University of Nevada, Reno

Evolving GPU-Accelerated Capsule Networks

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in Computer Science and Engineering

by

Daniel Anthony Lopez

Dr. Frederick C. Harris, Jr., Thesis Advisor

August, 2018

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

DANIEL ANTHONY LOPEZ

Entitled

Evolving Gpu-Accelerated Capsule Networks

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Frederick C. Harris, Jr., Ph.D., Advisor

Nancy Latourrette, Committee Member

Jeff Mortensen, Ph.D., Graduate School Representative

David W. Zeh, Ph.D., Dean, Graduate School

August, 2018

Abstract

Capsule Networks exploit a new, vector-based perceptron model, providing feature instantiation parameters on top of feature existence probabilities. With these vectors, simple scalar operations are elaborated to vector-matrix multiplication and multi-vector weighted reduction. Capsule Networks include convolutional layers which take the initial input and help it become a tensor. A novel data abstraction maps the individual values of this tensor to one-dimensional arrays but is conceptualized as a 2D grids of multi-dimensional elements. Moreover, loss function thresholds and architectural dimensions were arbitrarily set during the introduction of Capsule Networks. While current machine learning libraries provide abstractions for convolutional layers, a TensorFlow optimization requires structural overhead for a full Capsule Network implementation. They lack simple optimizations specifically for Capsule Network data allocation. This thesis presents a scalable GPU optimization for the training and evaluation of Capsule Networks. Furthermore, hyperparameters are adjusted with the help of a multi-objective evolutionary algorithm (MOEA) to minimize the loss function while maximizing accuracy.

Dedication

I dedicate this thesis to the people in my life who told me I probably couldn't finish this masters when I said I would; you've been a great source of motivation.

You know who you are.

Eat your heart out.

Acknowledgments

First, I would like to thank my committee members, Nancy Latourette, and Dr. Jeffrey Mortensen for their consideration and suggestions. Academically, I would also like to thank Dr. Sushil Louis for teaching me about genetic algorithms, Dr. George Bebis, who's Pattern Recognition class I took many semesters ago inspired me to focus my career path towards machine learning, and Dr. Jason Altieri for reminding to never sacrifice soul for precision. I'd also like to thank my career friend Harpreet Singh for invaluable insights support about genetic algorithms and neural networks, and simply being a friend to bounce ideas from. My mentor and advisor, Dr. Frederick C. Harris Jr. also deserves my deepest gratitude for kicking me in the rear during his brutal introductory data structures class, where I first got acquainted with him. His casual life lessons and anecdotes helped solidify my ambition for never ending self improvement, even in the little things. He also taught me how to program. And *well*.

I'd also like to thank my family members for their support. My dad instilled in me a passion for computers and clever thinking from early on in my life; it's something that I'm proud to port as part of my personality. My mother traumatized me with the discipline in looking over small details in everything I ever do. To my surprise, my sister never fails to put up with me. Above all, I could not have done this without the warmth and support of *mi* grandma who strive to always give the best to her children and grandchildren despite socio-political and inter-family calamities. Her underappreciated love for her family is what helped her become a shoulder to cry on when things got rough and the warmest welcoming hug whenever days were long.

This work was not supported by any grants, but donations through Bitcoin or Venmo are welcome. Seatbelts save lives. The white zone has always been for loading and unloading. No portion of this thesis, including its sound track, may be reproduced in any manner without consent or I won't be your friend anymore.

At the end, thanks to you, reader; if you are reading this line after all the others, you've at least read one page of my thesis.

Contents

Abstract i

Dedication ii

Acknowledgments iii

List of Tables vi

List of Figures vii

List of Algorithms ix

1 Introduction 1

2 Background and Related Work 4
- 2.1 Genetic Algorithms . . . 4
- 2.2 Neural Networks . . . 9
  - 2.2.1 Multi-layer Perceptrons . . . 9
  - 2.2.2 Convolutional Networks . . . 11
- 2.3 Capsule Networks . . . 12
  - 2.3.1 Dynamic Routing . . . 13
  - 2.3.2 Computational Walk through . . . 14
- 2.4 Genetic Acceleration of Neural Networks . . . 17
- 2.5 TensorFlow and GPU Accelerated Libraries . . . 18
- 2.6 Libraries and Frameworks . . . 19

3 Methodology 21
- 3.1 GPU Data Manipulation . . . 21
- 3.2 Algorithmic Definitions . . . 23
- 3.3 MOEA . . . 25
  - 3.3.1 Chromosome Definition . . . 25
  - 3.3.2 NSGA-II Algorithm . . . 26
- 3.4 Use Case Modeling . . . 27
  - 3.4.1 Application Use Cases . . . 29
  - 3.4.2 API Use Cases . . . 30

4 Implementation **34**
4.1 Capsule Network API and Example 34
4.1.1 Sequential Neural Networks 35
4.1.2 Parallel Capsule Network 38
4.1.3 Custom CUDA Kernels 40
4.2 NSGA-II Implementation 43

5 Results **46**
5.1 Architectural Results 46
5.1.1 Single Capsule Network Results 46
5.1.2 MOEA Results 48
5.2 Parallel Capsule Network Results 48
5.2.1 Speedup 52
5.2.2 Throughput 53

6 Conclusions and Future Work **55**
6.1 Conclusions 55
6.2 Future Work 56
6.2.1 Preexisting Tools 56
6.2.2 Extra Evolutional Parameters 56
6.2.3 Concurrency 57
6.2.4 Multi-GPU Implementations 57

Bibliography **59**

List of Tables

3.1 The hyper parameters of capsule networks are encoded in GA bit strings, where each set of bits corresponds to a point in a given range. The last four have a different step size, and they are as precise as double precision allows. 25

List of Figures

2.1 The plot of an example problem shown here delineates the Pareto front of the minimization of two functions [4]. The Utopia point at the origin, while infeasible, is the direction towards which previous solutions inch towards from the darkened region of obtained solutions. 7

2.2 Traditional neurons and capsule share similar architecture in taking weighted reductions of the previous layer. Capsules, however, have their vectors undergo an extra dimensionality transformation, specific to the relationships between lower-level and higher-level features. These **W** matrices learn relational information between features while c is dynamically evaluated per forward pass. 13

3.1 The $d_{l+1} \times d_l$ transformation matrices, shown in the left most tensor, are multiplied element wise with the d_l dimensional outputs from the lower level capsules. These outputs are stored column-wise in the middle tensor, but are duplicated by column-wise for each higher-level capsule. The Hadamard product of these tensor produces d_{l+1} sized vector inputs to the higher level capsules in the right most tensor. . . 22

