacency matrix or positive pointwise mutual information matrix (PPMI).

Important works:

- Deep Neural Network for Graph Representations (DNGR) [9]: The DNGR encodes and decodes the PPMI. Defined:

$$\mathbf{PPMI}_{v_1,v_2} = max(\log \frac{count(v_1, v_2) \cdot |D|)}{count(v_1)count(v_2)}), 0),$$

  where $v_1, v_2 \in V$, $|D| = \sum_{v_1,v_2} count(v_1, v_2)$, and $count(\cdot)$ returns the number of a times a given node (or nodes) occur (or co-occur) in sampled random walks. It manages this by using a stacked denoising autoencoder. The learned latent representation are robust even with missing values, and preserve highly non-linear regularity behind data.

- Structural Deep Network Embedding (SDNE) [50]: SDNE aims to preserve node first-order and second-order proximity by using a stacked autoencoder. Like the DNGR, the SDNE ignores node features and concerns itself only with node structural information.

- Variational Graph Autoencoder (VGAE)[27]: Using GCN [25] to encode both structural information and node features at the same time, the VGAE aims to reconstruct the adjacency matrix, and from this gain the relational information of the nodes. VGAE is trained by minimizing the negative cross entropy given the original adjacency matrix A and the reconstructed Â, defined as:

$$\hat{\mathbf{A}}_{v,u} = dec(\mathbf{z}_u, \mathbf{z}_v) = \sigma(\mathbf{z}_u^T, \mathbf{z}_v)$$

  where $\mathbf{z}_v$ is the embedding of node v, and σ is the activation function. VGAE is an expansion upon GAE* from the same paper, it expands upon GAE* by optimizing the variational lower bound L by using the Kullback-Lieber divergence function.

- Adverserial Regularized Variational Graph Autoencoders (ARVGA) [43]: The ARVGA employs generative adversarial networks (GAN) to learn the generative distribution of the data.

Graph Generation

The GAE used for graph generation are usually designed with the molecular graph generation problem in mind. These GAEs propose new graphs either sequentially or globally.

GAE use multiple graphs to learn graphs generative distrbution and graph structure by decoding hidden representations.

Early work -Gomez et al. [20], Kusner et al. [29], and Dai et al. [11]- on GAE used SMILES strings with deep Convolutional Neural Networks as encoders and deep Recurrent Neural Networks as decoders.

Later works show that generating graphs iteratively costs structural information, but generating graphs in single steps is unsalable due to the memory footprint ($O(n^2)$).

Important works:

- Deep Generative Model of Graphs (DeepGMG) [35]: While the works of Gomez et al. [20], Kusner et al. [29], and Dai et al. [11] are domain specific, DeepGMG is made applicable to general graphs by iteratively adding nodes and edges to a graph until a given criterion is met. It makes the assumption that the *probability* of a graph is the sum of all possible permutations:

  $$p(G) = \sum_{\pi} p(G, \pi)$$

  where π denotes a node ordering. The graph generation is done by a RecGNN iteratively making decisions (ie. making nodes or edges). This way it captures complex joint probability of all nodes and edges in the graph.

- Graph Recurrent Neural Network (GraphRNN) [55]: In the GraphRNN, it was proposed to use two RNNs: one graph-level RNN that adds new nodes to a node sequence and an edge-level RNN that produces binary sequences[3] indicating connections between the new node and previously existing node.

- Graph Variational Autoencoder (GraphVAE) [45]: GraphVAE outputs entire graphs at a time, being a global approach to the Graph Generation problem. It manages this by using a ConvGNN as an encoder and a MLP as a decoder. To learn, it optimizes a variational lower bound defined as:

  $$L(\phi, \theta; G) = E_{q(z|G)}[-\log p_\theta(G|\mathbf{z})] + KL[q_\phi(\mathbf{z}|G)||p(\mathbf{z})]$$

  , where p(z) follows a Gaussion prior, $q_\phi(\mathbf{z}|G)$ is a posterior distribution defined by the encoder, $q_\theta(G|\mathbf{z})$ is the generative distribution defined by the decoder, and ϕ and θ are learnable parameters.

- Regularized Graph Variational Autoencoder (RGVAE) [38]: Due to issues with the base GraphVAE regarding controlling global properties of the generated graph, such as connectivity, validity, and compatibility amongst nodes, the RGVAE introduces validity restraints on the GraphVAE to regularize the output distribution of the encoder.

- Molecular Generative Adversarial Network (MolGAN) [8]: The MolGAN applies a generative adversarial network (GAN) to construct graphs indistinguishable from empirical data. The generator makes fakes for the discriminator to compare against the data.

[3] An Adjacency Vector

2.3.5 Spatial-Temporal Graph Neural Networks (STGNN)

The STGNN intends to extrapolate both spatial and temporal data from a Spatial-Temporal graph. A spatia-temporal graph changes over time, and often has signals moving over the graph (making walks). The STGNN can be designed to forecast future node values or classifications, or predicting the labels of the graph itself.

STGNNs that employ RNNs use graph convolution to capture spatial-temporal dependencies. Graph Convolutional Recurrent Network (GCRN) uses an LSTM to achieve this. To better examine the process, Zonghan Wu et al. showed given a basic RNN:

$$\mathbf{H}^{(t)} = \sigma(\mathbf{W}\mathbf{X}^{(t)} + \mathbf{U}\mathbf{H}^{(t-1)} + \mathbf{b}),$$

after including the necessary graph convolution functions we have:

$$\mathbf{H}^{(t)} = \sigma(Gconv(\mathbf{X}^{(t)}, \mathbf{A}; \mathbf{W}) + Gconv(\mathbf{H}^{(t-1)}, \mathbf{A}; \mathbf{U}) + \mathbf{b})$$

where the graph convolutional layer is denoted $Gconv(\cdot)$ [52].

Important Works:

- Diffusion Convolutional recurrent Neural Network (DCRNN) [33]: Focusing on the traffic forecasting problem, the DCRNN is designed to treat the problem as a diffusion across the road map. Adopting a decoder and encoder framework, it can project the node values of future steps.

- Structural Recurrent Neural Network (Structural-RNN) [22]: The Structural-RNN suggests using a node-RNN on edge-RNNs, thus handling temporal informatin in more aspects at once. It uses multiple node groups and edge groups to select which RNNs should be used at a given situation.

- Spatio-Temporal Graph Convolutional Network(STGCN) [56]: Like the Structural-RNN the STGCN uses multiple ANNs to extract different features, in the case of the STGCN it is different convolutional blocks. To extract spatial features, it uses graph convolutional neural networks that use Chebyshev polynomials for approximation, and for temporal features it employs gated CNNs.

- Graph Wavenet [53]: Wavenet suggests an adoptive adjacency matrix, that through stochastic gradient descent learns hidden spatial dependencies. They achieve this by randomly initializing two node embedding dictionaries with learnable parameters $\mathbf{E}_1, \mathbf{E}_2 \in R^{N \times c}$, the matrix they used was defined:

  $$\tilde{\mathbf{A}} = SoftMax(ReLU(\mathbf{E}_1\mathbf{E}_2^T)),$$

  The $\mathbf{E}_1$ is the source node embedding and $\mathbf{E}_2$ is the target node embedding, Multiplying $\mathbf{E}_1$ and $\mathbf{E}_2$ reveals spatial dependency weights between source and target nodes, and the ReLU activation function eliminates the weak dependencies.

The STGNN inherits issues regarding computational expenses from the GNNs used in its execution. In the examples above Structural-GNN requires two RecGNNs [22] and STGCN uses multiple ConvGNN [56]. The general challenges of GNNs are addressed in the **Challenges** section, the challenges mentioned therein are even more prevalent for the STGNN.

2.4 Challenges

Practical Challenges

The GNNs -while powerful- are computationally demanding and might require large and unwieldy data sets. The computational requirements of the GNN means the run-time environment of the GNN is often Non Uniform Memory Access (NUMA) architectures or computer clusters.

[24] shows that unoptimized sparse matrix algorithms can be disastrously slow, in the paper they have timed the spMV-E algorithm to more than 9 million seconds (113 days) with the LiveJournal dataset [48] (the LiveJournal dataset constitutes 4,847,000 nodes and 68,993,000 edges, for a density of 2.94e-04).

Data Usage:

While sparse matrices can spare a lot of data real-estate compared to the dense matrix representation, the datasets necessary to represent molecules, city traffic networks, or physics-based systems are still likely to be massive.

Sparse Matrix Multiplication:

Dense matrix multiplication allows computers to use consistent data access patterns, this makes these matrices optimal for Single Instruction Multiple Data (SIMD) optimizations and other non-algorithmic improvements.

The sparse matrix does boast a significantly lower data-size for hypersparse matrices, but at the cost of computational predictability. Workloads are likely to become unbalanced between parallel threads if the sparse matrix itself is unbalanced, the values of addresses can not be located using table coordinates[4], and accessing data column major is even more computationally expensive.

Communications:

In the case of clusters, or architectures -like NUMA- where there are core groupings, there are new concerns: redundant memory usage (primarily replication across cores) and communication latency between processes.

When processes have separate memory there rises a necessity of duplicate memory dependent on requirements.

[4]Where the dense matrix G_d has G_d(i, j) stored at G[i][j], the sparse matrix (using f.ex. the Yale-format) has to locate the G_s(i, j) using a search.

More importantly, to avoid every independent core holding a complete duplicate of the memory, processeses usually broadcast information to each other. Alok Tripathy *et. al.*, proposes multiple algorithms to reduce communication [1]. The outlined 1D, 2D, and 3D algorithms show great promise, communication of dense matrices going down by $2\times$ given $4\times$ more devices, i.e. scaling with $\sqrt{P}$.

2.5 Conclusion

The modern GNN algorithms are powerful tools that work on problems with normal neural networks have long had much issue with F.ex. predictive toxicology [17], protein-protein interactions [49], chemical synthesis [8], and future prediction of networks [33] [56].

The main hurdle of the GNN is that it is more computationally expensive than conventional neural networks and the optimization is less straight forward.

Chapter 3

Background II: Intelligence Processing Unit

3.1 Layout And Latency

The Intelligence Processing Unit is a HPC processor designed for algorithms exhibiting frequent irregular memory accesses. It focuses on fine-grained parallelism.

The IPU works on irregular memory access model by having many independently executing tiles, in contrast to CUDA cores. The IPU-Tile$^{\text{TM}}$(Tile) has one core and a segment of memory.

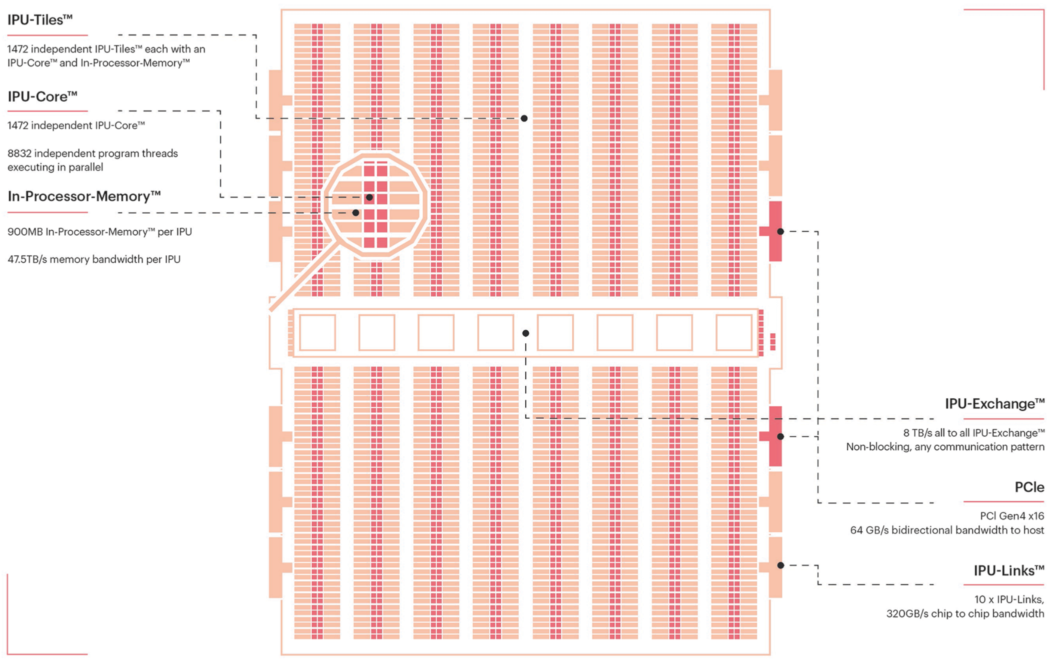
IPU-Tiles™
1472 independent IPU-Tiles™ each with an IPU-Core™ and In-Processor-Memory™
IPU-Core™
1472 independent IPU-Core™
8832 independent program threads executing in parallel
In-Processor-Memory™
900MB In-Processor-Memory™ per IPU
47.5TB/s memory bandwidth per IPU
IPU-Exchange™
8 TB/s all to all IPU-Exchange™
Non-blocking, any communication pattern
PCIe
PCI Gen4 x16
64 GB/s bidirectional bandwidth to host
IPU-Links™
10 x IPU-Links,
320GB/s chip to chip bandwidth

Figure 3.1: IPU architecture
(Mk. 2), courtesy of Graphcore.

The layout of the IPU is illustrated in Figure 3.1 and the IPU's hierarchy is described in Table 3.1. Historically, on the Mk. 1 IPU the tile to tile transfer rate is proximity dependent, being minimal within *islands* and increasing by roughly 1.25 ns for each island further away from the exchange. In general internal collumn latency ranges from 37 ns to 59 ns, cross column latency peaking out at 100 ns [23]. Similar sources on the IPU Mk. 2 are absent at the time of writing, but is estimated to be similar.

Set	Contents	Tot. Tiles
Tile	1 Core + SRAM	1
Island	4 Tiles	4
Column	23 *Islands*	92
IPU	16 Columns	1472

Table 3.1: Hierarchy of the Mk2. IPU

Specifically the IPU boasts 900 MB[1] of memory, divided across said tiles. The Mk1 on the M1000 machine was a two-chip machine, but the newer M2000 machine have four chips and performs significantly better: 7 to 9 times faster than its predecessor when training neural networks and 8 times faster in regards to inference processing [16].

[1]The Colossus Mk. 1 has only $\approx$ 300 MP

The 59.4 billion TSMC produced 7nm transistors delivers roughly 250 Trillion Operations per Second (TOPS) across the 1,472 cores and 900MB SRAM. This is interconnected across a 2.8Tb/s low-latency fabric [16].

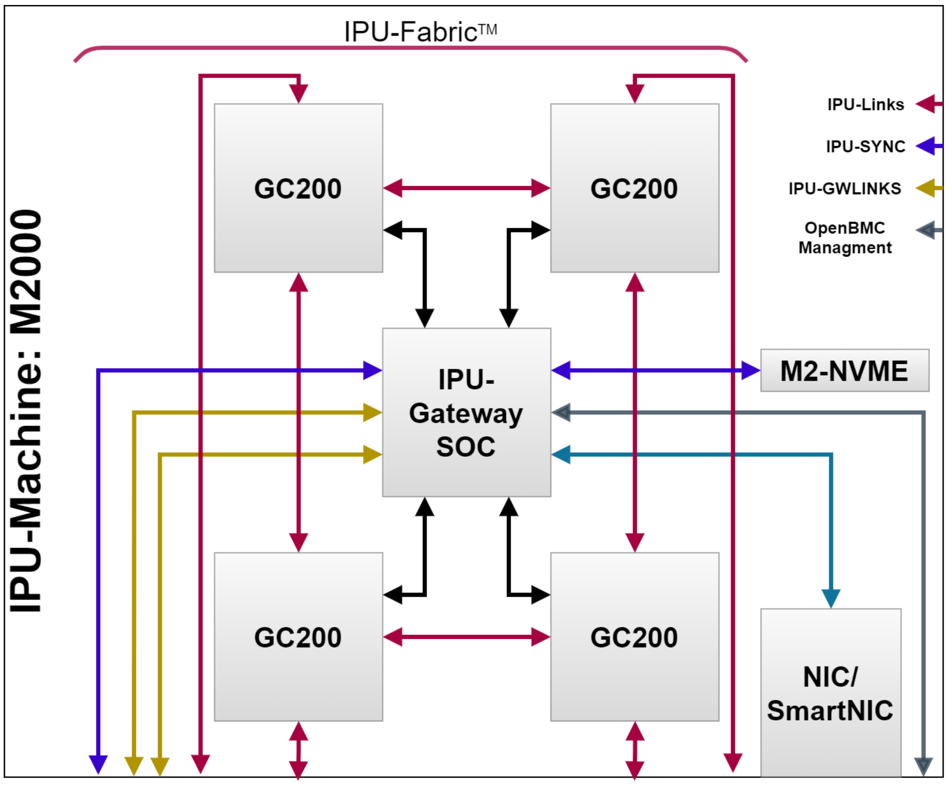

Figure 3.2: The M2000 Board

A graphical interpretation of the IPU board is displayed in Figure 3.2.

3.2 Programming Model

The IPU employs static computational graphs created on the host.