3.2 The output of the convolutional layer is reshaped from individual scalars to vectors, depth-wise. For this to happen, the number of filters in the convolutional layer must be a factor of the dimension of the vectors. This specific vector mapping is important to preserve; during back propagation, the list of vectors is converted back to a set of scalars. 23

3.3 The main loop of the NSGA sorts P_t and Q_t by Paredo fronts and put as many as it can into P_{t+1}. The odd front out is sorted by a distance-based crowding operator [4]. 26

3.4 The terminal interface provides options for running a single Capsule Network or the entire genetic algorithm. It also allows configuration settings for the user. 30

3.5 The API provides high level abstractions for simple construction and training of Capsule Networks. 31

4.1 The sequential implementation of Capsule Networks and other neural network primitives, following OOP standards 36

4.2 The parallel implementation of CUCapsuleNetworks and other GPU accelerated neural network primitives, following Struct-Of-Array Conventions. The bottom object is not an entity, but a list of unique kernels (wrappers). They are given the `CUUnifiedBlob` instances as reference parameters. 39

4.3 The NSGA-II implementation is simple; aggregating individuals into populations, and shorthanding populations into Paredo Fronts. A Paredo Front is abstracted as a pointer array for easily sorting for individual updating. The CapsNetDAO helps speedup Individual evaluation by creating SQL queries and checking for duplicate networks to prevent wasteful duplicate evaluation. 44

4.4 This PostgreSQL entity uses the individual bitstring as the primary key and contains decoded hyperparameters and Capsule Network metrics as other row attributes 45

5.1 The Accuracy for the NSGA-II configuration outperforms the original architecture, despite both having a small dip just before 200 iterations, due to local minima in $\mathbf{W}$ space. 47

5.2 The Loss of the NSGA-II configuration starts out at a smaller pace, but after finding a local minima, starts to rise again at 200 iterations. 47

5.3 The minimum, average, and maximum accuracies of the population of the MOEA after 100 epochs (top) and after 300 epochs (bottom). . . 49

5.4 The minimum, average, and maximum loss values of the population of the MOEA after 100 epochs (top) and after 300 epochs (bottom). . . 50

5.5 Forward propagation takes less time than back propagation in the sequential version of these methods since there is no data movement (despite the inner loop found in Dynamic Routing [25].) These methods are not multi-threaded and are compiled without compiler optimization flags. 51

5.6 Back propagation is clearly faster than forward propagation due to lack of communication overhead in data movement. Note the scale as compared to Figure 5.5. 52

5.7 Speedup of these methods start to slow down between 10-15 tensor channels (360-540 lower level capsules) as these methods increase due to Amdahl's law. Note that back propagation, which only communicates resulting $\mathbf{v}_j$ errors back to the host, a constant $k \times d_{l+1} = 160$ values, has higher speedup than forward propagation, which requires the movement of a 28×28 sized image from the MNIST dataset. [21]. 53

5.8 The throughput is measured by a factor the number of floating points required to compute (not the operations) at the variable layer divided by the amount of time taken to complete the meta-operation. These floating points are the ones for the interim layer only. 54

List of Algorithms

3.1 Forward Propagation 24
3.2 Back Propagation 25
3.3 Fast Non-Dominating Sort 28
3.4 Crowding-Distance Assignment 29

Chapter 1

Introduction

Given a low resolution image of a sloppily written digit, humans would be able to recognize what type of digit is written within milliseconds. However, such an example transforms from comically trivial to dauntingly difficult when prompted to a computer. This is due to the advanced biological infrastructure which allows us to assign abstract labels, such as digit classes, to complicated inputs, such as images. Inspired by these neurological processes found in nature, neural networks are a supervised machine learning technique which learn high-level relationships between sets of inputs and outputs presented. However, these relationships are non-linear and are thus rendered as difficult quantitative processes [5, 12, 18]. Nevertheless, these may be found through the mathematical advent of discriminatory learning [5].

Different optimizations have been introduced such as recurrent neural networks, long-short term memory networks, and convolutional neural networks, which introduce ideas such as cyclical inputs, in-place memory, and shared weights, respectively, in an effort to increase the robustness in estimations [18, 23]. The sudden rise in popularity in recent years has been brought has been propelled by advancements in computational hardware. With more robust computation power, convolutional neural networks have become synonymous with deep learning [23]. A recent novel optimization challenges the current perceptron model to perform operations on vectors rather than scalars. In this model, groups of neurons are called capsules, inputting and outputting mathematical vectors [25]. These vectors encode the probability of a higher level feature existing in their length, and estimated pose parameters as its orienta-

tion. Moreover, relationships between features detected in lower level and higher level capsules must be qualified [13, 25]. Nevertheless, simple iterative methods such as dynamic routing between capsule layers exist, enabling the network to become more robust to subtle variations in the input data [25].

When it comes to hardware, Graphical Processing Units (GPUs) have become more popular to accelerate the training and evaluation steps in these networks [2, 19, 26]. Data and task parallelism approaches have been introduced to scale these networks to multiple GPUs in accelerations for those. To take advantage of these, heterogeneous machine learning libraries such as TensorFlow or Keras provide a framework for fast experimentation of these models on distributed systems [1]. Nevertheless, overhead is introduced with inefficient memory transformations and data allocations. This thesis eliminates this overhead with efficient memory sequencing and optimized techniques, such as shared memory reduction, in hopes to provide a new technique specific to capsule networks. For example, convolutional neural networks may be implemented in GPUs using altered fully-connected layer tools. However, a later work optimized them, transforming two dimensional filters into redundant one dimensional weight arrays, thus introducing a new set of deep learning primitives, now found in TensorFlow [1, 2].

Moreover, certain hyper parameters in these networks are arbitrarily set and could be fine tuned. Genetic/Evolutionary Algorithms (GAs) could be the solution for this [8].

GAs emulate another aspect of nature, evolution, by allowing a population of potential solutions to evolve using operators analogous to natural selection and cross breeding [8]. Moreover, these solutions are binary string chromosomes that encode higher level parameters. These operators are proportional based selection, crossover and mutation, and effectively allow genes of information to proliferate throughout successive populations. Individuals are rated after their decoding is run through objective fitness functions, and their performance directly influences their prevalence in the GA. Multi-objective evolutionary algorithms (MOEAs) take this one step further

by having multiple functions help determine the fitness of an individual [4].

The capsule network hyper parameters are encoded as chromosomes, evolved by GAs. Thus, an individual represents a hyper-parameter configuration from which a capsule network is constructed. The accuracy in digit prediction and the margin loss of the network become two objective functions in the GA. Each capsule network is constructed in a pure CUDA implementation, with no external libraries such as cuDNN. Novel low level memory allocation and sequencing will enable this capsule network implementation to outperform a potential TensorFlow equivalent.

The rest of this thesis is structured as follows: a little background into capsule networks and their architectures, and MOEAs and their effectiveness over canonical GAs is presented in Chapter 2. Chapter 3 goes over the methodology of the network data abstractions and the cluster setup, as well as MOEA parameters. Chapter 4 introduces the high-level software class diagrams and requirements for implementation. Chapter 5 provides parallel based results such as speed up, and the fine tuned parameters for the GA. Finally, the paper wraps up in Chapter 6 with a discussion of the future work and the conclusion.

Chapter 2

Background and Related Work

This work is largely built off of the architecture and work on Capsule Networks [25]. Due to the size of data processing required, an efficient GPU implementation thereof requires careful data allocation and sequencing. Moreover, there exist hyper parameters that may be fine tuned with the help of genetic algorithms which exploit diversity from multiple solutions and literally cross breed higher performing solutions. Because neural networks deal with accuracy of its classification, as well as overall loss (how "badly" was something misclassified), the genetic algorithm must aim to optimize both of these objective functions.

2.1 Genetic Algorithms

In nature, the process of evolution exploits strands of DNA as a method to encode genes. These genes then dictate the existence of attributes in an individual which have a direct impact on the survival capabilities of the individual.

Genetic Algorithms (GAs), or evolutionary algorithms (EAs) are analogous to this schema by encoding potential solutions and then evaluating the performance of the solution (or fitness of the individual) through an objective fitness function [8]. The individuals in these algorithms are inputs to a problem, highly abstracted and encoded as bit strings. As part of a population of other solutions, each individual has their "fitness" measured through an objective fitness function, which decodes the bit strings back to the input format of the problem at hand. To produce new population

of solutions, they undergo three different operations to produce a child operation.

These operations are: selection, a method of systematically choosing individuals based on their fitness function performance; crossover, swapping bits between chromosomes in an effort to "share" information; and mutation, randomly flipping bits in this chromosome. The latter two operators are performed with some probability in the algorithm; crossover is typically done with 20%-80% probability, whereas mutation is performed with .001%-.1% probability [8]. With the proliferation of GAs, varieties of each operator have been developed and created. For example, elitist selection schemes favor higher performing solutions in the previous population when creating the new population, or when selecting individuals for crossover.

Given large populations and high probabilities of crossover and mutation, genetic algorithms are very adept at optimizing solutions in a very large space. Moreover, the high-level abstraction of these inputs as bit strings enable certain "genes" to be shared among other members in the population through crossover. This enables the GA to find a balance between exploiting performance bearing genes and exploring new possibilities which introduces randomness into the searching space. However, the proliferation of GAs as solution finders and optimizers does not directly imply their use as function optimizers [14]. Depending on the problem, highly performing solutions may be in an opposite direction from where "slopes" of better performance in the solution space. Nevertheless, GAs are shown to still be fundamentally great multi-peak optimization finders and routines when the size of the populations and the number of trials are repeated to ensure flukes or coincidental pitfalls do not hinder the performance of the genetic algorithm [8, 17].

NSGA-II

Some of these fitness functions may produce more than one value, if more than one attribute of the fitness of the individual is slated to being optimized. For these, a specialized type of GA, a multi-objective evolutionary algorithm (MOEA), is used which attempts to find the balance between multiple solution spaces [4, 8]. These

algorithms tend to have slightly more overhead, since they have to consider potentially countering objective solutions. NSGA-II is such an algorithm which also lower amount of computational complexity. NSGA-II addresses issues from NSGA-I, its predecessor: faster nondominated sorting (a subphase of the algorithm) and elitism [4].

Pareto Fronts

In single objective fitness function based GAs, the validity and the evolution of the individuals may be detailed by charting the best, average and/or worse individuals throughout each generation. These graphs will logarithmically, yet stochastically approach higher possible values until they eventually plateau. This is due to the highly varied "gene-swapping" caused from cross-over and selection operators. Since selection based operations allocate higher chances of selection to higher performing individuals, the higher-performing "genes" are more likely to get duplicated throughout the population [8, 10].

In MOEAs, however, validation is graphed by the placement of Pareto fronts [8]. These fronts represent the set of individuals that best attempt to maximize multiple objective functions while minimizing compromises between them, as seen in Figure 2.1. In each iteration of the genetic algorithm, these fronts inch towards the utopia point, the best possible (yet almost always unattainable) combined output from both objective functions. In NSGA-II, these fronts are explicitly found as solutions for which there does not exist another solution with a higher value for both values. In other words, to say that an individual is from Pareto Front, $\mathcal{F}_i$, is to say this individual *dominates* all other solutions from $\mathcal{F}_{i+1}$. The fronts are used for the operations in NSGA-II [4].

Operations

First, to create the population of individuals used for an iteration, t, of the GA, the current population and one created from the last iteration are unioned, $R_t = P_t \cup Q_t$. From here, the individuals are ranked by a nondominating sort; a stable sorting

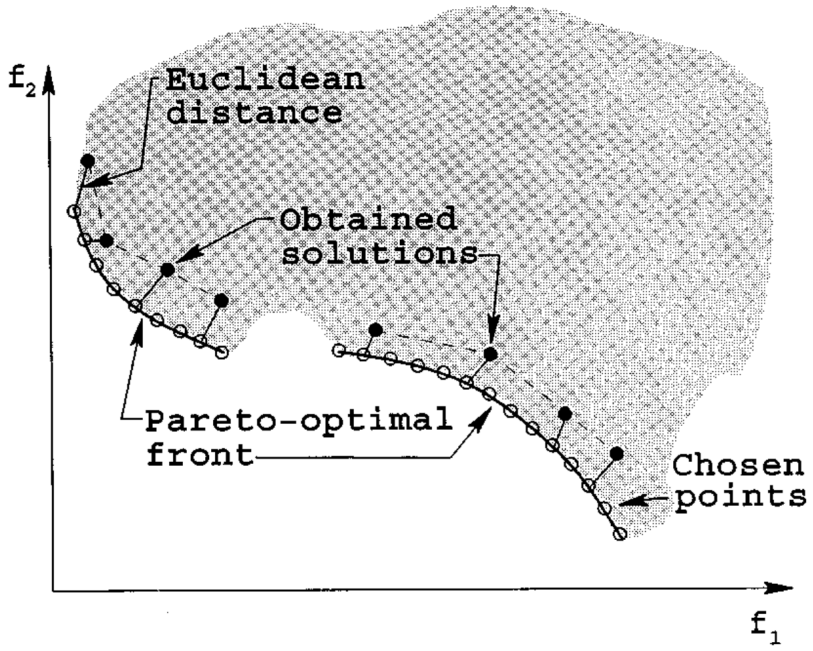

Figure 2.1: The plot of an example problem shown here delineates the Pareto front of the minimization of two functions [4]. The Utopia point at the origin, while infeasible, is the direction towards which previous solutions inch towards from the darkened region of obtained solutions.

algorithm based on whether or not an individual dominates another. A side effect of this sorting is that the individuals are now ranked into Pareto fronts.