Other sources place the creation of such graphs as being at compile time (see [7], which is accurate to the graph but introduces ambiguity as the exterior program has its own compile-time. Hence, "compile-" and "run-time" will always refer to the computational graph, unless otherwise specified.

The computational graph is defined by alternating layers of states and computational vertices. States refer to the memory stored in tensors. The computational vertices are operations applied to the states, leading into the next state.

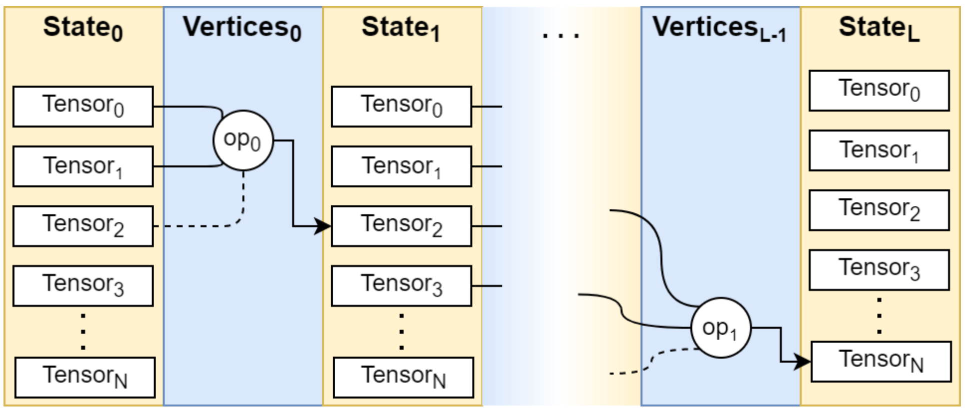

Figure 3.3: Example IPU Computational Graph

In Fig. 3.3 the alternating layers are illustrated. Note that Graphcore defines vertices as mapped to specific tiles.

The vertices are all associated with *codelets*, where a codelet is a piece of code executed *on* a tile (see sSction 3.1 and Table 3.1). Codelets can run in parallel at the same vertex layer as long as they do not write to the same state. All codelets of a vertex layer must compute before entering the next state layer. On the state layer, consistency is enforced across the tensors, creating a strict bulk-synchronous parallel superstep communication structure. See Section 3.3 for more on the BSP.

3.3 Bulk Synchronous Parallel

The IPU employs Bulk Synchronous Parallelism to organize compute and exchange operations. It does this in *supersteps*, each *superstep* having one stage of **local computation**, followed by **communication**, and finally a **barrier**.

During the local computation, there is no communication between processes, each process doing computations relying exclusively on local memory.

In the computation step, there is all-to-all exchange of data, this entailing both sending and receiving data. There is no computation during this step.

During the barrier phase, there is no computation or communication, save that necessitated by the barrier itself.

The BSP is *motivated* by memory contention in simultaneous memory-bound computation and communication being difficult, if not outright impossible [30]. Within a BSP structure memory transfer buffers are redundant, saving overall memory support and increasing communication efficiency. Luk Buchard, *et al.* notes that the BSP model demands that all transfers must be planned at compile time, which is not beneficial for unpredictable memory transfers (such as sparse data transfers required by the BFS algorithm. See Section 4.2) [7].

Chapter 4

Related Work

Other researchers have also worked on the applications of various algorithms to the IPU.

4.1 Traditional HPC tasks on the IPU

Louw and McIntosh-Smith present that the IPU framework can be used to implement structured grid stencil computations and has performance comparable to modern GPU's. Due to the alternative cache-less memory-model of the IPU many of the existing stencil optimizations are inapplicable. In further experiments with 2D convolutions in Gaussian Blur application, the paper claims the IPU has large performance benefits, particularly for the 16-bit precision computation [36].

4.2 Breadth First Search algorithms on the IPU

My personal involvement in this was confined to the first implementation, finished late August 2020, however, Luk Buchard's continued work on performance and published his paper on it in June 2021[7]. Luk *et al.*[7] showed that with a typical 1.5× speedup over the fastest competing GPU and CPU codes. The low memory capacity, $\approx$ 300mb, proved one of the Colossus Mk. 1's main issues, limiting its overall usefulness. This made it more suited for tasks with higher time-complexities, or even NP-hard optimization problems. It is further commented that kernelization techniques become more valuable if they make it possible to shrink problems to fit on the IPU [7].

4.3 The IPU as an Accelerator for Particle Physics simulations

Mohan, *et al.*, further researches the IPUs usefulness in the domain of Particle Physics, and finds promising results. They show that the IPU trains generative-adversarial networks (GANs) faster in all cases, and with small batch sizes ($\lesssim \mathcal{O}(100)$) the IPU outperforms the GPU by a factor typically in the comfortable range of 4 to 5. In the case of their experiment, they tested the first generation IPU (GC2) against the Nvidia TESLA P100 (GPU), Intel Xeon Platinum 8168 (CPU), and Intel Xeon E5-2680 v4 (CPU). Further implementation of a Kálmán filter also gives complimenting results. While their implementations are self-admittedly too different for fair comparisons, the IPU is shown to be much faster. A final note of import in the paper is the acknowledgment that the IPU and its framework are easy to work with, and places the supporting works of the paper in a six-month time-span with no participants having previous IPU programming experience [40].

4.4 The IPU as an Accelerator on Multi-Horizon Forecasting for Limit Order Books

In the domain of machine learning Zhang and Zohren show that the IPU does infact also outperform contemporary GPU's in regards to training networks, and in the case of training an encode decoder model being 4.5+ times faster than the GPU in question. Their experiments were on single-GPU to single-IPU performance, testing with with the NVIDIA GeForce RTX 2080 GPU [57].

Part II

The project

Chapter 5

The Spatio-temporal Graph Convolutional Network

In this section we give insights into the Spatio-temporal Convolutional Network (STGCN) from 2018. This is -in its entirety- the work of Bing Yu, Haoteng Yin, and Zhanxing Zhu, who are responsible for the research in this chapter. This chapter attempts to convey the information in their paper: Spatio-Temporal Graph Convolutional Networks: A Deep Learning Framework for Traffic Forecasting[56].

5.1 Preliminary

5.1.1 Data structuring

In the STGCN, traffic networks are defined on a graph. The observation $\mathcal{V}^t$ is linked into the graph by a a pairwise connection. In this fashion the data point $\mathcal{V}^t$ can be interpreted as a graph signal defined on a graph[1] $\mathcal{G}$. Thus at time step t the graph $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}, W)$, here $\mathcal{V}_t$ is a finite set of vertices that reflect the observations from n monitor stations in a traffic network $\mathcal{E}$ is the edges indicating the connectedness between the observation stations; and $W \in \mathbb{R}^{n \times n}$ describes the weighted adjacency matrix.

[1]Undirected or directed

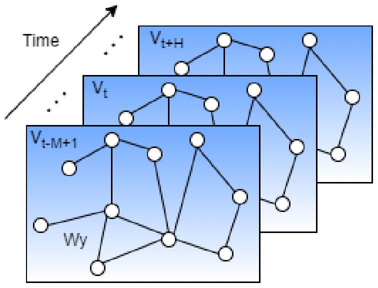

Figure 5.1: Graph-structured traffic data.

V_t denotes the current traffic at timestep t, recorded in a graph structured data matrix.

5.1.2 Performing Graph Convolutions

Regular grid convolutions are not applicable to the general graph. In 2018 (when the STGCN paper was published), contemporary methods of generalizing convolution to structured data forms to expand the spatial definition of a convolution[42] while the other was to manipulate the spectral domain with graph Fourier transforms[6]. The former (Niepert *et al.*, 2016) rearranges the vertices to form certain grid forms more accommodating to conventional graph operations, while the latter (Bruna *et al.*, 2013) employs a spectral framework to apply convolutions in spectral domain.

For the STGCN Bing Yu *et al.* introduced a new convolution operator "$*_{\mathcal{G}}$", the operation views and employs spectral graph convolution as a multiplication of a graph signal $x \in \mathbb{R}$ with a graph kernel Θ

$$\Theta *_{\mathcal{G}} x = \Theta(L)x = \Theta(U\Lambda U^T)x = U\Theta(\Lambda)U^T x, \tag{5.1}$$

where the graph Fourier basis $U \in \mathbb{R}^{n\times n}$ is the matrix of eigenvectors of the normalized Laplacian $L = I_n - D^{-\frac{1}{2}}WD^{-\frac{1}{2}} = U\Lambda U^T \in \mathbb{R}^{n\times n}$ (I_n is here an identity matrix, $D \in \mathbb{R}^{n\times n}$ is the diagonal degree matrix with $D_{ii} = \sum_j W_{ij}$); $\Lambda \in \mathbb{R}^{n\times n}$ is the diagonal matrix of eigenvalues of L, and filter $\Theta(\Lambda)$ is also a diagonal matrix.

5.1.3 Approximations

Due to the computational expense of the graph kernel two approximation efforts are employed.

Chebyslev Polynomials Approximation

By restricting the kernel Θ to a polynomial of Λ as $\Theta(\Lambda) = \sum_{k=0}^{K-1} \theta_k \lambda^k$, where $\theta \in \mathbb{R}^K$ is a vector of polynomial coefficients. K is the kernel size for the graph convolution, this variable determines the maximum radius of convolution for a given central node.

A Chebyslev polynomial $T_k(x)$ can be employed to approximate kernels: $\Theta(\Lambda) \approx \sum_{k=0}^{K-1} \theta_k T_k(\tilde{\Lambda})$ where $\tilde{\lambda} = 2\Lambda/\lambda_{max} - I_n$[2]. With this, the graph convolution can be written as,

$$\Theta *_{\mathcal{G}} x = \Theta(L)x \approx \sum_{k=0}^{K-1} \theta_k T_k(\tilde{L})x, \tag{5.2}$$

where $T_k(\tilde{L}) \in \mathbb{R}^{n\times n}$ is the Chebyshev polynomial of order k evaluated at the scaled Laplacian $\tilde{L} = 2L/\lambda_{max} - I_n$

By computing K-localized convolutions through the polynomial approximations recursively the computational cost of Eq.5.1 to $\mathcal{O}(K|\mathcal{E}|)$.

1st-order Approximation

By stacking multiple localized graph convolutional layers with the first-order approximation of the graph Laplacian, the authors acquire a layer-wise linear formulation. By further implementing a deeper architecture to recover spatial information in depth, the authors avoid being limited to the explicit parameterization given by the polynomials. Due to neural network scaling and normalization, the authors also make the assumption that $\lambda \approx 2$, further simplifying Eq. 5.2 to,

$$\begin{aligned} \theta *_{\mathcal{G}} x &\approx \theta_0 x + \theta_1(\frac{2}{\lambda_{max}L - I_n})x \\ &\approx \theta_0 x - \theta_1(D^{-\frac{1}{2}}WD^{-\frac{1}{2}})x, \end{aligned} \tag{5.3}$$

where θ_0 and θ_1 are shared parameters of the kernel. By replacing θ_0 and θ_1 with a parameter θ parameters are constrained and numerical performance is stabilized, $\theta = \theta_0 = -\theta_1$.; W and D are renomarlized accoring to $\tilde{W} = W + I_n$ and $\tilde{D}_{ii} = \sum_j \tilde{W}_{ij}$. By these means the authors reexpress the graph convolution as,

$$\begin{aligned} \Theta *_{\mathcal{G}} x &= \theta(I_n + D^{-\frac{1}{2}}WD^{-\frac{1}{2}})x \\ &= \theta(\tilde{D}^{-\frac{1}{2}}\tilde{W}\tilde{D}^{-\frac{1}{2}})x, \end{aligned} \tag{5.4}$$

A vertical stack of graph convolutions with a 1^{st}-order approximation achieves a similar effect to K-localised convolutions horizontally, and this exploits the information of a (K-1)-order neighborhood of central-nodes.

The authors also reason their decision in that a layer-wise linear structure scales good with parameters and becomes significantly more efficient for large-scale graphs, due to the order of the approximation being one in all cases.

[2]Here λ_{max} is the largest eigenvalue of L

5.2 Extracting Spatial Features with Graph Convolution

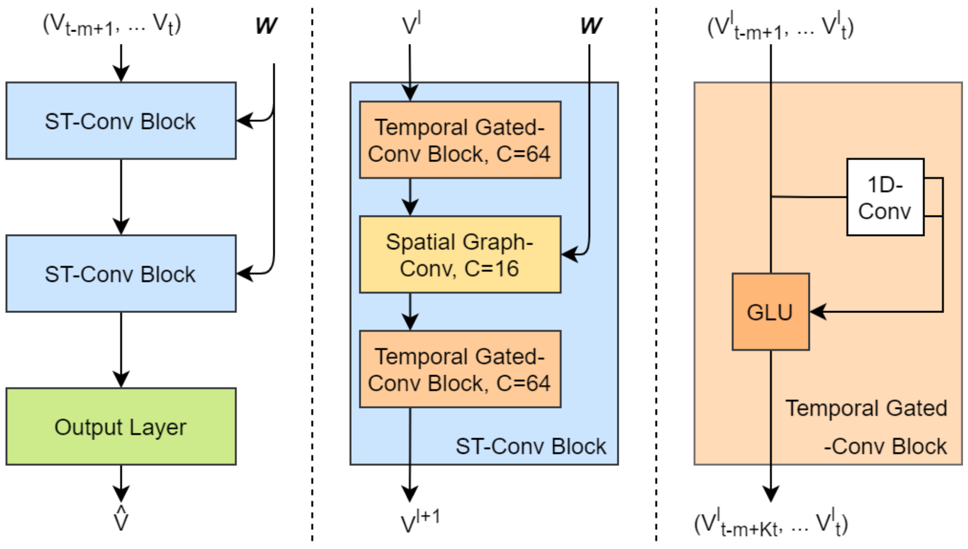

Figure 5.2: The STGCN Model layout

Element	**Algorithm**	**Page**
Temporal Gated-Conv	3	48
Spatio Graph-Conv	5	50
ST-Conv Block	5	50
Output Layer	7	51

Table 5.1: Page Reference Algorithms

The greater architecture of the STGCN model is illustrated in figure 5.2, and table 5.1 refers where said elements can be found in pseudo-code.

5.2.1 Generalizing the Graph Convolution

[56] shows how their graph convolution operator $*_\mathcal{G}$ on $x \in \mathbb{R}$ can be extended to multi-dimensional matrices. They generalize the operation over a signal given C_i channels $X \in \mathbb{R}^{N \times C_i}$ to be:

$$y_i = \sum_{i=1}^{C_i} \theta_{i,j}(L)x_i \in \mathbb{R}^n, 1 \leq j \leq C_o, \tag{5.5}$$

in which the $C_i \times C_o$[3] are the vectors of Chebyshev coefficients $\Theta_{i,j} \in \mathbb{R}^K$. Here we denote the graph convolution of 2-D variables with "$\Theta *_\mathcal{G} X$" with $\Theta \in \mathbb{R}^{K,C_i,C_o}$. Illustrated in 5.1, the input is composed of M frame of road graphs, where the frame v_t is regarded as a matrix where the column

[3] C_i and C_o are the input and output dimension size of the feature map, respectively.

i holds the vector value of v_t at the i^{th} node of graph $\mathcal{G}_t$, said vector having the length C_i.

In the model at all timesteps t of M the equal graph convolution is applied on $X_t \in \mathbb{R}^{n \times C_i}$ with the kernel Θ in parallel. This way Bing Yu, *et al.* further generalized the graph convolution to 3-D variables, expressed as "$\Theta *_\mathcal{G} \mathcal{X}$" with $\mathcal{X} \in \mathbb{R}^{M \times n \times C_i}$

5.2.2 Extracting Temporal Features with a Gated CNN

While there is much presidence for Recurrent Neural Networks (RNN) in this field, Bing Yo, *et al.* opts for a Convolutional approach, citing slow iterations, complex gate mechanisms, and a slow response to dynamic changes[56] in RNNs. Inspired by Gehring *et al.*, 2017[18] they employ instead convolutional structures on the time axis, capturing the temporal dynamic behaviours of traffic flows.

In 5.2 on the right we see the Temporal Gated-Convolution block: containing a one dimensional convolution leading to a gated linear unit (GLU). Note the convolution employs a K_t wide kernel. The aforementioned convolution on sequences of length K samples ranges of size K_t without padding, this shortens the resulting sequence by $K_t - 1$. In this regard the temporal convolution on each node can be regarded as a length-M sequence with C_i channels $Y \in \mathbb{R}^M \times C_i$. The model uses a temporal convolution kernel $\Gamma \in \mathbb{R}^{K_t \times C_i \times 2C_o}$ maps an input Y to a single element $[PQ] \in \mathbb{R}^{(M-K_t+1)\times(2C_o)}$. P and Q is split in two with the same size as channels. The gated convolution is thus defined as,

$$\Gamma *_\mathcal{T} Y = P \odot \sigma(Q) \in \mathbb{R}^{(M-K_t+1)\times C_o}, \tag{5.6}$$

where P and Q are the input of the respective GLU gates; $\odot$ denotes the element-wise Hadamard product. $\sigma(Q)$ is the sigmoid gate, it controls which input P of the current states are relevant for discovering the compositional structure and dynamic variances in time series.

5.2.3 Spatio-temporal Convolution Block (ST-Conv block)

The ST-Conv Block process graph-structured time series to fuse spatial and temporal domain features.