A new population, P_{t+1}, is generated by incrementally adding these Pareto fronts, while the size of the population would exceed N by adding a front, $|P_{t+1}| + |\mathcal{F}_i| \leq N$.

For the last Pareto front that does not evenly fit, the individuals are assigned a "crowding-distance" parameter which encodes the individuals distance from being an extreme solutions. An extreme solution is considered as an individual which has optimized one objective function, but has not considered the others, and is thus on the ends of a Pareto Front. The individuals here are sorted by this distance parameter, which favors those towards the center of the Pareto front. Only those needed to complete the population up to N are used, $P_{t+1} = P_{t+1} \cup \mathcal{F}_i[1 : (N - |P_{t+1}|)]$.

Crowding Distance Parameter

The parameter needs to help ensure individuals with similar consideration for all objective functions being considered have higher values than those that do not. The process by which this parameter is updated is thus done per objective function in the equation. For each objective, m, the individuals are sorted by their respective output for this function. The first and last individual are then given a distance parameter of ∞, acting as an upper bound. Every other individual then has its value incremented by a factor of the difference in output of its surrounding individuals divided by the range of the function. More explicitly, if $f_m(\mathcal{F})$ represents the output of this individual through objective function m, then the distance parameter for the intermediate solutions is incremented as

$$\mathcal{F}[i]_{distance} = \mathcal{F}[i]_{distance} + \frac{f_m(\mathcal{F}[i-1]) - f_m(\mathcal{F}[i+1])}{f_m^{\max} - f_m^{\min}} \tag{2.1}$$

In this paper, multiple neural networks are generated with distinct hyper parameters. Each set of hyper parameters are encoded into binary strings, found by an NSGA-II architecture MOEA. The objective functions used will be the accuracy and the overall loss of a trained network, after a fixed amount of iterations. These hyper

parameters help to fine tune the overall performance of the neural network by changing dimensionalities and other arbitrary parameters set. For example, when evolving convolutional networks, it had been assumed that adding more layers would automatically increase the overall accuracy of the network. In fact, however, there exists a threshold past which adding more layers does not necessarily improve performance, but rather, hinders it.

2.2 Neural Networks

In another biological analogue, neural networks are constructed as a model of synapses found in human brains [18]. As a machine learning model, these may be used for pattern classification using discriminatory learning, the advent being able to detect features in an input that differentiate elements from different classes during training [18, 24]. Using supervised training, images are assigned a label for the class they belong to, and neural networks will find high dimensional associations between these [18].

All neural networked based variation works off of a layered structure. Each layer consists of nodes which receive weighted inputs from nodes of the previous input; the first layer being the input image and the last output layer being the estimated label for that input image, evaluated through forward propagation. During back propagation, these estimations are "corrected" through the help of a loss function. The gradient of the loss functions (as well as the partial derivatives of other non-linear activation functions throughout the insides of the network) help push internal weights for node activations [12].

2.2.1 Multi-layer Perceptrons

Nodes in multi-layer perceptrons are as simple as possible, a weighted reduction of the single scalar activation outputs from the previous nodes undergoes a non-linear activation function, such as a sigmoid function [12, 18]. The non-linearity of these functions normalizes the input from $(-\infty, \infty)$ down to $(-1, 1)$, to prevent large values from dominating, and thus misrepresenting, the influence of layer from one layer to

the next. Conventionally, a sigmoid activation is used as the non-linear function:

$$\sigma(x) = \frac{1}{1 + \exp -x} \tag{2.2}$$

At the end of forward propagation, where an input vector is passed from each layer to the last layer, the error of the network at that point is the euclidean distance between the output vector and the true desired output vector. This desired output vector is all zero, except for the element with the index corresponding to the class the image belongs to.

For example, if using a 28×28 pixel image of a handwritten digit from the MNIST data set, the image will be considered a single 784 dimensional vector [21]. This vector is then dotted with a weight vector, where each element corresponds the weight between the higher level node it belongs to and every lower level input node. Finally, this scalar product is undergoes the non-linear activation function to become the output for that weight. At the last layer, there is one node for every class, and strength of the output at each node should indicate the strength with which the network believe this input belongs with this class [23].

During back propagation, an error vector is calculated, and in the reverse of back propagation, this error is propagated from layer to layer. In the process, at each layer, since the input originally goes through a non-linear function, the error must go through the derivative of the same non linear function.

In our example, the error of the last layer is defined by

$$\delta x_i = (\sigma(x_i)(1 - \sigma(x_i))) * (x_i - y_i). \tag{2.3}$$

However, in any other layer l, the error is the weighted accumulation of the error from higher level, gone through the derivative of the non-linear activation function:

$$\delta x_i^l = \sum_j \sigma(\delta x_j^{l+1})(1 - \sigma(\delta x_j^{l+1})) \tag{2.4}$$

Moreover, although the error gradient indicates the direction towards which the outputs are inched, the different in the weight between the node in x and the node

in y, w_{ij}^{l} is computed as the product of the original input and the error, $x^{l}\delta x^{l+1}$. Overtime, these changes are accumulated, but not applied onto the actual weights themselves for a while for mini-batch processing.

2.2.2 Convolutional Networks

The artificial intelligence community shifted its attention to image processing with 2D inputs, and Convolutional Networks, based largely on other biological features (such as visual cortex processing) were more popular [19, 20, 23]. This was especially true since the computational power required to process large batches of data became more readily available with hardware advances [18]. The novelty and effectiveness of the convolutional and pooling layers in these networks were enough to make them a highly attractive commodity for large businesses in the tech industry. Convolutional Networks has become synonymous with Deep learning in recent years thanks to the "AlexNet" and "GoogLeNet" architectures that came shortly thereafter [19, 29].

Convolutional Networks make use of two main layers, along with a highly simplified activation function (which happens to be mostly linear this time): Convolutional Layers, which introduce depth into cubes of data, and Pooling Layers, which significantly lower computational effort by consequently losing information. The convolutional layer is comprised of a user-specified number of "filters", the same depth,[1] but potentially smaller height and width of the original input image. The filter is then "dotted" with a portion of the input, to produce a scalar output specific to that filter, for that image. Here, the filter values may be considered weights which are shared among multiple input nodes in a very specific fashion. It is this shared weight property of convolutional layers which enable invariance: the novel principle allowing a feature to be detected *anywhere* in the image [21, 25].

The output of a convolutional layer is an array of feature maps, one for every filter in the convolutional array, creating a three dimensional cube of data, which

[1] input may either be a potentially 1 (greyscale) or 3 (RGB) channel image, or the output of another layer

significantly increases the amount of processing to be done by other layers down the pipeline. To help with this problem, pooling layers reduce the height and width of the images by, in a similar striding method, will look only a single value in this window, and output that. This single value is either the maximum value, or the average of all the values from that location in the image. In the case of max-pooling, this layer effectively operates equivalently to a convolutional layer, with a large window and small striding step (resulting in a feature map with a significantly lower height or width) consisting of filters with only a 1 in the filter location for the highest value in *only that* window for that specific moment, per channel.

Finally, after each convolutional layer, each value passes through its activation function, known in its most basic form as a Rectified Linear Unit (ReLU), $x^+ = max(0, x)$. This ReLU activation truncates all negative values to zero, in an effort to quick calculation, although several other benefits come from it, and its variations, such as Leaky ReLU: hidden nodes that are activated are sparser for a newly initialized network, and it helps the vanishing gradient problem.[2]

2.3 Capsule Networks

Despite the proliferation and versatility of convolutional networks, a new architecture of networks was introduced by [25] in which groups of nodes work together in a "capsule", effectively dealing with vectors of information for inputs and outputs rather than scalars in all other classical methods. Now, the neuron model has been expanded from single scalars to vectors, where input and output vector cardinalities need not be the same, as seen in Figure 2.2. In this model, the values of the output vector correspond to specific instantiation parameters of an object detected in an input, and the length of the vector encodes the probability of the feature (for that capsule) being present in the input.

Analogously to a conventional neural network, the capsule outputs from one layer

[2]During back propagation (and potentially during simulated annealing techniques), the derivative of a non-linear activation gradient makes small error gradient values propagate as even smaller values, until eventually vanishing before it has been able to make in impact on lower layers [7].

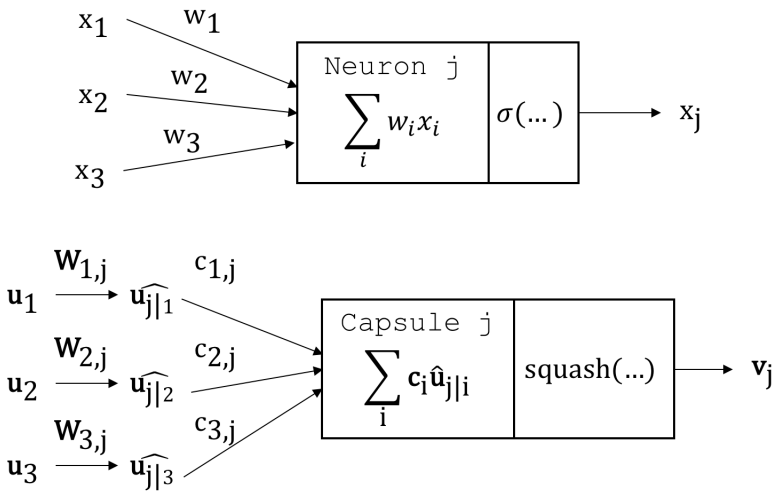

Figure 2.2: Traditional neurons and capsule share similar architecture in taking weighted reductions of the previous layer. Capsules, however, have their vectors undergo an extra dimensionality transformation, specific to the relationships between lower-level and higher-level features. These $\mathbf{W}$ matrices learn relational information between features while c is dynamically evaluated per forward pass.

(after undergoing some dimensionality transformation via transformation matrices) to the next are compiled via some weighted sum-reduction algorithm. The weights for this algorithm are computing by a routing-by-agreement algorithm which, similarly to K-means [16], grants a higher weight to vectors whose outputs tend to cluster together, thus exploiting the rareness of "agreeing" vectors in higher dimensions found in discriminatory learning [5]. Although the authors of [25] present this straightforward method, they also stress that there are many different ways to produce a similar output.

2.3.1 Dynamic Routing

The presented CapsNet architecture in [25] features a convolutional layer, followed by two capsule layers, the latter of which contains a capsule for each feature class defined. The dynamic routing algorithm occurs between two capsule layers in a network, where the initial "output" vectors of the first capsule layer are composed of the same pixel value from many filter outputs from the previous convolutional layer.

In machine learning libraries, such as TensorFlow, individual tasks are distributively organized to accommodate many different devices and easy scaling from a single machine to a distributed system [1]. However, when implementing Capsule Networks, memory resources are wasted when the tensor representing the cube output from the convolutional layer is transformed via these tasks to a set of 8D vectors. Moreover, during reconstruction error generation, although tasks are modeled alongside their dependencies in a graph-based representation in an effort hide latency, redundant or unnecessary tasks may accidentally execute. This GPU acceleration method does not compute the reconstruction error.

2.3.2 Computational Walk through

The CapsNet Architecture is defined as a convolutional layer (PrimaryCaps), followed by a capsule layer, which reinterprets the output of the convolutional layer as its own output, followed by a smaller capsule layer (DigitCaps). The principle novel computation in capsule networks lies in the operations between capsule layers during the forward propagation stage. Here, lower-level, lower-dimensional capsules undergo dimensional transformation and a dynamically weighted reduction to become the output of a higher-level, higher-dimensional capsule network. This transformation is analogous to multiplying individual scalars by weights in feed-forward, fully-connected layers.

The lower-dimensional capsules has "outputs" by the feature map output of a convolutional layer with ReLU activation. In PrimaryCaps, the number of filters must be divisible by the lower dimension to reshape the feature map outputs as vector maps. For referential integrity, the lower dimension, d_l, will be 8, and the higher dimension, d_{l+1} will be 16. Therefore, depth wise, d_l sized increments may be considered a vector map, and the number of lower level capsules is d_l times the number of vector channels. The number of vector maps (and consequently, the number of lower-level capsules) are varied in this paper to study the speed up effectiveness. Furthermore, these methods will use the MNIST data set of hand written digits of 28×28 pixels.