In 5.2 (mid) the ST-Conv Block is illustrated. We see two temporal-gated convolution layers "sandwich" the spatial graph convolution. This "sandwich" design has a spatial layer bridging two temporal layers, a feature that according to the authors is responsible for achieving fast spatial-state propagation from graph convolution through temporal convolutions. Furthermore, this structure aids the network applying bottleneck strategies to achieve scale compression and feature squeezing, this is achieved by down-scaling and up-scaling of channels C though the layers.

To avoid overfitting, a normalization layer is applied pen-ultimately in every ST-Conv block, immediately before a dropout operation.

The ST-Conv Block takes and returns 3D tensors as input/output. For an input $v^l \in \mathbb{R}^{M \times n \times C^l}$ of block l and output $v^{l+1} \in \mathbb{R}^{(M-2(K_t-1)) \times n \times C^{l+1}}$ the computation of the convolution block is given by

$$v^{l+1} = \Gamma_1^l *_{\mathcal{T}} \mathrm{ReLU}(\Theta^l *_{\mathcal{G}} (\Gamma_0^l *_{\mathcal{T}} v * l)), \tag{5.7}$$

where Γ_0^l and Γ_1^l are the upper and lower temporal kernel within block l, respectively; Θ^l is the spectral kernel of the graph convolution; ReLU$(\cdot)$ is the function of the rectified linear unit.

5.2.4 The Output Layer

After two stacked ST-Conv Blocks there is a final Output layer (see 5.2). This final layer maps the outputs of the last ST-Conv block to a single-step prediction. This way the STGCN can obtain a final output $Z \in \mathbb{R}^{n \times c}$, with this it calculates a speed prediction for n nodes by applying a linear transformation across c-channels as $\hat{v} = Zw + b$, where $w \in \mathbb{R}^c$ is the weight vector and b is the bias. The authors employed L2 Loss to measure the performance of their model, giving the loss function of the STGCN for traffic prediction as

$$L(\hat{v}; W_\theta) = \sum_t ||\hat{v}(v_{t-M+1}, ..., v_t, W_\theta) - v_{t+1}||^2, \tag{5.8}$$

where W_θ are all trainable parameters in the model; v_{t+1} is the ground truth.

Chapter 6

Development

6.1 Planning

The overarching goal of the project is to implement the Spatio-Temporal Graph Convolutional Network[56] on Graphcore's Intelligence Processing Unit. The initial plan would follow a series of developmental steps:

- Sequential C++ implementation.
- Parallelized C++ implementation employing OpenMP.
- IPU implementation.

At the time it was assumed the transition from C to IPU would be more convenient than a direct to IPU approach. The terminal decision was to implement the stgcn directly on the IPU as C code, not making intermediate OpenMP or TensorFlow versions.

A point is made that a C++ implementation is indeed desired, even in front of a much simpler python implementation. While the code could be far more easily be implemented in python, doing minimal work to refit the code for the IPU's python-supported eco-system, this implementation would only illustrate computational performance differences, without giving much insight into the device itself.

6.2 Translating the Blueprint

The project involves implementing the Spatio-Temporal Graph Convolution Network[56] to work on the IPU. This meant there was a strong blueprint for the STGCN written in Python.

Translating the Python code to C introduced a series of immediate challenges:

- Python-specific slice operations.
- Implementation of API operations.
- C implementation decisions.

6.2.1 Python-specific slice operations

Unlike C++, Python employs basic slicing for arrays and matrices. This slicing is heavily used in the original STGCN code and could potentially bloat the memory usage in a C++ implementation.

6.2.2 Implementation of API operations

The original code employs heavy use of the TensorFlow (v1) library, the implementation sets up a TensorFlow internal program it calls multiple times, and as such the entire loop iteration of the machine learning algorithm is internally based in its operations. It was early decided to implement the C++ code using TensorFlow for C++.

6.2.3 C implementation decisions

A lot of C implementation decisions became integral to the initial development of the software. It was opted for using Tensor-Flow for C++ to alleviate the need to code every employed matrix operation.

As mentioned in the planning section, it was later decided to code directly on the GraphCore API for C++.

6.3 TensorFlow C++ implementation challenges

Due to the heavy use of TensorFlow in the original implementation, a heavy effort was put into implementing a solution using TensorFlow, which would (theoretically) bridge into a parallel GPU solution, and due to TensorFlow's IPU crossover API's likely a good starting point for the IPU implementation as well.

However, this proved to be far more complicated to attempt than initially believed.

TensorFlow c++ installation: While TensorFlow is easily installed and well documented for Python, installation and documentation for the C++ variants are another matter. The webpage details that it does exist for C++, but there is no installation guide for this library. In addition, there is a Language Binding installation for C, but this is to create TensorFlow support in other languages, not for direct C support.

TensorFlow's limited IPU support: The support for the IPU for TensorFlow does not work like its GPU support. Like many other libraries, TensorFlow has implemented GPU support for computational graph as an optional choice, ie. automatic. To the contrary, the support for the IPU is *included* as separate implementation in the TensorFlow Keras library of Python. This meant that while a C++ TensorFlow implementation might be feasible, it does not lead into an IPU implementation, as it would be based on unrelated functions and -unlike GPU implementations- would not be inherently supported.

A lot of the aforementioned issues have a relatively simple solution by simply using GraphCore's own API directly.

6.4 Using GraphCore's C++ API: Poplar

The IPU requires the construction of a static computational graph, i.e. the CPU side creates a program to be ran on the IPU, that can interface with the CPU throughout. This means that all input into the IPU has to be addressed explicitly, and C/C++ objects and data cannot be directly transferred.

The API does however have support for C++ primitives, and has advanced support for machine-learning operations[1].

The Graphcore API works similarly to other static computational graph systems (F.ex. Tensorflow) in that a graph is compiled dynamically in the code, before being executed as an object.

Its main differences are tied to its tile-based implementation and the care that has to be made for it.

Like all performance optimization, understanding the architecture it is based on is a necessity. The IPU's main features are its distributed memory model and the enforced Bulk Synchronous Parallel execution leaves memory optimization as a crucial consideration for any implementation.

The inbuilt support for this is the tile-mapping system, a property of the tensor object. Transfer of data between tiles is automatized, and end-users do not need to implement the transfer of data between tiles. The default tile-mapping supported (during early development) only specific and linear tile-mapping.

Specific tile-mapping requires the user to set the distinct tile a tensor or subtensor is mapped to.

Linear tile-mapping distributes a tensor object across the tiles as described in the Graphcore API documentation:

> "*The variable will be spread evenly across the tiles with the element ordering matching the tile number ordering.*"
>
> - Graphcore: Poplar and PopLibs API Reference

Certain functions automatically tile-map their tensor output based on planned function calls or input options. Examples of these:

- **poplin::createInput:** createInput is a function designed to create a special tile-mapped tensor as a parameter for the poplin::convolution function, it and its sister function createWeights takes a ConvParams[2], which is used to optimize tile-mapping efficiency without manual entry.

- **poplin::createWeights:** see createInput.

- **poplin::createBiases:** createBiases takes as input a tensor which the biases are to be applied to, and allocates tile-mapping for efficient application later.

[1]Not including weight and bias optimization functions

[2]Convolution Parameters

- **poprand::dropout:** The dropout function takes a tensor of the output shape as reference, the output will not be written to the reference, but it will use the same tile-mapping.

To maximize the performance of the IPU, it is crucial to employ the correct tile-mapping, minimizing the most expensive communication patterns.

6.5 Design Approach

Disambiguation: *This section is on planned desiscions relating the project, for information on the produced code see chapter 7: "The IPU STGCN Implementation".*

It was decided early that adherence to the blueprint would lead to a minimum of ambiguity and make it easier in the later stages of development to test the code. Recreating Python code in other languages often incurs some overhead code for utility purposes, and further additional code would likely be necessary to facilitate TensorFlow like functionality in the interface between the CPU and the replacement of the aforementioned: Poplar libraries.

Due to the likeness between the TensorFlows and Poplar APIs, a lot of the code has natural correlating functions in the other, with some notable exceptions. This correlation means that the resulting code is markedly *similar* to the blueprint (ie. the TensorFlow STGCN code[56]).

The main difference in regards to this implementation is the absence in Poplar of TensorFlow's inherent object-operation relation.

TensorFlow does in Python have the upstream of a variable defined as a part of it, when TensorFlow runs a session it is run in regards to a list of objects, executing so that the aforementioned objects are calculated according to the functions trace of said objects.

Unlike TensorFlow, Poplar's variables are defined exclusively as data containers with tile-mapping, without regards to upstream computation.

This distinction means that the handling of the computational graph in Poplar requires additional measures due to a lower level of abstraction.

6.5.1 On the use of API

The project is mostly aligned with inspecting the IPU as an accelerator and working with its API, as such the intent is to rely, if possible, on the IPU's inbuilt libraries and not custom solutions.

Furthermore, the STGCN model (ref fig. 5.2) itself is a time-sensitive target for optimization. If possible, custom implementation of supporting code is to be avoided as it neither aids in accessing the IPU or its libraries.

6.5.2 On the implementation of back-propagation

The IPU's API does not have any innate support for back-propagation, in the sense that there are no functions to compile a back-propagation operation over a sequence, and there is little to no support for generating gradients for back-propagation either.

While there are specific references to back-propagation in the form of forward and backwards passes, specifically in regards to recurrent units, there is little overall support.

In section 10.2.4 I will talk about inspecting programs and its hypothetical use for the purpose of working *on* programs. However, here I would like to supplement that I believe the tensors themselves should be classifiable as gradients, inputs, and weights, etc.. Overall I believe such changes would allow the poplar libraries to become more flexible, making it possible for developers to create reusable functions for generating programmatic by-products akin to back-propagation.

What back-propagation is to be added has to be made from scratch, most likely in the same functions as the layer they are propagating error over, or using cached variables.

It follows from this that learning will be difficult to implement and has not been made a major target of this project.

What work has been devoted to backpropagation had to be done from scratch in its entirety.

6.6 Implementation Issues

6.6.1 Data Over-saturation

The early poplar STGCN configuration would not be able to compile the graph due to over-saturating the tiles. This is to say that one of the tiles would be prescribed too much memory for its local SRAM to contain.

This issue was circumvented by reducing the batch-size, but not solved.

The single-tile overload highlights a bigger underlying weakness with the IPU architecture: The importance of smart tile-mapping, and the discrepancy between *absolute* memory capacity and *effective* memory capacity.

While the IPU Mk. 2 boasts a total of $\approx$900 MB of memory, the capacity of any given tile is no more the $\approx$ 0.6 MB. Meaning that if the division across the tiles are uneven the effective memory capacity decreases rapidly in tandem with the unevenness of data management.

6.6.2 Verifiable output

Due to the randomizing step of dropouts, performed at the end of each Spatio-Temporal Convolution Block (more about this at section 5.2.3, during runs intended for optimization the results can no be verified as accurate against the original implementation.

However, when verifying the results *before* the dropout, and then inserting the post-dropout numbers from the python code into the IPU code for continuity purposes the accuracy before each dropout was calculated to roughly $\{\approx -5e^{-6}, \approx 5e^{-6}\}$, and the final error is down to the range $\{\approx -5e^{-5}, \approx 5e^{-5}\}$. The error is unlikely to aggregate beyond reasonable accuracy over these two dropouts.

$$\begin{aligned} \Delta_M &= M_{c++} - M_{py} \\ E_- &= MIN(\Delta_M) \\ E_+ &= MAX(\Delta_M) \\ E_r &= (E_-, E_+) \end{aligned} \tag{6.1}$$

Error is calculated as outlined in equation 6.1, where Δ_M is a matrix with the point-wise subtraction of M_{py} (the python computed matrix) from M_{c++} (the native computed result); further the error range E_r is defined as the range between the biggest negative error and biggest positive error.

6.6.3 Standard Computational Bugs

While not hyper relevant to the project, a plethora of minor issues were necessarily removed during the development, most of these standard issue bugs like the wrong use of operation, out of sequence operations, or simply a misreading of either the poplar or TensorFlow documentation.

6.6.4 Flawed Sequence Embedding

One of the pen-ultimate flaws that was worked on during the project was a minor issue where an operation was not computed in order due to being embedded in the erroneous sequence.

This issue is one of the more complicated to root out, but it is simple to rectify.

6.6.5 Error in Cross-Iteration verification

This issue came after the model produced accurate results, however, when the model was re-executed the an error was introduced.

This error was traced back to a de-randomization step injected to verify accuracy across random operations. This extra step relied on an additional matrix to contain the Python post-randomization product.

6.6.6 Back-propagation

During backpropagation the data-stream would become saturated with *inf*, *-inf*, and *nan* states.

While several errors where found the ultimate cause was not detected. This is addressed further in the chapter 7: "The IPU STGCN Implementation".

Chapter 7

The IPU STGCN Implementation

7.1 Layout

7.1.1 Files

```
+-- data
|   +-- csv_reader.hpp
|   +-- data_utility.hpp
+-- main.cpp
+-- model
|   +-- convolution.cpp
|   +-- layer.cpp
|   +-- meta_verify.cpp
|   +-- model_assets.hpp
|   +-- model.cpp
|   +-- transpose.cpp
+-- time
|   +-- time_run.cpp
+-- util
    +-- arguments.cpp
    +-- arguments.hpp
    +-- ipu_interface.cpp
    +-- ipu_interface.hpp
    +-- Logger.cpp
    +-- Logger.hpp
    +-- math.cpp
    +-- Matrices.cpp
    +-- util.hpp
```

Figure 7.1: Files

Figure 7.1 outlines the file/folder hierarchy. Note that some other folders are required for python-synchronized runs.

The model's main assets are all confined to the *model* folder.

7.1.2 Structures

A list of the objects and their functions:

class CSVReader in *data/csv_reader.hpp*

A utility class for reading .csv files. This class object is almost exclusively employed by the *read_csv* function, see section 7.1.3.

- **CSVReader(std::string filename, std::string delm);**
  Constructor, prepares reader.
- **void read();**
  Reads the file into a *vector<vector<string>* > architecture with inherent structure.
- **std::vector<std::vector<std::string> > get_raw();**
  Returns the *vector<vector<string<* > object holding the CSV data, given that it has been initiated with the *read()* function of this object.
- **size_t outer();**
  Returns the length of the outer input *raw*-vector.

  ```
  return CSVReader::raw.size();
  ```

- **size_t inner();**
  Returns the length of the first element in the *raw* vector

  ```
  return CSVReader::raw[0].size();
  ```

class Dataset in *data/data_utility.hpp*

The dataset class is a container for three different Matrix_4D objects, this is used as a container for the train, valid, and test matrices.

- *private* **size_t tot_size(vector<size_t> &shape);**
  Calculate the product of all variables in the shape.
- *private* **void transfer(Matrix_2D &src, Matrix_4D &dst, int len_seq, int n_frame, int n_route, int day_slot, int offset);**
  Copies between data segments.
- *private* **void remember_shape(vector<size_t> a, vector <size_t> b);**
  Retain shape *a* in *b*.

- **Dataset(Matrix_2D &data_seq, int config[3], size_t n_route, size_t n_frame, size_t day_slot = 288, vector<vector<size_t> > shapes = vector<vector<size_t> > 0, 0, 0) : train(shapes[0], "Training Sequence"), valid(shapes[1], "Validation Sequence"), test (shapes[2], "Testing Sequence"), shape_train(4), shape_valid(4), shape_test(4);**
  Sets up the inherent wrapper, duplicates multiple values, and converts others to z_score variants.

class Logger in ***util/Logger.hpp***

This function opens and overwrites a file given as input, used to log activity without clogging up the terminal.

- **Logger(string s, string m_entry = " Logfile ");**
  Creates or overwrites a file with the new name m_entry.
- **Logger();**
  Deconstructor, close the file.
- **void log(string s, string end = "\n", bool announce = false);**
  Append a string *s* to the file.

class Raw_Matrix in ***util/Matrices.cpp***

A simple raw data-space, used as underlying buffer for inheritor classes to impose dimensionality on.

- **Raw_Matrix(size_t n, string titled = "Untitled") : title(titled);**
  Creates an underlying data segment of size *n*, also embeds a title (titled).
- **Raw_Matrix();**
  Deconstructor.
- **float *ptr_to_raw();**
  Returns a pointer to the underlying data.
- **void glorot_fill(size_t fan_in, size_t fan_out);**
  fills the values of this raw data segment with glorot random values.
- **void fill_mat(float val);**
  Fill matrix with the float val.
- **size_t _size();**
  Returns the size of the underlying data segment.
- **float mean();**
  Calculates the mean of the underlying array.

- **float standard_deviation(float mean);**
  Calculate standard deviation given a mean.

- **size_t nnz();**
  Count zeroes in segment.

- **float nnz_percent();**
  Return percentage of segment that is zeroes.

- **void z_score(float mean, float std);**
  Convert data to z_score of data:

  $$z\_score(D[i]) = (D[I] - \mu)/\sigma,$$

  where μ is the mean, *D[i]* is element i of the underlying data array, and σ is the standard deviation. This overwrites existing data segments.