Forward Propagation

First, the lower level capsules, j, produce an output vector, $\hat{\mathbf{u}}_{j|i}$, for each higher level capsule, i, once for each possible output class, estimating the parameters of *their* output vector, $\mathbf{v}_i$, which is transformed to the dimensional space of the higher layer by an evolved transformation matrix, $\mathbf{W}$, such that $\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\mathbf{u}_i$. Each higher level capsule then computes a dynamic weighted sum of these vectors as their output, $\mathbf{v}_j$.

For a vector to calculate its output, the weighted sum result, $\mathbf{s}_j$, undergoes a vector squishing activation function,

$$squash(\mathbf{s}_j) = \frac{\|\mathbf{s}_j\|^2}{1 + \|\mathbf{s}_j\|^2} \frac{\mathbf{s}_j}{\|\mathbf{s}_j\|} \tag{2.5}$$

This is analogous to the sigmoid activation functions usually applied onto the weighted sums in traditional capsule networks.

Dynamic Routing

The dynamic weights, $\mathbf{c_i}$, are updated by computing the log prior probabilities, $\mathbf{b_i}$, which are iteratively updated. During each iteration, the probabilities, $\mathbf{b_i}$ are incremented by the scalar product of the activated weighted sum,

$$\mathbf{v}_j = squash(\mathbf{s}_j), \mathbf{s}_j = \sum_i c_{ij}\hat{\mathbf{u}}_{j\|i} \tag{2.6}$$

and the vector in question, $\hat{\mathbf{u}}_{j\|i}$.

The final capsule vectors outputted encode the probability of the existence of that feature in the length of the vector, while encoding the instantiation parameters of the pose of that feature in the orientation of the vector. The orientation parameters are heavily determined by the transformation matrix, which are optimized using μ momentum, instead of the Adam optimizer. The network will classify an image as being part of class i from k classes with the maximum length, $\|\mathbf{v_i}\|$.

Back Propagation

The loss function for Capsule Networks takes into account the length of the vector, not unlike the magnitude of the scalar value found in normal neural networks. The

loss function may be described as

$$L_k = T_k \max(0, m^+ - \|\mathbf{v}_k\|)^2 + \lambda(1 - T_k)\max(0, \|\mathbf{v}_k\| - m^-)^2 \tag{2.7}$$

where $m^+ = 0.9$, $m^- = 0.1$, and $\lambda = 0.5$. Its corresponding derivative taken with respect to this length is a piece-wise function since it must consider the max functions

$$\frac{d}{d\|\mathbf{v}_k\|}[L_k] = \begin{cases} -2T_k(m^+ - \|\mathbf{v}_k\|) & \|\mathbf{v}_k\| < m^+, \|\mathbf{v}_k\| \leq m^- \\ 2\lambda(T_k - 1)(m^- - \|\mathbf{v}_k\|) & \|\mathbf{v}_k\| \geq m^+, \|\mathbf{v}_k\| > m^- \\ 2(\lambda(T_k - 1)(m^- - \|\mathbf{v}_k\|) + T_k(\|\mathbf{v}_k\| - m^+)) & \|\mathbf{v}_k\| < m^+, \|\mathbf{v}_k\| > m^- \end{cases} \tag{2.8}$$

In back propagation, a corresponding error value is computed as a factor of the length of the vectors. This error value is the combination of the gradient of the loss function, multiplied by the derivative of the vector squashing activation function, effectively representing the error gradient towards which the free $\mathbf{W}$ values scattered throughout the network collectively inch towards. The derivative of the activation function is simply

$$\frac{\partial}{\partial\|\mathbf{s}_j\|}[squash] = \frac{2\|\mathbf{s}_j\|}{(\|\mathbf{s}_j\|^2 + 1)^2}\frac{\mathbf{s}_j}{\|\mathbf{s}_j\|} \tag{2.9}$$

These are weighted by their respective c values to produce the error gradients, $\delta\mathbf{u}_{j|i}$, for all i output capsule vectors.

The error gradients are then transformed into the dimension of the lower level capsules, by being multiplied by the transpose of the original transformation matrix, $\mathbf{W}_{ij}^T$. Before then, however, a matrix product of these error gradients, and the original inputs to the network become part of the $\Delta\mathbf{W}_{ij}$. The output of the lower level capsules, however, are in truth, the rearranging of the output of the earlier convolutional layer. Back propagation occurs in this layer as normal.

Mini-Batch vs. Sequential Updating

Given the time it takes to train a network, and the potential bias the ordering of the training examples gives to the network, different techniques were created in order to

speed up processing and reduce potential bias and equalize the change all training examples provide. The latter is important to greatly increase the chances the network will converge to a more global optima. Mini-batching is one such technique.

Although there have been advances in high performance computer for large amounts of data, accumulating and periodically applying weight changes waiting to apply the changes in these weights to the weights themselves is still a widely adopted practice, as it will minimize bias on the network from external sources.

In mini-batching, a batch of input images are provided to the network, where forward, and subsequently, back propagation are computed in parallel to one another. Afterwards, the resulting $\Delta\mathbf{W}$'s are reduced from all these "layers" to provide one main error, by which the network is updated. This paradigm is used in machine learning frameworks such as TensorFlow.

All matrices and vectors used in these computations are allocated as 1D arrays and indexed very precariously in the proposed method. Traditional mini-batching compilations of images would include higher complexities in these indexing. Moreover, the reduction of these $\Delta\mathbf{W}$'s from multiple devices would increase the communication needed between the host and all potential devices, thereby reducing potential scaling benefits. Therefore, this method does not use this technique, but rather computes forward and back propagation for each image sequentially, accumulating the error in $\mathbf{W}$ and then applying it to $\mathbf{W}$ at the end.

2.4 Genetic Acceleration of Neural Networks

This project is not the first to use a genetic algorithm to update structural parameters of a neural networks. Due to the exploitative abilities of a genetic algorithm, architectural and non-intuitive, programmatic insights about neural networks have been discovered [9, 15]. These niche hyper-parameter discoveries validate using an MOEA for evolving a novel type of neural networks[9].

In other neural network procedures, genetic algorithms have been shown to help optimize back propagation [15]. Learning-rate-optimizing genetic back propagation

(LOG-BP) helped evolve the neural network weights and the solver. LOG-BP helped choose scheduling learning rates, which were a problem for increasingly elaborate neural network architectures despite GPU acceleration, such as GoogLeNet [29]. Helping evolve the learning rate throughout training would help with other known side effects such as the vanishing gradient problem found in stochastic gradient [7, 22, 24].

2.5 TensorFlow and GPU Accelerated Libraries

Given the embarrassingly parallel nature of some of the filter operations, operations on the convolutional layer may be facilitated with previous abstractions of multidimensional data. Such abstractions allow input and output to be distributed in nature for multiple GPUs [2, 3, 19]. Moreover, although other linear algebra optimizations exist, they are used for every individual step. A specialized library of operations with access to the memory allocation itself include bigger-picture optimizations.

TensorFlow provides an API to a model-loss centric framework; a model is defined as a directed acyclic graph of tasks that eventually lead to a loss function which is minimized through a solver [1]. This generalization allows it to attempt to complete tasks that are independent of one another in parallel, as well as distributing a task across many nodes and potential multi-core devices. Generalized as it may be, this has serious pitfalls, as resources may be wasted on needless tasks, simply because a later dependent task decides not to used based on other input. The framework seizes the control flow and may even waste computational power on values that are thrown away. However, the solvers in this framework still enable a Tensorflow implementation to reach around 90% accuracy within hours sequentially, and within minutes with GPU computation. Therefore, TensorFlow implementations of capsule networks have room for efficiency and speed up optimizations.

Much like how convolutional layers can be abstracted at a high-level to built-in CUDA functions, capsule layers can be reffered to as a collection of low-level functions that handle data and its manipulation [1, 2, 3]. Similar the n-dimensional tensor data structure required in TensorFlow, data is grid-wise allocated, each cell potentially

containing multi-dimensional elements. To reduce data structure allocation, one-dimensional arrays are maintained for each individual data transformation mentioned in the procedures above. The work presented in this thesis hopes to lead to the development of low level CUDA kernels, specific for capsule layer abstractions.

2.6 Libraries and Frameworks

Different parts of the methods presented depend on multiple libraries and frameworks. These include integrated development environments (IDEs), services for database networking, and specialized hardware with multiprocessing capabilities.

The genetic algorithms and the capsule networks they produce will all be implemented and handled in C++/CUDA. C++ is an object oriented language, juxtaposing the ability of low level data management and high level class encapsulations for full control of the data being shifted while retaining an understandable, abstract view. Although not highly revered in its networking capabilities, it provides great speed for CPU (host) sided code. Nevertheless, its age and prevalence in programming have garnered a vast standard templated library (STL), maintained on the open source community [28].

For PostgreSQL database accessing, the standard C++ library PQXX is used. For linear algebra calculations and data structures, Armadillo is used [27]. Finally, for GPU side processing and access, NVIDIA's CUDA library and language is used [26].

PostgreSQL is a simple relational structured query language (SQL) database. Although it has capabilities for heavy load and quick fetching, its use in this project is for caching configurations for previous networks in a genetic algorithm; a sort of duplicate checking for optimization. This was chosen for its ease of setup, formidable network configurations, and its accompanying multitude APIs for a variety of languages.

For an IDE, CLion by Jetbrains was chosen for its intuitive design and navigation. Moreover, CLion make extensive use of CMake for its library and package management for deploying programs, necessary for all aforementioned external libraries used.

The use of CMake enables this IDE to provide error checking, completion, import optimization and automation, and version control software integration, rendering it invaluable for larger projects such as this one.

Chapter 3

Methodology

This project has three parts associated to it: a CPU-based MOEA, a GPU enabled Capsule network, and, for convenience, a database for chromosome result caching. The latter is done for help speeding up the execution of the genetic algorithm. For the execution of the Capsule networks themselves, the data sequencing may be abstracted as a two-dimensional grid of elements. These high level elements may even be matrices themselves, and such, they must be indexed with precariously.

The network is trained with several forward and back propagation passes for images from a training data set with periodic weight updates. To go through all data points is an epoch, and several epochs are performed in an effort to minimize the overall network loss, and conversely, maximize accuracy. Network accuracy evaluation is done after training, where a different testing data set is used to eliminate biased estimations.

3.1 GPU Data Manipulation

Due to the architecture of CUDA, software design patterns are favorable if they are built around object-oriented structures (or structs, for short) of arrays rather than arrays structs [26]. All data structures are allocated using Unified memory, where data movement is optimized by the device scheduler [26]. Since everything is allocated with 1D arrays, memory management and organizing is highly significant, and this method presents one way to arrange the data.

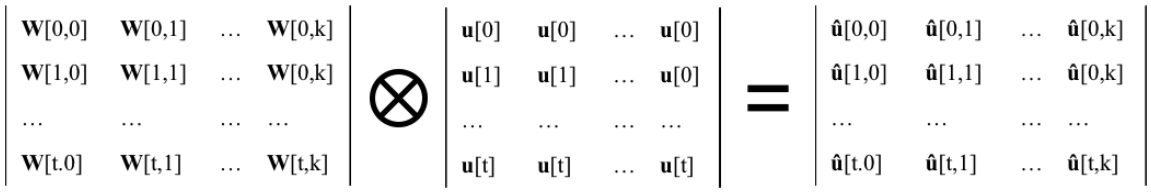

Figure 3.1: The $d_{l+1} \times d_l$ transformation matrices, shown in the left most tensor, are multiplied element wise with the d_l dimensional outputs from the lower level capsules. These outputs are stored column-wise in the middle tensor, but are duplicated by column-wise for each higher-level capsule. The Hadamard product of these tensor produces d_{l+1} sized vector inputs to the higher level capsules in the right most tensor.

Data may be thought of as a $k \times t$ grid of potentially multi-dimensional elements in row-major ordering, where an element i, j corresponds to class i and lower level capsule j. An example of the data layout may be found in Fig. 3.1, where $\hat{\mathbf{u}}$ is being created for each higher-level capsule column-wise, from each lower-level capsule row-wise. The lower level capsule outputs, $\mathbf{u}_{i,j}$, are represented in the middle tensor, and are duplicated along each column. Although this is potential memory storage waste during forward propagation, the extra storage will be used to save appropriate $\delta\mathbf{u}_{i,j}$ during back propagation.

For the capsule layer interface operations, there are a total of two 1D element grids, $\mathbf{b}$ and $\mathbf{c}$, three vector-element grids, $\mathbf{u}$, $\hat{\mathbf{u}}$, and $\mathbf{v}$, ($\mathbf{v}$ has a height of 1 but otherwise shares the dimensionality of $\hat{\mathbf{u}}$) and three matrix-element grids, $\mathbf{W}$, $\Delta\mathbf{W}$, and $\mathbf{W}_{\text{velocity}}$, the latter two of which are used for updating.