- **void z_inverse[1](float *x, float mean, float std);**
  Rewrites the internal data according to calculation:

  $$D[i] = (x[i] \times \sigma) + \mu$$

class Matrix_2D : public Raw_Matrix in *util/Matrices.cpp*

Applies a 2d matrix overlay to the raw matrix.

- **Matrix_2D(size_t size, vector<size_t> shape, string titled="Untitled") : Raw_Matrix(size, titled);**
  Constructor, creates a 2d matrix overlay to an invoked underlying **Raw_Matrix**.

- **Matrix_2D();**
  Deconstructor, empty.

- **float* operator[](size_t idx);**
  allows 2D accessing of the raw underlying memory.

class Matrix_4D : public Raw_Matrix in *util/Matrices.cpp*

Applies a 4d matrix overlay to the raw matrix. The shape of this 4D matrix is a 4D cuboid and is referenced as such

- **Matrix_4D(vector<size_t> shape, string titled="Untitled") : Raw_Matrix(shape[0]*shape[1**
  Constructor, associate underlying values for ease of access later.

- **float* operator[](size_t cube);**
  returns the address of an underlying cuboid in the 4d cuboid, given by:

[1]Misnomer: This calculates *reverse* z_score

```
        return &data[ZYX*cube];
```

Where ZYX is the product $Z \times Y \times X$ and Z, Y, and X are the inner three dimensions of the 4d cuboid.

- **size_t cube_nnz(ssize_t w, size_t z);**

class Arguments in *util/arguments.hpp*

Simply a wrapper object for the terminal arguments, defaults values on its own.

- **void set(string s, string v);** Attempts to set a dictionary item *s* to the value *s*.

- **Arguments(int argc, char const *argv[]);**
  Basic constructor, calls set on all values *argv*.

class FeedinVector in *util/ipu_interface.hpp*

Feedin vector creates a buffer that can either be filled with an external matrix or float (distributed to the entire vector). This can be used to feed in a tensor to the IPU with a single value (most commonly zero).

FeedinVector has the following three functions, omitting the deconstructor:

- **FeedinVector(size_t s);**
  Constructor, mallocs an area of *s* floats.

- **void copy_in(float * data);**
  Attempts to copy data to the FeedinVector's space.

- **void valfill(float x);**
  Fill every item in the array with the float x.

class FileFedVector in *util/ipu_interface.hpp*

The FileFedVector works much like the FeedinVector, but instead reads a file into its buffer automatically. This is used extensively to fill in tensors.

The FileFedVector has only one function, its constructor:

- **FileFedVector(string filename, size_t _size)**,

this function automatically creates a memory array of size $\_size \times sizeof(float)$ and fills it with the contents of the file *rawdata/<filename>.txt*.

class IPU_Interface in *util/ipu_interface.hpp*

A wrapper object for multiple poplar objects, and handles multiple graph interactions.

- **Tensor_Entry[2] new_entry(string s, Tensor t, int type=0, float *ptr=dummy) size_t unique_exp_adr();**
  tensor entry is used in a list for reacquisition of arrays based on names, and are automatically tilemapped before compilation.

- **size_t unique_exp_adr();**
  Creates a unique expression to append to a string to ensure that it will not match a string search.

- **void update_allocated_bytes(vector<size_t> shape);**
  A function to count allocated bytes on the IPU.

- **size_t tensor_size(vector<size_t> shape);**
  Returns the product of the items in shape, ie: $\leftarrow shape[0] \times shape[1] \times ... \times shape[size(shape)]$.

- **string shape_display(vector<size_t> shape, string name = "x", string end="\n");**
  Creates a string that displays the values in *shape*, starting with *name* and terminated with *end*.

- **string shape_display(Tensor &t, string name = "x", string end="\n");**
  Similar to the above, working on *t.shape()* instead of a *shape*.

- **Tensor expandTensor(Graph &g, Tensor &t, vector<size_t> shape, string name="");**
  Expand the tensor t such that is has the new shape *shape*, it will not shrink dimensions.

- **size_t exists(std::string s);**
  Verifies whether or not a tensor entry with the name *s* in the list of archived tensors.

- **Tensor getVariable(Graph &g, std::string name, std::vector<size_t> shape=NULL_VECTOR, int type=3, float *ptr=dummy);**
  This will create and automatically store a tensor of the shape in shape, with the name *name* and an attached finalization protocol.

- **Tensor getVariable_OLD(Graph &g, std::string name, std::vector<size_t> shape=NULL_VECTOR, int type=3, float *ptr=dummy);**
  This function attempts to retrieve an existing tensor labeled *name*, if not existing it will attempt to create one.

[2]Tensor entry is defined as: std::tuple<std::string, Tensor, int, float*>

- **Tensor padTensor(Graph &g, Tensor &core, vector<size_t> avant_padding, vector<size_t> post_padding);**
  This is similar to expandTensor, however it does not look at the input size of the *core*, rather expanding it as the expand tensors avant_padding and post_padding directs. As such, the resulting shape can be calculated by:

$$\begin{aligned} shape_{new} = \{&pad_{pre}[0] + shape_{old}[0] + pad_{post}[0], \\ &pad_{pre}[1] + shape_{old}[1] + pad_{post}[1], \\ &... \\ &pad_{pre}[n-1] + shape_{old}[n-1] + pad_{post}[n-1]\}, \end{aligned} \tag{7.1}$$

  where n is the shape length of the shape of the core, and pad_{pre} and pad_{post} are shorthand for avant_padding and post_padding respectively.

- **Tensor getExistingVariableSlice(std::string name, std::vector<size_t> shape=NULL_VECTOR, std::vector<size_t> offset=vector<size_t>{0, 0, 0, 0}, int type=0, float *ptr=dummy);**
  This function attempts to find an archieved tensor with the name *name*, and attempts to extract a *shape*-shaped slice starting at the offset. With default parameters, it will only handle 4D tensors. The requirement of type and pointer is vestigial code from earlier functionality.

- **void addVariable(std::string s, Tensor &t, int type=AUTOFL_TENSOR, float *ptr=dummy);**
  This archives the tensor *t* under the label *s*, it also stores compile-time filling protocol and if entered its feed-in address, this is the address used to feed certain tensors with CPU side data..

- **Tensor getAlternatingSpace(string type, std::vector<size_t> shape);**
  This function is a SRAM-recycling function, designed to allow for reusing the alternating layer data, it returns a space designated by the *type*, alternating between two different nonoverlapping segments of said data. While not infinitely thread safe, this function is very useful in acquiring large segments of data with a relatively high chance of not damaging the downstream pipeline. It is to the best of my knowledge, not employed erroneously in the current code, but it should not be employed unless the downstream is safe, and never as an input node.

- **float random(float top, size_t div = 10000);**
  This functions aquires a *raw* random value from C++'s inbuilt *rand()* function, it then performs a set of functions on it to get a random

number in a given range:

$$
\begin{aligned}
r &= rand() \\
r_c &= r \;\%\; (top + 0.5) \\
r_d &= r / \begin{cases} r_c & \text{if } r_c \neq 0 \\ 1 & otherwise \end{cases} \\
r_t &= r_d \;\%\; div \\
v &= r_c + \frac{r_d}{div}
\end{aligned}
\tag{7.2}
$$

- **void glorot_fill(float *ptr, size_t len, std::vector<std::size_t> shape);**
  This function is designed to fill a range from *ptr* of length *len* with glorot random numbers. It should do this by calling the function *random* with the value $sd \times 2 + 1$ and subtacting *sd*, where *sd* is defined as:

  $$sd = \sqrt{\frac{6}{n_{in} - n_{out}}} \tag{7.3}$$

  where n_{in} and n_{out} are in regard to nearby convolutions. With this, the final random values should be in the range $(-sd, sd]$.

- **Engine finalize_and_run(Graph &g, Program model, bool run=false);**
  This function has its own subsection: section 7.1.3.

- **Program notification(Graph &g, string notification);**
  This function returns a Program that gives a notification with the message given (var: *string notification*), used for debugging purposes.

- **void retain(FileFedVector &ffv);**
  This function will hold onto the FileFedVector, a countermeasure against garbage collection.

7.1.3 Functions

This section contains functions from the repositories. Functions with a parameter Sequence &bwd also attempts to construct a backward pass of their function.

int read_csv(string path, Matrix_2D &data)

This function initiates a CSV_Reader (ref section 7.1.2) object, activates it to read the CSV, then it parses it into *data*.

Engine finalize_and_run(Graph &g, Program model, bool run=false)

Found in class IPU_Interface, this function is given its own section to better see all its functions.

The function has multiple distinct phases:

- **Create Graph Tensor Writes:**
  Iterates over the archived tensors, facilitates a write to the graph. In the simpler cases of glorot randoms, FeedIn tensors, or FileFed tensors, it simply employs the poplar *Graph.createHostWrite(...)* function. In the vaguely more complicated case of zeroed out arrays it adds itself to a queue of zeroed arrays. These are initiated overlapping due to the identical region, saving space in the process.

- **Compile Graph:**
  A call to *poplar::compileGraph(...)*.

- **Mounting Graph to Engine:**
  Initiation of the *Engine(...)* class.

- **Prepare and Deploy Engine:**
  Prepares and deploys the engine, using preexisting poplar calls.

- **Data Transfers:**
  Iterates over the archived tensors, this time for transfer to the unit.

  - *if* Zero tensor *then* load a sized segment of a zeroed data segment to the IPU.
  - *if* Glorot tensor *then* Create a glorot random area, fill it with the required random numbers, and transfer them to the IPU.
  - *if* FeedIn tensor *then* Load a region starting at the pointer with the required size to the IPU.
  - *if* FileFed tensor *then* Load a region starting at the pointer with the required size to the IPU.
  - *if* Autofill Tensor *then* Do not fill this tensor, for various reasons this tensor does not require preinitiated variables.

- **Run [Optional]:**
  If the input parameter *run* is set to *true*, also test-run the application. This had a vestigial function during early testing.

void layer_entry_note (IPU_Interface &ipu, string layer, string scope, string notes="");

This function adds prints a debug notification when entering a given model layer function.

void layer_exit_note (IPU_Interface &ipu, string layer, string scope, string notes="");

This function adds prints a debug notification when exiting a given model layer function.

Tensor transp_0123_0231 (IPU_Interface &ipu, Graph &g, Tensor &in, Tensor &out, Sequence &seq);

Vestigial function wrapper, the original functionality involved constructing a computation set that would perform a transposition of the nature $(0, 2, 3, 1)$, currently it calls ::dimShuffle() on the tensor with configuration "{0, 2, 3, 1}" and copies into the out tensor. While it is not necessary to copy, this mimics the previous functionality.

Tensor transp_0123_0312 (IPU_Interface &ipu, Graph &g, Tensor &in, Tensor &out, Sequence &seq);

Vestigial function wrapper, the original functionality involved constructing a computation set that would perform a transposition of the nature $(0, 3, 1, 2)$, currently it calls ::dimShuffle() on the tensor with configuration "{0, 3, 1, 2}" and copies into the out tensor. While it is not necessary to copy, this mimics the previous functionality.

Tensor gconv (IPU_Interface &ipu, Graph &g, Tensor &src, Tensor &dst, Tensor &theta, size_t Ks, size_t c_in, size_t c_out, Sequence &seq, Sequence &bwd, string scope="unscoped");

This function correlates to a function of the same name in the original STGCN implementation [56].
This function creates a program sequence that performs a graph convolution as introduced in [56], outlined in algorithm 1. The aforementioned algorithm does not describe the functions full functionality, but it does describe the layer embodied in the model.

Algorithm 1: Graph Convolution

```
input  : $X, \phi, \theta, Ks, c\_in, c\_out$
output: $X_{conv}$ : Convolved Graph Data
/* $X$ : Input                                              */
/* $\phi$ : Graph Kernel                                    */
/* $\theta$ : Trainable Graph Kernel                        */
/* $Ks$ : Kernel size of Graph Convolution                  */
/* $c_{in}, c_{out}$ : size of input/output channel respectively */
1  begin
2      $n \leftarrow$ outer shape of $phi$
3      $X_t \leftarrow X^{T_{(0,2,1)}}$
4      $X_r \leftarrow \text{reshape}(X_t, [-1, n])$
5      $X_m \leftarrow X_r \times \phi$
6      $X_{mr} \leftarrow \text{reshape}(X_m, [-1, c_{in}, Ks, n])$
7      $X_{mt} \leftarrow X_{mr}^{T_{(0,3,1,2)}}$
8      $X_{mtr} \leftarrow \text{reshape}(X_{mt}, [-1, c_{in} \times Ks])$
9      $X_{mm} \leftarrow X_{mtr} \times \theta$
10     $X_{gconv} \leftarrow \text{reshape}(x_{mm}, [-1, n, c_{out}])$
```

Tensor layer_norm (IPU_Interface &ipu, Graph &g, Tensor &src, Tensor &dst, Sequence & Seq, Sequence &bwd, string scope="unscoped");

This function correlates to a function of the same name in the original STGCN implementation [56].
The layer_norm function performs as it name insinuates a layer normalization, this is described further in algorithm 2. The aforementioned algorithm does not describe the functions full functionality, but it does describe the layer embodied in the model.

Algorithm 2: Layer Normalization

```
   input : $X, \gamma, \beta$
   output: $X_{norm}$
   /* $X$ : Input                                  */
   /* $\gamma$ : Trainable Kernel                   */
   /* $\beta$ : Trainable Bias                      */
 1 begin
 2     $\mu \leftarrow mean(X)$
 3     $\sigma \leftarrow std\_Deviation(X, \mu)$
 4     $X_{norm} = (X - \mu)/\sqrt{\sigma + 1e^{-6}} \times \gamma + \beta$
```

Tensor temporal_conv_layer(IPU_Interface &ipu, Graph &g, Tensor &src, Tensor &dst, size_t Kt, size_t c_in, size_t c_out, Sequence &seq, Sequence &bwd, string scope="unscoped", string act_func="relu");

This function correlates to a function of the same name in the original STGCN implementation [56].
The temporal convolution is described in length in algorithm 3. The aforementioned algorithm does not describe the functions full functionality, but it does describe the layer embodied in the model.

Tensor spatio_conv_layer (IPU_Interface &ipu, Graph &g, Tensor &src, Tensor &dst, size_t Ks, size_t c_in, size_t c_out, Sequence &seq, Sequence &bwd, string scope="unscoped");

This function correlates to a function of the same name in the original STGCN implementation [56].
The spatio convolution layer employs the Graph Convolution, the former described in algorithm 4 and the latter in algorithn 3. The aforementioned algorithms do not describe the functions full functionality, but it does describe the layer embodied in the model.

Algorithm 3: Temporal Graph Convolution

```
    input  : $X, Kt, c_{in}, c_{out}, \alpha$
    output: $X_{conv}$: Temporal Convolution Results.
    /* $X$ : Input                                                    */
    /* $Ks$ : Kernel size of Temporal Convolution                     */
    /* $c_{in}, c_{out}$ : size of input/output channel respectively  */
    /* $\alpha$ : activation function                                 */
 1  begin
 2      $o, T, n \leftarrow$ dimensions 0, 1, and 2 of X
 3      if $c_{in} > c_{out}$ then
 4          $wt_{in} \leftarrow$ trainable variable shaped$(1, 1, c_{in}, c_{out})$
 5          $x_{in} \leftarrow$ 2D-conv$(X, wt_{in})$ // PAD="SAME", stride={1,1,1,1}
 6      else if $c_{in} < c_{out}$ then
 7          $Z \leftarrow zeros([o, T, n, c_{out} - c_{in}])$
 8          $X_{in} \leftarrow concat(X, Z)$
 9      else
10          $X_{in} \leftarrow X$
11      $X_{in} \leftarrow X_{in}[:, Kt - 1 : T, :, :]$
12      if $\alpha = "GLU"$ then
13          $K_{iw} = K_{iw} \times 2$// $K_{iw}$:  The inner Kernel Width
14      else
15          $K_{iw} \leftarrow c_{out}$// $K_{iw}$:  The inner Kernel Width
16      $wt \leftarrow$ Trainable variable shaped$[Kt, 1, c_{in}, K_{wi}]$
17      $bt \leftarrow$ Trainable variable shaped$[K_{wi}]$
18      $X_{conv} \leftarrow$ 2D-conv$(X, wt) + bt$
        /* PAD="VALID", stride={1,1,1,1}                              */
19      if $\alpha = "GLU"$ then
20          $X_{sig} \leftarrow sigmoid(X_{conv}[:, :, :, -c_{out} :])$
21          $X_{conv} \leftarrow (X_{conv}[:, :, :, 0 : c_{out}] + X_{in}) \times X_{sig}$
22      else if $\alpha = "linear"$ then
            /* Do nothing, auto returns $X_{conv}$                    */
23      else if $\alpha$ = "sigmoid" then
24          $sigmoid(X_{conv})$
25      else if $alpha$ = "relu" then
26          $relu(X_{conv})$
27      else
28          Terminate, an unsuported activation function has occured.
```

Algorithm 4: Spatial Convolution Layer

```
    input : $X, Ks, c_{in}, c_{out}$
    output: $X_{sc}$
    /* $X$ : Input                                              */
    /* $Ks$ : Spatial Convolution with                          */
    /* $c_{in}, c_{out}$ : Channel width in and out of layer    */
 1  begin
 2      $T, n \leftarrow$ dimensions 1 and 2 of X, counting from zero.
 3      if $c_{in} > c_{out}$ then
 4          $ws_{in} \leftarrow$ trainable variable shaped$[1, 1, c_{in}, c_{out}]$
 5          $x_{in} \leftarrow$ 2D-conv$(X, wt_{in})$ // PAD="SAME", stride={1,1,1,1}
 6      else if $c_{in} < c_{out}$ then
 7          $Z \leftarrow zeros([o, T, n, c_{out} - c_{in}])$
 8          $X_{in} \leftarrow concat(X, Z)$
 9      else
10          $X_{in} \leftarrow X$
11      $ws \leftarrow$ trainable variable shaped$[Ks \times c_{in}, c_{out}]$
12      $bs \leftarrow$ trainable variable shaped$[c_{out}]$
13      $X_r \leftarrow$ reshape$(X, (-1, T, n, c_{out}))$
14      $X_{conv} \leftarrow$ GCONV$(X_r, ws, Ks, c_{in}, c_{out}) + bs$
15      $X_{cr} \leftarrow$ reshape$(X_{conv}, [-1, T, n, c_{out}])$
16      $X_{sc} \leftarrow$ RELU$(X_{cr}[:, :, :, 0 : c_{out}] + X_{in})$
```

Tensor st_conv_block (IPU_Interface &ipu, Graph &g, Tensor &src, size_t Ks, size_t Kt, size_t channels[3], Sequence &seq, Sequence &bwd, string scope="unscoped", string act_func="GLU");

This function correlates to a function of the same name in the original STGCN implementation [56].
The st-conv-block, or the spatio-temporal graph convolution block, is visualized in figure 5.2 and described in algorithm 5. The aforementioned algorithm does not describe the functions full functionality, but it does describe the layer embodied in the model.

Tensor fully_con_layer (IPU_Interface &ipu, Graph &g, Tensor &src, Tensor &out, size_t n, size_t channel, Sequence &seq, Sequence &bwd, string scope="unscoped");

This function correlates to a function of the same name in the original STGCN implementation [56].
The fully connected layer is the final layer to the output layer, it is described in algorithm 6. The aforementioned algorithm does not describe the functions full functionality, but it does describe the layer embodied in the model.

Algorithm 5: ST-Conv Block

```
  input  : $X, Ks, Kt, C, kb, \alpha$
  output: $X_{out}$
  /* $X$ : Input                                              */
  /* $Ks, Kt$ : Spatial and Temporal Convolution size          */
  /* $C$ : Channel vector, 3 numbers                           */
  /* $kb$ : Keep Probability for dropout                       */
  /* $\alpha$ : Activation function                            */
1 begin
2     $c_{in}, c_t, c_{oo} \leftarrow C$
3     $X_s \leftarrow$ Temporal Graph Convolution$(X, Kt, c_{si}, c_t, \alpha)$
4     $X_t \leftarrow$ Spatial Convolution$(X_s, Ks, c_t, c_t, )$
5     $X_o \leftarrow$ Temporal Graph Convolution$(X_t, Kt, c_{oo}, c_{oo}, "relu")$
6     $X_{out} \leftarrow$ dropout$(X_o, kb)$
```

Algorithm 6: Fully Connected Layer

```
  input  : $X, n, c$
  output: $X_{fc}$
  /* $X$ : Input                                              */
  /* $n$ : width                                              */
  /* $c$ : channel                                            */
1 begin
2     $w \leftarrow$ trainable variable shaped$[1, 1, c, 1]$
3     $b \leftarrow$ trainable variable shaped$[n, 1]$
4     $X_{fc} \leftarrow$ 2D-conv$(X, w) + b$ // PAD="SAME", stride={1,1,1,1}
```

Tensor output_layer (IPU_Interface &ipu, Graph &g, Tensor &src, Tensor &x2, size_t T, Sequence &seq, Sequence &bwd, string scope, string act_func="GLU");

This function correlates to a function of the same name in the original STGCN implementation [56].

Algorithm 7: Output Layer

```
   input : $X, T, \alpha$
   output: $X_{out}$
   /* $X$ : Input                                          */
   /* $T$ : Kernel size of temporal convolution.           */
1  begin
2      $n, c \leftarrow$ dimensions 1 and 2 of X, counting from zero.
3      $X_i \leftarrow$ Temporal Graph Convolution$(X, T, c, c, \alpha)$
4      $X_{ln} \leftarrow$ Layer Normalization$(X_i)$
5      $X_o \leftarrow$ Temporal Graph Convolution$(X_l n, 1, c, c, "sigmoid")$
6      $X_{out} \leftarrow$ Fully Connected Layer$(X_o, n, c)$
```

tuple<Program, Program, Tensor> build_model(size_t blocks[2][3], Arguments args, IPU_Interface &ipu, Graph &g)

This function constructs the model, ie. fig. 5.2. It returns a tuple containing the forward pass, backward pass, and output respectively.

string _shapestring(vector<size_t> shape);

Simply returns a shape vector's contents as a string representing said shape.

vector<size_t> out_shape(vector<size_t> input_shape, vector<size_t> kernel_shape, vector<size_t> padding = vector<size_t>{1, 1, 1, 1}, vector<size_t> stride = vector<size_t>{1, 1, 1, 1});

This function calculates the out_shape given a convolution with the given parameters.

Tensor conv2D_w_bwd(IPU_Interface &ipu, Graph &g, Tensor &input, vector<size_t> filter_shape, string filter_scope, string bias_scope, Tensor &output, Sequence &seq, Sequence &bwd, bool biased, bool same)

The conv2D_w_bwd is a wrapper of the Poplar convolution environment that creates a forward and backward convolution operation.

It will create a program for the forward pass and save it in the Sequence &Seq and create a backward pass implementation that it embeds in Sequence &bwd.

The forward pass thus entails:

$$X \otimes \theta + \beta = \sigma \tag{7.4}$$

or (if same is set to *true*):

$$X \otimes_{=} \theta + \beta = \sigma \tag{7.5}$$

where *in* is the input, θ is the filter, β is the bias, $\otimes$ is the convolution operation, and $\otimes_{=}$ is a variant of the convolution that zero-pads the input such that the result has the same shape as the input.

Dependent on repo it will either attempt to initiate the weight and possible bias as glorot random values *or* read values from files dependent on the filter_scope and bias_scope variables.

The bools *biased* and *same* toggle bias and padding respectively.

The implementation of back-propagation employs the already existing poplar update functions for weights and biases, however as no functional way to find the backward error using single-poplar functions, this is implemented from scratch:

Given the 2D function for finding the error w.r.t. the input

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial \sigma} \otimes_{\boxdot} 180^\circ rot\{\theta\} \tag{7.6}$$

where $\theta \in \mathbb{R}^{h \times w}$ is the filter of the forward pass, $\frac{\partial L}{\partial \sigma} \in \mathbb{R}^{H' \times W'}$ is the loss propagated through the backward pass, and $\otimes_{\boxdot}$ is a *full* convolution, that is to say that zero-padding is applied in such a manner that the output has dimensions $\mathbb{R}^{(H'+h-1)\times(W'+w-1)}$, or more comprehensibly: given that the first parameter of the full convolution is the output of the forward convolution this operation will return a matrix with the same dimensions as the input of the forward convolution: $\mathbb{R}^{H \times W}$.

The forward pass of a convolution involving channels is more complex, as the input becomes the three-dimensional: $\mathbb{R}^{C_{in} \times W \times H}$, the filter becomes four-dimensional: $\mathbb{R}^{C_{in} \times C_{out} \times w \times h}$, such that the output has shape $\mathbb{R}^{C_{out} \times H' \times W'}$. To work this backwards, the outer two dimensions of the filter have to be transposed.

Note: The rotation of θ is synonymous with reversing both dimensions of the filter:

$$\begin{aligned} 180^\circ rot \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} = & \qquad\qquad \begin{pmatrix} a_{2,2} & a_{2,1} \\ a_{1,2} & a_{1,1} \end{pmatrix} \\ \leftrightarrow \left(\updownarrow \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \right) = \leftrightarrow & \begin{pmatrix} a_{2,1} & a_{2,2} \\ a_{1,1} & a_{1,2} \end{pmatrix} = \begin{pmatrix} a_{2,2} & a_{2,1} \\ a_{1,2} & a_{1,1} \end{pmatrix} \end{aligned} \tag{7.7}$$

Using this we can utilize the `poplar::Tensor::reverse(`*`unsigned int`*` dimension)` function to rotate the matrix.

Tensor verify_same_2(Graph &g, Tensor &res, Tensor &inp, Sequence &seq, string label = "UNTITLED_VER", bool correct = false);

This function prints the difference (and on deep verification dimension-wise difference) of two tensors. This is the function that calculates equation 6.1.

Tensor verify_same(IPU_Interface &ipu, Graph &g, Tensor &res, string file, Sequence &seq, bool correct = false);

This function calls *verify_same_2* after reading the file with name *"rawdata/<file>.txt"* to a tensor.

Tensor verification_pass(IPU_Interface &ipu, Graph &g, Tensor &inp, string file, Sequence &seq);

The verification_pass function will only execute code if the compile-time variable -D_VERIFY is set. If it is active, it will call verify_same.

Tensor derandomize(IPU_Interface &ipu, Graph &g, Tensor &rnd, string file);

Simply returns the contents of the file *"rawdata/<file>.txt"* fitted into a tensor of the same shape as *rnd*.

7.2 Forward Pass

The functions listed in section 7.1.3 are were made with the forward pass in mind, and were later retrofitted to also construct the backward pass.

The forward pass is designed to follow the original STGCN model [56] as closely as possible, the model is described at length in chapter 5.

To make the two models ([56] and this project) relatively comparable the functions responsible for creating the different layers have been made such that the C++ variants have a natural Python equivalent.

7.3 Backward Pass

7.3.1 Preliminary

In the python implementation [56] the backwards pass is automatically generated by either an RMSPropOptimizer or an AdamOPtimizer from calls to `tf.train.RMSPropOptimizer(...).minimize(...)` or `tf.train.AdamOptimizer(...).minimize(...)` respectively.

These functions are relatively speaking black boxes, making it difficult to inspect the computational graph responsible for the learning process.

Thus the backward pass as it is implemented in this solution is made from scratch and does not equate the forward pass of the Python variant.

7.3.2 Backpropagation

The basic theory of the backwards pass is to propagate an error L backwards through the algorithm.

For linear transformations like transposition and reshaping of matrices the backpropagation simply involves reversing the operation.

For more complicated operations like matrix multiplication, and convolutions backpropagation involves calculating the partial in respect to the inputs.

The generic function

Given a generic function

$$f(x, \theta) = \sigma$$

and the calculated error of the output $\frac{\partial L}{\partial \sigma}$ the backward pass calculates $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial \theta}$ and applies the latter to θ.

Matrix Multiplication

The matrix multiplication is the natural representation of the multi layer perceptron, and thus the main target of backpropagation.

We take the basic example

$$X \odot \theta = \sigma$$

where we have the input $X \in \mathbb{R}^{K,L}$, filter $\theta \in \mathbb{R}^{L,M}$, and output $\sigma \in \mathbb{R}^{K,M}$.

Given the error $\frac{\partial L}{\partial \sigma} \in \mathbb{R}^{K,M}$ we want to find the errors w.r.t to X: $\frac{\partial L}{X}$, and θ: $\frac{\partial L}{\theta}$.

Following the concept of backpropagation [44]:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial \sigma} \odot \theta^T \tag{7.8}$$

$$\frac{\partial L}{\partial \theta} = X^T \odot \frac{\partial L}{\partial \sigma} \tag{7.9}$$

Note: the matrix multiplication between two tensors with shapes (K, L) and (L, M) will have the shape (K, M).

2D Convolution

We have a given convolution between $X \in \mathbb{R}^{H,W}$ and $\theta \in \mathbb{R}^{h,w}$:

$$X \otimes \theta = \sigma$$

and the error of $\sigma \in \mathbb{R}^{H-h+1,W-w+1}$: $\frac{\partial L}{\partial \sigma}$. We want to find the errors $\frac{\partial L}{\partial X}$ and $\frac{\partial L}{\partial \theta}$ of X and θ respectively.

The in-grain understanding of a convolution follows:

$$\sigma_{i,j} = \sum_{\Delta=1}^{h} \sum_{\delta=1}^{w} \left(X_{i+\Delta,j+\delta} * \theta_{\Delta,\delta} \right) \tag{7.10}$$

Following the work of Pavithra Solai [46], we have that:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial \sigma} \otimes_{\boxdot} 180^\circ rot\{\theta\} \tag{7.11}$$

and

$$\frac{\partial L}{\partial \theta} = X \otimes \frac{\partial L}{\partial \sigma} \tag{7.12}$$

Here the operation $\otimes_{\boxdot}$ is the operation that zero-pads the input such that the output has the same shape as the original input X.

2D Convolution with Channels

The more complex cases with channels have an at-surface similar operation:

$$X \otimes \theta = \sigma$$

where the shapes are of higher dimensionality:

$$\begin{aligned} X &\in \mathbb{R}^{C_{in},H,W} \\ \theta &\in \mathbb{R}^{C_{in},C_{out},h,w} \\ \sigma &\in \mathbb{R}^{C_{out},H-h+1,W-w+1} \end{aligned} \tag{7.13}$$

Defining the behaviour of multi-dimensional convolutions we have:

$$\sigma_{c,i,j} = \sum_{\triangle=1}^{C_{in}} \sum_{\Delta=1}^{h} \sum_{\delta=1}^{w} \left(X_{\triangle,i+\Delta,j+\delta} * \theta_{\triangle,c,\Delta,\delta} \right) \tag{7.14}$$

We have to address the extra dimensions in regard to the backward pass.

To get an error w.r.t. to the input with the input's shape, one must transpose the outer dimensions of the kernel [41], giving:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial \sigma} \otimes_{\boxdot} \leftrightarrow\updownarrow \left(180^\circ rot\{\theta\}\right) \tag{7.15}$$

As it is already implemented in Poplar, we do not have to calculate $\frac{\partial L}{\partial \theta}$.

Gated Linear Units

The Gated Linear Unit (GLU):

$$GLU(X) = (X \otimes W_a + b_a) * \sigma(X \otimes W_b + b_b) \tag{7.16}$$

where σ is the sigmoid function, W_a and W_b are weights, and b_a and b_b are biases, has a derivative defined as

$$\nabla[X * \sigma(X)] = \nabla X * \sigma(X) + X * \sigma'(X) \nabla X \tag{7.17}$$

according to Yann N. Dauphin, *et al.* [12].

Batch Layer Normalization

The batch layer normalization, detailed in algorithm 2:

$$\delta X = \frac{(X-\mu)}{\sqrt{\sigma + 10^{-6}}} * \gamma + \beta \tag{7.18}$$

has the derivative

$$\begin{aligned}
\delta\sigma^{-1} &= \sum^{N}(\delta Z * \gamma) && (i)\\
\delta\hat{X} &= \delta\gamma x * \gamma && (ii)\\
\delta x\mu_1 &= \delta\hat{X} * \sigma^{-1} && (iii)\\
\delta\sigma &= \frac{1}{2*\sqrt{\sigma+\epsilon}} * \frac{-1}{(\sigma+\epsilon)^2} * \sum^{N}(\delta\hat{X} * (X-\mu)) && (iv)\\
\delta sq &= \frac{1}{N} * \begin{pmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{pmatrix}_{(N,D)} * \delta\sqrt{\sigma} && (v)\\
\delta x\mu_2 &= 2 * (x-\mu) * \delta sq && (vi)\\
\delta x1 &= (\delta x\mu_1 + \delta x\mu_2) && (vii)\\
\delta\mu &= -(\sum^{N}\delta x1)^{-1} && (viii)\\
\delta x2 &= \frac{1}{N} * \begin{pmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{pmatrix}_{(N,D)} * \delta\mu && (ix)\\
\delta X &= \delta x1 + \delta x2 && (x)
\end{aligned} \tag{7.19}$$

according to Stanford [15].

7.3.3 Terminal State

Efforts to make the backwards pass functional was ultimately left in an ambiguous state, requiring further development to implement a contemporary learning algorithm. The original STGCN [56] model used RMSProp and ADAMOptimziers, and these two would be the desired targets of such further development.

Chapter 8

Results

8.1 Experiment I: Verifying Forward Pass

For the model to be of any interest, it is essential that it produces the correct results.

This experiment is to verify that given the same input variables, the model produces identical results to the original[56] code.

To actually be able to verify the results, some measures have to be employed to

It is not the intent to measure the accuracy of the trained data in this experiment, rather that it is the same mathematical series of operations as the original[56].

- **Trained Variable Insertion:** Trained variables from the original code is to be employed in their respective places in the produced model.

- **Derandomization:** Due to the random results of the dropout function, the pipeline becomes inherently unverifiable across this operation. As such, the results of the dropout operation from the original code is supplanted with the output of dropout of the IPU variant.

The dropout operation is the last operation of the ST-Conv-block, see fig. 5.2. To verify that the aforementioned derandomization is not hiding error, I verify the accuracy of the pre-randomization, hence referred to as *predrop*. Following this we have three important points to verify: predrop 1, predrop 2, and the model output.

To calculate how accurate the results are at any given step, we use the the equations 6.1. For quick reference, it is reiterated here:

$$\begin{aligned}
\Delta_M &= M_{c++} - M_{py} \\
E_- &= MIN(\Delta_M) \\
E_+ &= MAX(\Delta_M) \\
E_r &= (E_-, E_+)
\end{aligned}$$

Error is calculated as outlined in equation 6.1, where Δ_M is a matrix with the point-wise subtraction of M_{py} (the python computed matrix) from M_{c++} (the native computed result); further the error range E_r is defined as the range between the biggest negative error and biggest positive error.

Point	**Error**$^{Pre}_{Min}$	**Error**$^{Pre}_{Max}$
Drop 1	-7.62939e-06	6.67572e-06
Drop 2	-1.04904e-05	1.04904e-05
Output	-8.34465e-07	8.34465e-07

Table 8.1: Error

The results of this process are displayed in table 8.1. Due to the error after randomization in this case being zero (as it is parsed over), one should consider the aggregation of error over these elements as a genuine concern.

As there is inherently concern with the aggregation of errors, we can see that by the small nature of the error in predrop, it should not explode.

8.2 Experiment II: Timing Forward Pass

The second experimental setup employed is to run the STGCN forward pass N times. Due to the very short elapse of the forward pass a relatively high *odd* N is suggested.

We compare the following entries:

Code Version	**Machine**	**Notes**	**Core Count**
STGCN^{GPU_T}	NVIDIA Tesla T4	Code from [56]	2560_1
STGCN^{GPU_V}	NVIDIA V100 SXM3	Code from [56]	5120_1
STGCN^{IPU_1}	IPU Colossus 2	1 IPU(s)	1472_2
STGCN^{IPU_2}	IPU Colossus 2	2 IPU(s)	2944_2
STGCN^{IPU_4}	IPU Colossus 2	4 IPU(s)	5888_2
STGCN^{IPU_8}	IPU Colossus 2	8 IPU(s)	11776_2

Table 8.2: Experiment II: Configuration Map

$_1$: Cuda Cores, $_2$: IPU Tile

The configurations of the STGCN are listed in table 8.2.

The only parameter introduced is to restrict the batch size to 30 [1] , we run the model 101 times.

[1]This batch size is only possible on commits of the repository that predate the implementation of the backwards pass.

Code Version	Median	Average	Min	Max
STGCN^{GPU_T}	11.41119	11.69903	11.24454	39.26921
STGCN^{GPU_V}	4.30584	4.66168	4.18591	33.7596
STGCN^{IPU_1}	1.03255	1.03333	1.00372	2.54524
STGCN^{IPU_2}	3.83186	3.84557	3.80070	5.10547
STGCN^{IPU_4}	2.79797	2.82104	2.77530	5.39448
STGCN^{IPU_8}	4.7773	4.79508	4.74420	8.39822

Table 8.3: Experiment II: Results in *ms*

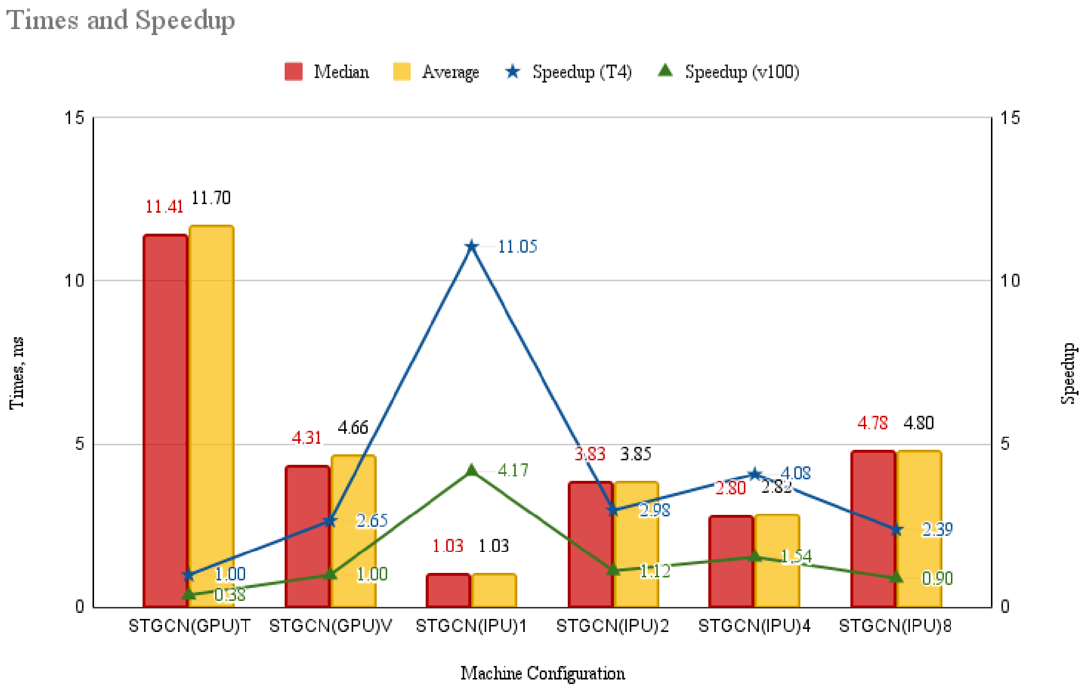

Figure 8.1: Experiment II: Speedup

Speedup (T4) and (V100) are calculated in respect to the *median*, they are graphed to the right vertical axis. Speedup T4 is in respect to STGCN(GPU)T, ie. STGCN^{GPU_T} and Speedup V100 is in respect to STGCN(GPU)V, ie. STGCN^{GPU_V}.

The results are tabulated in table 8.3 and graphed out in figure 8.1. There are several observations that can be made immediately:

- **Powerful Initial Speedup:** The speeedup over the v100 are explained with the FP32 statistics of both devices. The Nvidia V100 SCM3 has a theoretical FP32 performance of 15.67 TFLOPS, where the IPU boasts a performance of 62.5 TFLOPS. The roughly 4 times higher FP32 is seen in the speedup of roughly 4.
- **Slowdown on secondary chip:** A significant slowdown when

employing two IPUs is emblematic of communication bloat, however, as the number of tiles are doubled and the relative speedup from one IPU to two IPUs is $\approx$ .25 we see that the number of tiles does not explain this slowdown.

- **Relative Speedup on third and fourth IPU:** While not a significant speedup compared to a single IPU, four IPUs executes significantly faster than two. This suggests that the overhead of using multiple IPU's is partially alleviated by increasing the number of cores, or at the very least that it is less significant.

  The expected result of running on multiple IPU's should be a steady speedup-decay as the number of IPU's increases. This is mostly tied to communication[7]. This is to say that the expected performance at four IPU's should fall at roughly *4.35* assuming a linear degradation. As exponential decay is more likely, it should be a little faster yet.

- **Slowdown on fifth through eighth chip:** A preface to this observation is that the M2000 machine architecture has four chips on a board[16], meaning that as we transition from four to eight chips we are adding another layer of communication between the most distant elements. The further decaying speedup of this transition is then understandable, particularly in the scope of the previous major decay from one to two IPUs.

However, there are some significant caveats to these results.

In the related works section we address similar strong results on the IPU, however most work on the IPU is currently limited to problems on one accelerator (see section 4). One of the major benefits of GPUs is superior scalability, something that is significantly more difficult with the current iterations of the IPU.

GPU's can work on far larger problems, exploiting scalability to increase throughput. Illustrated in our results are the challenges of implementing IPU code that can exploit multiple chips at the same time. Considering the IPU-Machine (M2000) consists of no less than four chips [16] the issues with effectively applying them in tandem -even over just four chips- becomes a more significant drawback of the architecture.

The overall IPU v. GPU performance when projecting high numbers of both is less favourable to the IPU than initial numbers on single IPU v. GPU performance may suggest. This only pertains to large models, as if possible, data-parallel experiments are projected to still be far more efficient on IPUs.

In conversation with Alexander Titterton, an AI Field Applications Engineer at Graphcore, the relative speedup at four chip execution was suggested caused by the extra PCI-connections (See Figure 3.2), postulated that the extra memory transfer flexibility reduced the overall communications time.

8.3 Experiment III: Estimating Training Time

Using the current implementation of the back-propagation we can make a rough estimate of how long a training process could take on the IPU.

The experiment is attempted under the following conditions for both python and IPU variant of the STGCN's:

○ Batch Size	10
○ Epochs	50
○ Batches	911

Note: The number of batches are the number of trainable series in the dataset.

The STGCN for GPU is run on a NVIDIA Tesla T4.

Further, the learning rate is set to *0* for the IPU variant, while the GPU runs with learning rate 10^{-3}. This is to protect the IPU pipeline from numbers that could impact the cycle count.

A concern is that altering the learning rate will alter the performance. To verify both that the GPU performs similarly over a normal learning rate[2] and a learning rate of 0, and to test the relative performance between an IPU and a GPU both with learning rate 0, the GPU was also tested at 0 learning rate.

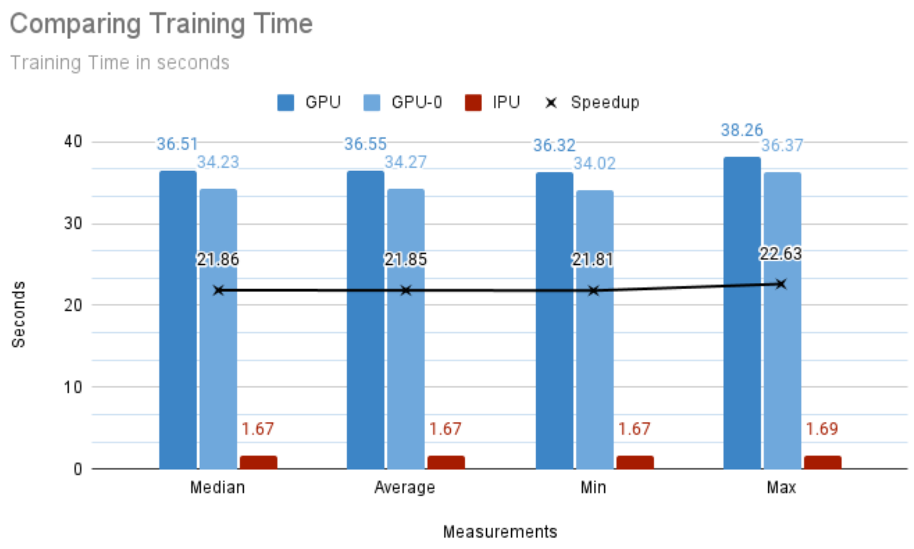

Figure 8.2: Training Time

GPU-0 and the IPU are both tested with learning rate 0.

The GPU performs very similarly over learning rates 0 and 10^{-6}, indicating that the performance numbers are not off base.

The $\approx$ 22 time speedup seen in figure 8.2 is very promising. However, the comparison hides some caveats.

[2]Standard Learning Rate: 10^{-3}

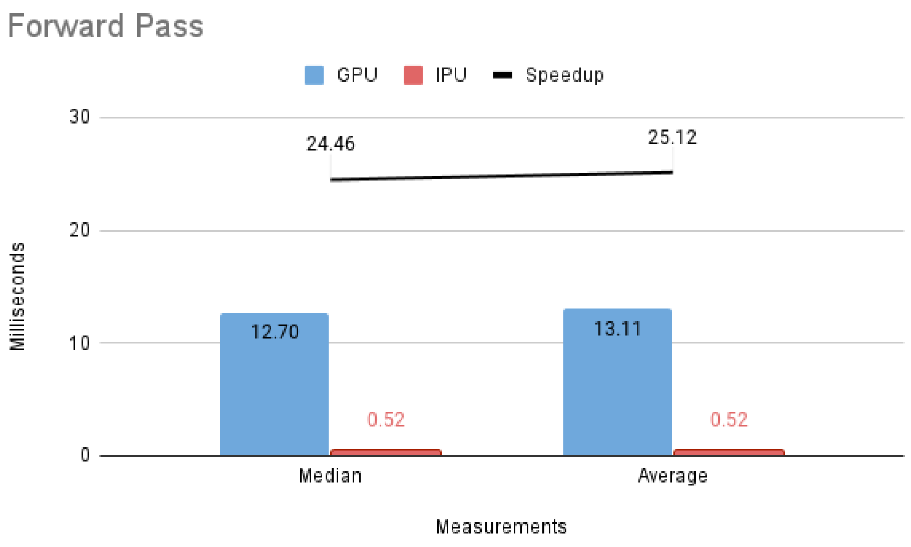

Figure 8.3: Forward Pass

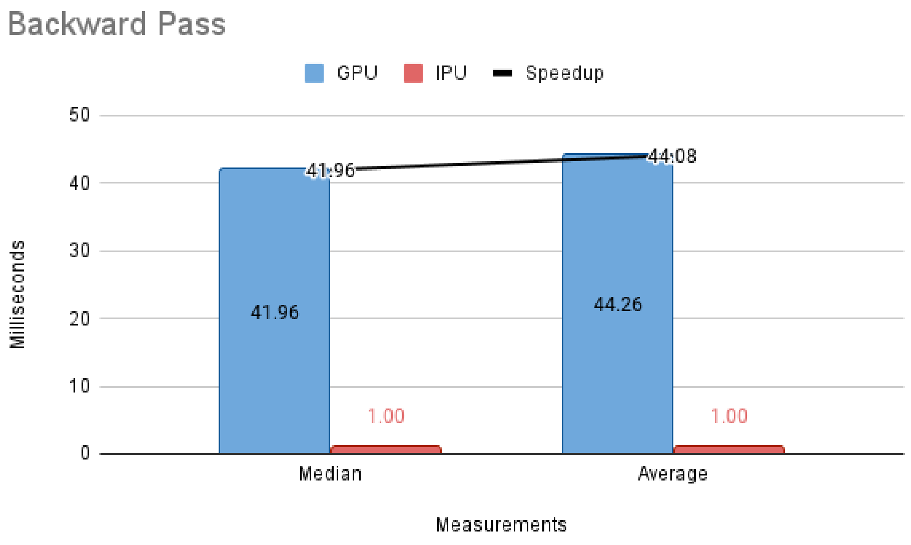

Figure 8.4: Backward Pass

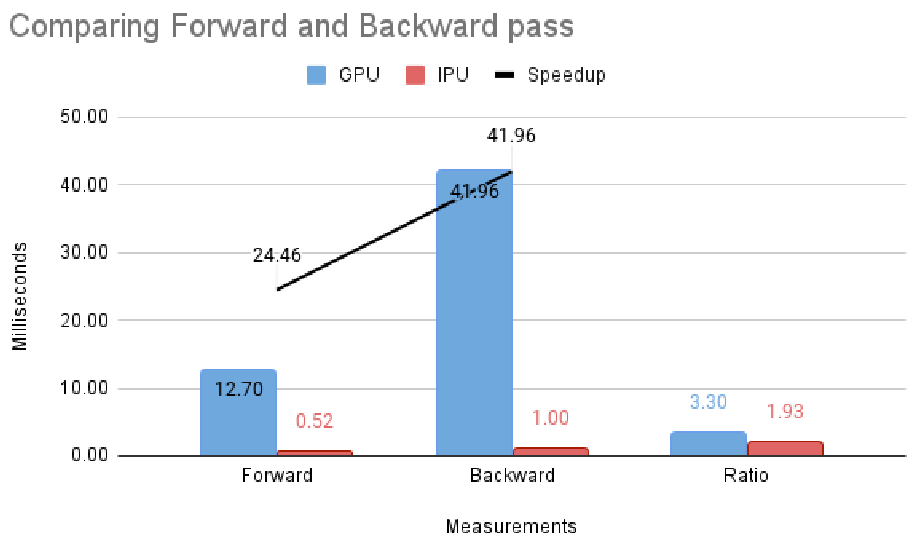

Figure 8.5: Comparing Passes

The ratio represents the difference in speed between forward and backward passes.

The diagnostics figures 8.3, 8.4, and 8.5 illuminate the situation.

Note that the median performance of the IPU forward pass is twice as fast as it was with batch size 30, but the GPU has no performance boost from the lower batch size. Considering the speedup of 11 on batch size 30 between these two devices, it then follows that if one has not changed and the other is twice as fast that the final performance of the IPU in this case should be roughly 20 times faster.

That the GPU operates similarly at batch size 10 and 30 makes sense, the GPU workload is dependent on the size of its warps due to vector parallelization. The IPU has no such dependency. Assuming the GPU performs similarly when running with a batch size of 30 the real throughput speedup is likely closer to ≈ 7.

Keeping in mind that the backward pass of the IPU implementation is nothing more than an estimation of the backpropagation, we should note that the ratio between forward and backward passes (as seen in figure 8.5) are very different from the GPU to the IPU.

Due to the Python implemented backpropagation being a blackbox, it is difficult to verify to which degree this ratio is dependent on the Python implementation. We can assume that the backward propagation on the IPU is the outlier, in which case the *Backward/Forward* ratio can be used to extrapolate a backwards pass IPU performance that might be more accurate.

Multiplying the duration of a forward pass on the IPU with the ratio experienced between forward and backward passes on the GPU gives us that a *possibly* more accurate backward pass time is $1.7ms$ approximately[3].

Assuming the training time is accurate and we observe a rounded down speedup of 20 the GPU would still have to operate equally fast at batches an order of magnitude larger to execute at similar efficiency.

It is concluded that training neural networks on the IPU is faster than standard GPU implementations.

3

$$\text{Estimate} = T_{IPU}(Forward) * \frac{T_{GPU}(Backward)}{T_{GPU}(Forward)} = 0.52 * 3.3 = 1.716$$

Part III

Discussion

Chapter 9

Implementation Observations

9.1 4D array index order

This is in regards to the STGCN code [56].

The 2D convolution: $\otimes$, takes in the simplest instructive case a grey-scale image (ie. a 2D matrix) and applies a 2D filter. The dimensions of the image are height (H) and width (W). In the case of colored images (f.ex RGB) the image is conceptualized as multiple grayscale images in different channels (C), creating a 3D cuboid. A visualization of this for a possible RGB image is displayed in figure 9.1.

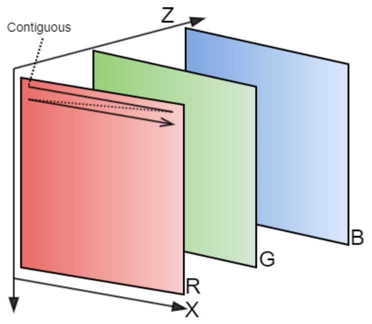

Figure 9.1: RGB Image, configured Channel$\times$Height$\times$Width

Here $Z \rightarrow C, Y \rightarrow H, X \rightarrow W$. The line labeled "Contiguous" display the order of data entries in sequential memory.

In the case of machine learning, multiple images are processed in a batch. To avoid iteration over the batch size (N), the entire set is often configured as a 4D matrix.

The default order for this used by tf.nn.conv2d[1] is NHWC: batch, height, width, and channels. The inner cuboids (HWC) of the NHWC format is illustrated in figure 9.2. This is fine for point-wise operations, and likely very unproblematic for images with very few channels (like 3, which is common for RGB images).

[1]Link: https://www.tensorflow.org/api_docs/python/tf/nn/conv2d

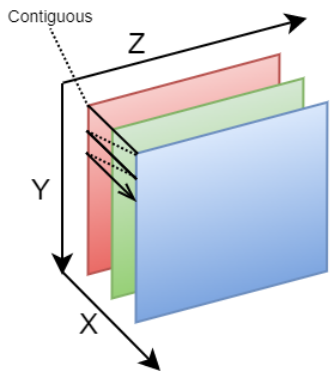

Figure 9.2: RGB Image, configured Height×Width×Channels

Here $Z \rightarrow H, Y \rightarrow W, X \rightarrow C$

This is however not the case for the STGCN, which employs multiple different number of channels. The channels for the STGCN represent the width of their internal layers, and their default values range from 1 to 128[2].

The use of NHWC mainly becomes a problem when memory accesses are not being made across channels, but rather across the shape of the image.

Another note on the ordering here is that the STGCN code makes frequent use of transpositions, some of which would be redundant had the matrix been in NCHW format throughout, which is the cause of some redundancy.

This issue is also something that becomes more important on the IPU as the IPU has stricter limitations to the convolution inputs, see section 10.2.1.

9.2 Memory Saturation

The STGCN code in its most naive implementation is very memory intensive, demanding upward of 1000 mb of data. While the initial tensor input (by default) is of dimensions: [50, 12, 228, 1][3] and the second dimension is decreased to 1 throughout the two ST-Conv Blocks, the channels are expanded to 128. For brevity, assuming each layer output had its own tensor output, the IPU has little capacity to contain the tensors.

The simplest way to work around this is to reduce the batch size, however, the strict limitations are a general concern for large algorithms.

For implementations where large data-segments are non-recyclable, this issue is far more prevalent. A good example of this is the backwards pass of a multilayer perceptron.

[2]Specifically there are four temporal convolution layers over two spatio temporal blocks, the layers do the following channel conversions: $1 \rightarrow 32$, $32 \rightarrow 64$, $64 \rightarrow 32$, and $32 \rightarrow 128$.

[3][Batch Size × History × Depth × Channels]

9.3 Issues in operations with inherent reductions

Due to the premium any memory segment comes at, the *XInPlace*(...) operations (f.ex: *addInPlace(...)*, *subInPlace(...)*, *mulInPlace(...)*, *divInPlace(...)*), are very useful. They become particularly useful in the case of the back-propagation, where we want to adjust existing variables. We are going to constrain this subject to the *addInPlace*(...) function, although suffice it to say the same traits are applicable to all similar *XInPlace*(...) functions.

$$\begin{aligned} &\beta \in \mathbb{R}^N \\ &\delta^\beta \in \mathbb{R}^{B\times N} \\ \text{Add-in-place:}\quad &\beta \leftarrow \beta + \delta^\beta \end{aligned} \tag{9.1}$$

The overview of the add-in-place (omitting calculations to get δ^β) is outlined in equations 9.1. Here B is the batch-size of the learning set, and N is the length of β.

Note that δ^β does not go into β, and so the operation is more adequately explained as the operation:

$$\beta_n \leftarrow \beta_n + \sum_b^B \delta^\beta_{b,n} \tag{9.2}$$

In poplar, these operations are allowed and should in theory work most of the time. However, due to tile-mapping issues, it will not execute the reduction first, but rather try to write to the space multiple times, ie:

$$\begin{aligned} &\beta_1 \leftarrow \beta_1 + \delta^\beta_{1,1}, \;\; \beta_1 \leftarrow \beta_1 + \delta^\beta_{2,1}, \quad ... \; \beta_1 \leftarrow \beta_1 + \delta^\beta_{B,1} \\ &\beta_2 \leftarrow \beta_2 + \delta^\beta_{1,2}, \;\; \beta_2 \leftarrow \beta_2 + \delta^\beta_{2,2}, \quad ... \; \beta_2 \leftarrow \beta_2 + \delta^\beta_{B,2} \\ &\qquad\qquad ... \\ &\beta_N \leftarrow \beta_N + \delta^\beta_{1,N}, \;\; \beta_N \leftarrow \beta_N + \delta^\beta_{2,N}, \quad ... \; \beta_N \leftarrow \beta_N + \delta^\beta_{B,N} \end{aligned} \tag{9.3}$$

This set of operations is theoretically fine if operations can be computed atomically[4] with respect to elements in β. However, the Poplar system responsible for inPlace operations does not take this into considerations, and may on its own distribute the tasks across the tiles in such a manner that multiple tiles will be writing to the same elements in β.

Due diligence: I do commend them for avoiding reductions, as reductions are inherently some of the most time-consuming operations across high-count multi-core architectures, exponentially worse on multi-IPU's [7]. However, in this case the reduction is recommended.

9.4 Backpropagation memory consumption on the IPU

The IPU operates with significantly less memory (9MB) capacity than contemporary GPU's (NVIDIA Tesla T4: 16GB). As a consequence of this,

[4]I.e writes are protected from race-conditions.

the IPU operates with a far smaller capacity for model complexity.

During the backward pass calculating error with respect to a filter in matrix multiplication and convolutions are calculated using the input (See Equations 9.4 and 9.5 respectively).

$$\frac{\partial L}{\partial \theta_m} = X^T \odot \frac{\partial L}{\partial \sigma} \tag{9.4}$$

$$\frac{\partial L}{\partial \theta_c} = X \otimes \frac{\partial L}{\partial \sigma} \tag{9.5}$$

Here X is the input of the forward pass, and $\frac{\partial L}{\partial \sigma}$ being the error with respect to the output of the convolution.

The implication of this is that a machine learning algorithm will have to *store* the inputs of any multi-parameter function. With this the amount of data one can *recycle* in the implementation is reduced by a significant margin.

The IPU, with its far lower total memory capacity, is thus far more fitted to algorithms that are more friendly to memory recycling.

Chapter 10

Comments on Poplar and the IPU

This chapter is dedicated to noting final thoughts and comments on the Poplar libraries for the IPU, the IPU itself, and its supporting software.

10.1 Useful traits

POPLAR has many strong traits, particularly its performance.

10.1.1 Recognizable and intuitive tensors

The IPU tensors work much like familiar data structures from python, particularly reminiscent of the tensorflow objects. This is natural as Tensorflow's keras package has support for running on the IPU.

The aforementioned tensors cannot be accessed as basic data structure, much like other systems designed for lazy execution, but has all the wanted basic operations inbuilt: slicing, custom multidimensional transpositions, reversal across dimensions, etc., making it fairly intuitive when translating algorithms to POPLAR.

The only unfavorable interaction throughout development with the tensors is that their transpose functionality is not ideal. As is the *transpose* function *only* performs transpositions on 2D matrices, and any other transpose operation over ND objects is instead the functionality of the function *dimShuffle*. As dimShuffle is not a recognized synonym of transpositions, is further ambigious as a term, *and* the documentation of dimShuffle does not contain the phrase transpose[1] the function is not easily found in non-extensive searches of the Tensor documentation unless you know what you are looking for.

10.1.2 Native Support for Most Operations

Another strong side of the IPU is its inbuilt support for all conventional machine learning operations, making it possible to implement machine

[1]As of documentation on SDK 2.1.0

learning algorithms, even in C++, with minimal custom codellet creation.

This is however limited to strict machine learning operations, or more broadly defined in the terms of dense operations, ie. operations over sections of data with predictable memory accessing. As seen in [7] sparse operations are still behind dense operations in implementation.

10.1.3 High Performance

In section 8.2: "Experiment II: Timing Forward Pass" the computational power of the IPU is displayed, boasting a per-core efficiency of 19 times the GPU[2]. This efficiency is even more notable as the IPU Tiles are not warped like CUDA Cores, and can work on individual tasks. The benefit of which is further deliberated on in section 10.1.4: "Graph Applications.

While the aforementioned performance can be suffocated by cross-IPU latency [3]these challenges are likely to dissipate as the architecture matures, and sufficient tools are created to give developers easy access to more efficient tile-mapping.

10.1.4 Graph Applications

Graph algorithms, like the ones mentioned in chapter 2: "Background I: Graph Neural Networks", are growing in popularity as sparse data structures are becoming more popular. While GPU's can execute sparse operations effectively if empowered with sufficient optimization efforts, the IPU stands to offer on-demand high efficiency graph operations in the future.

10.2 Comments

In this section I attempt to describe the issues that have arose when using the Graphcore API, and the IPU in general.

10.2.1 Documentation Deficiencies

Important omissions

The Graphcore API has extensive support for many operations, but important features of these operations are some times not outlined in the documentation. There are two particular examples of this that underline the importance of these omissions, and how they can make the documentation at times confusing:

The **first** is attached to the use of convolutions, this has already been addressed in subsection 6.4, but in brief: a convolution requires an Object ConvParams (Convolution Parameters) and have further 3 functions to create tensors for input, weights, and biases based on the aforementioned parameters and a function that adds created bias to a convolution output.

[2]NVIDIA Tesla T4

[3]further addressed in section 8.2

Nowhere in the documentation[4] does it explain the differences between the functions that create input, weights, and biases and the regular tensor creation functions, or addBiases from a simple addition operation.

In conversation with the GraphCore support, it came to the fore that the distinction between the create[Input/Weights/Biases] functions and the standard functions is custom memory-mapping, an essential distinction.

The **latter** is in regards to the dropout function, this is also mentioned in subsection 6.4. The dropout function has the interface:

```
Tensor dropout(poplar::Graph &graph,
               const poplar::Tensor *seed,
               const uint32_t seedModifier,
               const poplar::Tensor &input,
               const poplar::Tensor &reference,
               double keepProbability,
               double scale,
               bool outputClonesRef,
               poplar::program::Sequence &prog,
               const poplar::DebugContext
                        &debugContext = {})
```

Figure 10.1: Dropout Function Signature, POPLAR

The function does the same as similar functions in different systems like Tensorflow and PyTorch: multiplying the input with a random mask of ones and zeros, the probability of ones is given in keepProbability.

Most important is the parameter "reference", the reference was described in the documentation as:

> "A tensor that specifies the layout of the output tensor. Must be the same shape as the input."

It is notably omitted *what* the reference is a reference for in the documentation, however in conversation with GraphCore support it was explained that this was for the purpose of memory mapping.

In summation, the GraphCore API has a lot of significant and useful features, but its documentation often omits to explain said features. This in turn can make it arduous to implement otherwise simple functionalities, or simply difficult to employ efficiently.

The Backward pass

One of machine learning's most staple operations is the backward pass informed by backpropagation[44], used to *train* weights and biases. In many frameworks there are advanced functions to calculate the backward

[4]At the time of writing

pass and inherent gradients from the logical execution graph, this hand's off approach is not found in the Graphcore API for C++.

Note that when using TensorFlow with Python for IPU, such functions are prebuilt and inherent to TensorFlow, thus requiring no implementation for native code. This makes implementation of IPU code from python far more convenient. This is why I do not count it as a grievance that similar functionality does not exists for C++ Poplar, instead my grievance is with the *existing* support for backwards pass and gradient calculation not having any easily accessible resources.

There are functions in the documentation that seem custom made specifically for the aforementioned purposes, particularly prevelant in the *popnn*::**Lstm** package. In the aforementioned package there are functions like *lstmBwd(...)* which by documentation reference running a backward pass on an Long Short-Term Memory network, using that operations on the IPU can have an alternate functionality defined for their backward operation.

With what appears to be extensive support for these machine learning tools, it is disappointing that the resources necessary to efficiently use them, or to know their inherent limitations[5], is sorely missing.

Convolution

Convolution is one subject the aforementioned issues with documentation intersect.

Strange choices in order: The documentation makes a strange choice to place the explanation of the convolution options (ie. OptionFlags) in the documentation of the createWeights(...) function instead of the more logical convolution(...) function. The createWeights documentation does appear *before* convolution(...), which slightly justifies it, however createWeights also occurs *before* the createInputs(...) and createBiases(...) documentation, both of which should be before createWeights in alphabetical order and one of which (createInputs(...) is another candidate to have the option flag documentation.

The Convolutional Parameters: Not to be confused with the Option flags, the Convolutional Parameters are stored in a class object ConvParams. The ConvParam class requires a series of inputs: dataType, batchSize, inputFieldShape, kernelShape, inputChannels, outputChannels, and numConvGroup. These parameters are of course necessary to implement an effective convolution, however they are not sufficiently described in the documentation to make implementation easy. Furthermore, when inspecting the parameters one finds that due to the very strict dimensional ordering of the input, weights, and kernel they are also to a degree redundant. The input is required to have the shape {batchSize, inputChannels,

[5]f.ex: Which functions does not have a backward functionality,

...} where ... is the inputFieldShape. Giving the input shape as a full vector should be sufficient to fill these out, making for a more concise overall documentation.

The *create* functions: The three functions createBiases(...), createInputs(...), and createWeights(...) are designed to create tensors of the appropriate dimensions and more importantly automatically map the tensors such that tile-mapping is optimal for the following convolution and adding of the bias. While this is very useful, the tile-mapping feature is notably omitted from the documentation throughout, leaving one with little clue as to *why* they exist.

The backward convolution support: First it is worth noting that there are functions in the ecosystem of the convolution that suggest the support for backwards propagation, or straight up makes it much easier. These include calculateWeightDeltas(...), convolutionWeightUpdate(...), convolutionBiasUpdate(...), and fullyConnectedWeightTranspose(...). This is to a certain degree essential to appreciate, as it is not necessary for the base convolution. The weight delta should be calculated:

$$\delta W = i \oplus \mathrm{rot}_{180^\circ}\{E\}$$

where i is the input and E is the error gradient. Meaning that the convolution environment as *some* support for the backwards pass convolutions ($\oplus$). However, it does not entail details on the final convolution, the one designed to acquire the back-propagated error gradient w.r.t. the input.

Due to the immense thoroughness with which the convolution environment has been engineered, it seems unlikely that some support for this convolution too is omitted, particularly as the backwards convolution implemented independently would require an entirely new environment overlapping at points with the forward environment, undermining the forward pass.

10.2.2 Advanced Tilemapping Support

There are two promoted ways to map tensors to tiles: Graph.setTileMapping(const Tensor &t, unsigned *tileNum*) and Graph.setTileMapping(const Tensor &t, const TileToTensorMapping &mapping). The former of these two will map a tensor (or potentially a tensor slice) to a *specific* tile, and the latter uses either the tilemapping of another tensor (acquired through getTileMapping(...) or getVariableTileMapping(...)) or VariableMappingMethod, an enum that prescribes how to devide the tensor among the nodes. Note that VariableMappingMethod is an enum with only two variations: NONE and LINEAR, where NONE defers to setTileMapping and LINEAR "The variable will be spread evenly across the tiles with the element ordering matching the tile number ordering.".

This means that there are two promoted ways to map data: fine-grained by hand or naively. There are other ways too, which should be noted, amongst them the aforementioned create(Input/Weights/Biases) mentioned in subsections 6.4 and 10.2.1, however these methods are inscrutable to the developer, making it difficult to otpimize a pipeline.

With the importance of efficient memory management in high time-complexity algorithms on multicore architectures it would be desirable with more variable mapping methods, or a resource map over different functions that manage memory mapping on some to allow programmers to more efficiently map memory pipelines.

The biggest motivator of large-scale mapping strategies are reduction operations. While we have so far looked at single IPU implementations, previous work on the IPU[7] and results from this project has illustrated that multi-IPU code suffers significantly higher from cross-IPU latency aggravated further during reduction.

The first suggestion are *variable* mapping methods that work to map within certain constrains, primarily to place elements of a dimension on the same tile. On all cases where this is possible this is posed to eliminate cross-IPU transfer latency with reductions on these dimensions.

Secondly, I would suggest a customized memory distribution based on per-tile memory density. By this I mean a function to map a tensor across the tiles attempting not to spread the data too sparsely, as this is likely to also cause issues.

A third suggestion is to have functions to optimize tile-mapping with natural constraints. While this is unlikely to be optimal, it is highly likely to be more optimal than the naive linearly mapped tensors.

10.2.3 Tensor visualisations

While not a large concern: on memory-intensive tasks where recycled memory becomes a necessity it would be useful to be able to view the memory pipeline and review its integrity.

10.2.4 Sequence and Program Inspection

One feature that is sorely lacking from the API is the ability to review a program and further inspect the input/output tensors of such programs at run-time.

The best example of the usefulness of this feature would be viable implementations of back-propagation as a *product* of an existing sequence of programs. Currently the back-propagation has to be implemented at the same time as its corresponding layer (or storing variables for later facilitation), creating a potentially bloated code without cross-project re-usability.

10.2.5 Multi-IPU support

The issues with multiple IPUs seen in section 8.2: "Experiment II: Timing Forward Pass", and further addressed in section 10.2.2: "Advanced Tilemapping Support" still remains one of the biggest drawbacks of the

IPU: demanding high amounts of custom optimization for efficient multi-IPU systems in poplar.

10.2.6 Poplar as an algorithm development tool

While it is possible to develop graph algorithms [7] and Machine Learning Algorithms in Poplar, Poplar is in large designed for internal use at Graphcore. This has come out in conversations with people at the company after most of the experiments were concluded.

Poplar is the foundation for their internal frameworks, i.e. their support for PyTorch and TensorFlow, and intermediate levels between Poplar and these frameworks are responsible for handling tile-mapping, multi-IPU handling, and various memory and computational optimizations. Most of the published computational results from Graphcore are achieved in these higher level frameworks.

This is to say that Poplar is not designed to be (or maintained as) a library for users to implement machine learning algorithms in. From members of the Graphcore applications teams it has been suggested that work in the domain is better

Chapter 11

Conclusion

The primary contribution in regards to this work is the implementation of the Spatio-Temporal Graph Convolutional Network in Graphcore's Poplar framework. With the forward pass verified to the margin of rounding errors, this implementation can be employed to accurately compare the Graphical Processing Unit execution [56] performance of the original implementation with the Intelligence Processing Unit variation of the same code.

As shown in section 8.2 where the performance of the Nvidia V100 SCM3 and the IPU Colossus 2 are compared w.r.t. the STGCN it is shown that the IPU delivers on its promise of 62.5 TFLOPS performance when with its $\approx 4X$ speedup over the V100 and its 15.67 TFLOPS.

The secondary contributions is an attempt to approximate the learning rate of the machine learning algorithm by implementing back-propagation and RMSProp. While the numbers in regards to the results of these contributions are less reliable they are equally promising. The IPU is shown to deliver on its performance promises again.

The aforementioned result that the IPU performs with an $\approx 4X$ speedup over the V100 is the primary finding in regards to the hardware of the IPU. The dense operation matchup between the GPU and IPU is unfavourable to the IPU, and it outperforming the former highlights the promise it delivers on a true Graph Algorithm Accelerator.

Currently the main drawback of the IPU architecture is its still limited memory capacity. Currently models on the IPUs are limited in size by the constrain of the chip's on-board memory and the inhibiting memory transfers. Hopefully future variants of the IPU architecture will follow the current trend of increased memory capacity.

The technology lags behind w.r.t. the software: Poplar being a prohibitively low level framework for machine learning purposes. While not available when the project began, the higher level alternatives Popart, Poptorch, and its support for TensorFlow supposedly make the development significantly simpler. Particularly in the domain of graph algorithms does this bear an impact, constraining users between the lackluster support for sparse operations on modern high level frameworks and the low level Poplar systems.

Even with the challenges faced by the architecture and its accompanying software the IPU delivers on promised performance, and more importantly delivers a computational flexibility that is intrinsically not achievable with the architecture of a Graph Processing Unit. The result is an architecture that can compete with Graph Processors in their ideal condition and retain their performance into the sparse domain.

Of particular interest are other Graph Neural Networks to future research on IPU based models. The domain is relatively unexplored and the IPU is the primary technology addressing the computational challenges of the field.

Another direction research can be directed is optimization on the Intelligence Processing Units. Optimizations for Graph Processing Units are not inherently applicable to IPU and the IPU poses new and interesting topological problems, both for Tile to Tile communication optimizations and IPU to IPU optimizations. In fact optimization of the Static Computational graph and the tile-mapping within is perhaps one of the spaces with the greatest rooms for improvement.

Bibliography

[1] Aydin Buluc Alok Tripathy Katherine Yelick. *Reducing Communication in Graph Neural Network Training*. 2020. arXiv: 2005.03300v1 [cs.LG].

[2] James Atwood and Don Towsley. 'Diffusion-Convolutional Neural Networks'. In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee et al. Curran Associates, Inc., 2016, pp. 1993–2001.

[3] Davide Bacciu, Federico Errica and Alessio Micheli. 'Contextual Graph Markov Model: A Deep and Generative Approach to Graph Processing'. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden: PMLR, July 2018, pp. 294–303.

[4] J. M. Baker et al. 'Developments and directions in speech recognition and understanding, Part 1 [DSP Education]'. In: *IEEE Signal Processing Magazine* 26.3 (2009), pp. 75–80.

[5] J. M. Baker et al. 'Developments and directions in speech recognition and understanding, Part 1 [DSP Education]'. In: *IEEE Signal Processing Magazine* 26.4 (2009), pp. 78–85.

[6] Joan Bruna et al. 'Spectral networks and locally connected networks on graphs'. In: *arXiv preprint arXiv:1312.6203* (2013).

[7] Luk Burchard et al. 'iPUG: Accelerating Breadth-First Graph Traversals Using Manycore Graphcore IPUs'. In: *High Performance Computing*. Ed. by Bradford L. Chamberlain et al. Cham: Springer International Publishing, 2021, pp. 291–309. ISBN: 978-3-030-78713-4.

[8] Nicola De Cao and Thomas Kipf. *MolGAN: An implicit generative model for small molecular graphs*. 2018. arXiv: 1805.11973 [stat.ML].

[9] Shaosheng Cao, Wei Lu and Qiongkai Xu. *Deep Neural Networks for Learning Graph Representations*. 2016. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12423.

[10] Kyunghyun Cho et al. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 2014. arXiv: 1406.1078 [cs.CL].

[11] Hanjun Dai et al. 'Syntax-directed variational autoencoder for molecule generation'. In: *Proceedings of the International Conference on Learning Representations*. 2018.

[12] Yann N. Dauphin et al. *Language Modeling with Gated Convolutional Networks*. 2017. arXiv: 1612.08083 [cs.CL].

[13] Michaël Defferrard, Xavier Bresson and Pierre Vandergheynst. 'Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering'. In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee et al. Curran Associates, Inc., 2016, pp. 3844–3852. URL: http://papers.nips.cc/paper/6081-convolutional-neural-networks-on-graphs-with-fast-localized-spectral-filtering.pdf.

[14] Michaël Defferrard, Xavier Bresson and Pierre Vandergheynst. 'Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering'. In: *CoRR* abs/1606.09375 (2016). arXiv: 1606.09375. URL: http://arxiv.org/abs/1606.09375.

[15] *et al.* Fei-Fei li. *CS231n: Convolutional Neural Networks for Visual Recognition*. 2016. URL: http://cs231n.stanford.edu/ (visited on 29/08/2021).

[16] Karl Freund. *The Graphcore Second-Generation IPU*. Tech. rep. Moor Insights & Strategy, July 2020.

[17] C. Gallicchio and A. Micheli. 'Graph Echo State Networks'. In: *The 2010 International Joint Conference on Neural Networks (IJCNN)*. 2010, pp. 1–8.

[18] Jonas Gehring et al. 'Convolutional Sequence to Sequence Learning'. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, Nov. 2017, pp. 1243–1252. URL: http://proceedings.mlr.press/v70/gehring17a.html.

[19] Justin Gilmer et al. 'Neural Message Passing for Quantum Chemistry'. In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia: JMLR.org, 2017, pp. 1263–1272.

[20] Rafael Gómez-Bombarelli et al. 'Automatic Chemical Design Using a Data-Driven Continuous Representation of Molecules'. In: *ACS Central Science* 4.2 (2018). PMID: 29532027, pp. 268–276. DOI: 10.1021/acscentsci.7b00572. eprint: https://doi.org/10.1021/acscentsci.7b00572. URL: https://doi.org/10.1021/acscentsci.7b00572.

[21] Sepp Hochreiter and Jürgen Schmidhuber. 'Long Short-Term Memory'. In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://doi.org/10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[22] Ashesh Jain et al. 'Structural-RNN: Deep Learning on Spatio-Temporal Graphs'. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[23] Zhe Jia et al. *Dissecting the Graphcore IPU Architecture via Microbenchmarking*. 2019. arXiv: 1912.03413 [cs.DC].

[24] Yong-Yeon Jo, Sang-Wook Kim and Duck-Ho Bae. 'Efficient Sparse Matrix Multiplication on GPU for Large Social Network Analysis'. In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*. CIKM '15. Melbourne, Australia: Association for Computing Machinery, 2015, pp. 1261–1270. ISBN: 9781450337946. DOI: 10.1145/2806416.2806445. URL: https://doi-org.ezproxy.uio.no/10.1145/2806416.2806445.

[25] Thomas N. Kipf and Max Welling. 'Semi-Supervised Classification with Graph Convolutional Networks'. In: *CoRR* abs/1609.02907 (2016). arXiv: 1609.02907. URL: http://arxiv.org/abs/1609.02907.

[26] Thomas N. Kipf and Max Welling. 'Semi-Supervised Classification with Graph Convolutional Networks'. In: *CoRR* abs/1609.02907 (2016). arXiv: 1609.02907. URL: http://arxiv.org/abs/1609.02907.

[27] Thomas N. Kipf and Max Welling. *Variational Graph Auto-Encoders*. 2016. arXiv: 1611.07308 [stat.ML].

[28] Alex Krizhevsky, Ilya Sutskever and Geoffrey E Hinton. 'ImageNet Classification with Deep Convolutional Neural Networks'. In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[29] Matt J. Kusner, Brooks Paige and José Miguel Hernández-Lobato. *Grammar Variational Autoencoder*. 2017. arXiv: 1703.01925 [stat.ML].

[30] Johannes Langguth, Xing Cai and Mohammed Sourouri. 'Memory Bandwidth Contention: Communication vs Computation Tradeoffs in Supercomputers with Multicore Architectures'. In: Dec. 2018, pp. 497–506. DOI: 10.1109/PADSW.2018.8644601.

[31] Yann LeCun, Yoshua Bengio et al. 'Convolutional networks for images, speech, and time series'. In: *The handbook of brain theory and neural networks* 3361.10 (1995).

[32] Ron Levie et al. 'CayleyNets: Graph Convolutional Neural Networks with Complex Rational Spectral Filters'. In: *CoRR* abs/1705.07664 (2017). arXiv: 1705.07664. URL: http://arxiv.org/abs/1705.07664.

[33] Yaguang Li et al. 'Graph Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting'. In: *CoRR* abs/1707.01926 (2017). arXiv: 1707.01926. URL: http://arxiv.org/abs/1707.01926.

[34] Yujia Li et al. *Gated Graph Sequence Neural Networks*. 2015. arXiv: 1511.05493 [cs.LG].

[35] Yujia Li et al. *Learning Deep Generative Models of Graphs*. 2018. arXiv: 1803.03324 [cs.LG].

[36] Thorben Louw and Simon McIntosh-Smith. *Using the Graphcore IPU for traditional HPC applications*. Tech. rep. EasyChair, 2021.

[37] Thang Luong, Hieu Pham and Christopher D. Manning. 'Effective Approaches to Attention-based Neural Machine Translation'. In: *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Lisbon, Portugal: Association for Computational Linguistics, Sept. 2015, pp. 1412–1421. DOI: 10.18653/v1/D15-1166. URL: https://www.aclweb.org/anthology/D15-1166.

[38] Tengfei Ma, Jie Chen and Cao Xiao. 'Constrained Generation of Semantically Valid Graphs via Regularizing Variational Autoencoders'. In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 7113–7124. URL: http://papers.nips.cc/paper/7942-constrained-generation-of-semantically-valid-graphs-via-regularizing-variational-autoencoders.pdf.

[39] A. Micheli. 'Neural Network for Graphs: A Contextual Constructive Approach'. In: *IEEE Transactions on Neural Networks* 20.3 (2009), pp. 498–511.

[40] Lakshan Ram Madhan Mohan et al. *Studying the potential of Graphcore IPUs for applications in Particle Physics*. 2020. arXiv: 2008.09210 [physics.comp-ph].

[41] Son Nguyen. *A gentle explanation of Backpropagation in Convolutional Neural Network (CNN)*. 2020. URL: https://medium.com/@ngocson2vn/a-gentle-explanation-of-backpropagation-in-convolutional-neural-network-cnn-1a70abff508b (visited on 25/08/2021).

[42] Mathias Niepert, Mohamed Ahmed and Konstantin Kutzkov. 'Learning convolutional neural networks for graphs'. In: *International conference on machine learning*. PMLR. 2016, pp. 2014–2023.

[43] Shirui Pan et al. 'Adversarially Regularized Graph Autoencoder'. In: *CoRR* abs/1802.04407 (2018). arXiv: 1802.04407. URL: http://arxiv.org/abs/1802.04407.

[44] D. Rumelhart, Geoffrey E. Hinton and Ronald J. Williams. 'Learning representations by back-propagating errors'. In: *Nature* 323 (1986), pp. 533–536.

[45] Martin Simonovsky and Nikos Komodakis. 'GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders'. In: *Artificial Neural Networks and Machine Learning – ICANN 2018*. Ed. by Věra Kůrková et al. Cham: Springer International Publishing, 2018, pp. 412–422. ISBN: 978-3-030-01418-6.

[46] Pavithra Solai. *Convolutions and Backpropagations*. 2018. URL: https://medium.com/@pavisj/convolutions-and-backpropagations-46026a8f5d2c (visited on 25/08/2021).

[47] A. Sperduti and A. Starita. 'Supervised neural networks for the classification of structures'. In: *IEEE Transactions on Neural Networks* 8.3 (1997), pp. 714–735.

[48] *Stanford Large Network Dataset Collection*. URL: http://snap.stanford.edu/data/.

[49] Petar Veličković et al. *Graph Attention Networks*. 2017. arXiv: 1710.10903 [stat.ML].

[50] Daixin Wang, Peng Cui and Wenwu Zhu. 'Structural Deep Network Embedding'. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1225–1234. ISBN: 9781450342322. DOI: 10.1145/2939672.2939753. URL: https://doi-org.ezproxy.uio.no/10.1145/2939672.2939753.

[51] Yonghui Wu et al. *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. 2016. arXiv: 1609.08144 [cs.CL].

[52] Zonghan Wu et al. *A Comprehensive Survey on Graph Neural Networks*. 2019. arXiv: 1901.00596 [cs.LG].

[53] Zonghan Wu et al. *Graph WaveNet for Deep Spatial-Temporal Graph Modeling*. 2019. arXiv: 1906.00121 [cs.LG].

[54] Keyulu Xu et al. 'How Powerful are Graph Neural Networks?' In: *CoRR* abs/1810.00826 (2018). arXiv: 1810.00826. URL: http://arxiv.org/abs/1810.00826.

[55] Jiaxuan You et al. 'Graphrnn: A deep generative model for graphs'. In: *arXiv preprint arXiv:1802.08773* (2018).

[56] Bing Yu, Haoteng Yin and Zhanxing Zhu. 'Spatio-temporal Graph Convolutional Neural Network: A Deep Learning Framework for Traffic Forecasting'. In: *CoRR* abs/1709.04875 (2017). arXiv: 1709.04875. URL: http://arxiv.org/abs/1709.04875.

[57] Zihao Zhang and Stefan Zohren. 'Multi-Horizon Forecasting for Limit Order Books: Novel Deep Learning Approaches and Hardware Acceleration using Intelligent Processing Units'. In: *CoRR* abs/2105.10430 (2021). arXiv: 2105.10430. URL: https://arxiv.org/abs/2105.10430.

[58] Chenyi Zhuang and Qiang Ma. 'Dual Graph Convolutional Networks for Graph-Based Semi-Supervised Classification'. In: *Proceedings of the 2018 World Wide Web Conference*. WWW '18. Lyon, France: International World Wide Web Conferences Steering Committee, 2018, pp. 499–508. ISBN: 9781450356398. DOI: 10.1145/3178876.3186116. URL: https://doi-org.ezproxy.uio.no/10.1145/3178876.3186116.