The preceding convolutional layer, however, requires simpler, sequentially indexed (channel, then depth if applicable, then height, then width) of 3 and 4 dimensional arrays. These are required for the input, $\mathbf{x}$, the output, $\hat{\mathbf{x}}$, and the filters of the arrays. Much like the $\mathbf{W}$ in the capsule layer, the filters have two other equally sized companion arrays, to hold the errors, and the velocities required in momentum updating.

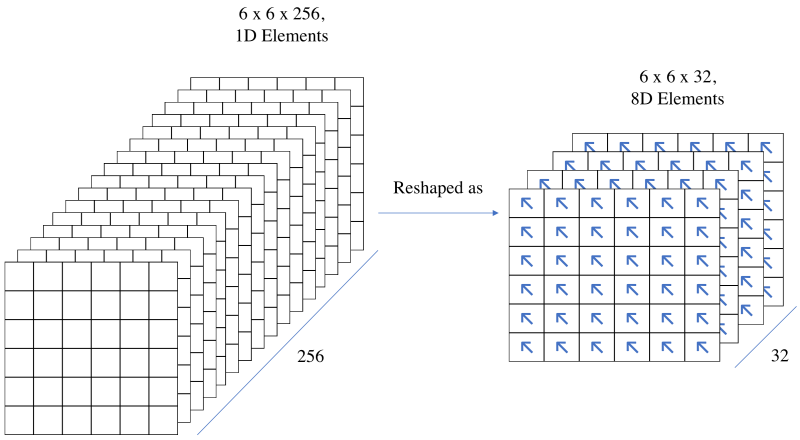

Figure 3.2: The output of the convolutional layer is reshaped from individual scalars to vectors, depth-wise. For this to happen, the number of filters in the convolutional layer must be a factor of the dimension of the vectors. This specific vector mapping is important to preserve; during back propagation, the list of vectors is converted back to a set of scalars.

3.2 Algorithmic Definitions

In Algorithm 3.1, forward propagation is given the image as a vector, $\mathbf{x}$, and requires the use of the dynamic routing procedure defined in [25].

First, the output of the convolutional layer must be transformed into a set of vectors [25]. The output is three dimensional, where each filter detected feature map corresponds to a depth-wise layer [5, 11]. From one dimensional scalars to d_l dimensional vectors, this tensor reshaping generates the interim, pre-activation vectors of the lower capsule layer, as seen on Figure 3.2. The original convolutional output shape has a memory allocation dependent on the library used; some optimizations include redundant ordering of filter layers to produce row-wise ordering output tensors [3]. Assuming a straightforward, width, height, then depth ordering, output from the convolutional layer, $\mathbf{x}$, the vectors are rearranged in a list to become $\mathbf{u}$, taken care of by the *Rearrange* method.

These vectors are then undergo the vector squash activation function, the nonlinear function which facilitates discriminatory learning by scaling vector lengths close to zero and long vectors closer to one. Afterwards, as is illustrated the middle tensor

shown in Figure 3.1, these vectors are duplicated along the "columns", representing each of the DigitCaps classes. Element-wise matrix-vector multiplications then produces distinct $\hat{\mathbf{u}}_{i,j}$ used in dynamic routing.

Algorithm 3.1 Forward Propagation

1: **procedure** FP($\mathbf{x}$)
2: $\hat{\mathbf{x}} \leftarrow PrimaryCaps.FP(\mathbf{x})$
3: $\mathbf{u} \leftarrow Duplicate(Activate(Rearrange(\hat{\mathbf{x}})))$
4: $\hat{\mathbf{u}} \leftarrow \mathbf{W} \otimes \mathbf{u}$ ▷ Element-wise matrix-vector multiplication
5: **return** $Routing(\hat{\mathbf{u}}, 3, 2)$ ▷ This is defined in [25]
6: **end procedure**

On the other hand, during back propagation, the corresponding label to the vector, $y_{\mathbf{x}}$, is provided to calculate the error functions. This is performed by the *DerivativeActivationAndLoss* kernel, shown in Algorithm 3.2.

From there, the $\delta\hat{\mathbf{u}}$ vectors are generated by giving a weighted portion of the $\delta\mathbf{v}$ set. These weights are the same $\mathbf{c}$ value set during forward propagation. For the dimensionality transformation step to produce $\delta\mathbf{u}$ vectors, another round of matrix multiplication between $\delta\hat{\mathbf{u}}$ and $\mathbf{W}^T$ is done.

Given that the same data structures from forward propagation will be used for the sake of memory, this step must happen after the $\Delta\mathbf{W}$ calculations are made. These are found by the matrix product of the previous input $\mathbf{u}^T$ and the error gradient for the output, $\delta\mathbf{v}$.

After $\delta\mathbf{u}$ has been calculated for all j along the columns, they are reduced to the left, to compile all the error gradients proposed by each higher level capsule, before having each undergo another "unsquashing". This inverse activation is performed to match the initial activated squashing done during forward propagation. Finally, these vectors are rearranged and handed back to the convolutional layer for tradition convolutional back propagation.

Algorithm 3.2 Back Propagation

1: **procedure** BP($y_{\mathbf{x}}$)
2: $\delta\mathbf{v} \leftarrow DerivativeActivationAndLoss(\mathbf{v}, y)$
3: $\delta\hat{\mathbf{u}}_{ij} \leftarrow \mathbf{c}_{ij}\delta\mathbf{v}_j$
4: $\delta\mathbf{u}_{ij} \leftarrow \mathbf{W}_{ij}^T\delta\hat{\mathbf{u}}_{ij}$
5: $\Delta\mathbf{W}_{ij} \leftarrow \Delta\mathbf{W}_{ij} + \delta\mathbf{v}_j\mathbf{u}_{ij}^T$
6: $\delta\hat{\mathbf{x}} \leftarrow DerivativeActivation(ColReduction(\delta\mathbf{u}))$
7: **return** $PrimaryCaps.BP(\delta\hat{\mathbf{x}})$
8: **end procedure**

3.3 MOEA

3.3.1 Chromosome Definition

Each capsule network is constructed from a configuration structure that holds loss function hyper parameters and dimensionality transformations. These parameters will be decoded from a binary string determined by the individuals of the GA, and they are detailed in Table 3.1.

Note the number of filters in the convolutional layer is the product of the lower level capsule dimensions and the number of vector maps. Therefore, only the factors will be included in the chromosome.

Moreover, although m^+ and m^- were implied to sum to 1, they will be included as two separate parameters in this chromosome. This facilitates the GA in exploring other potential threshold combinations; potentially yielding values that do not sum

Table 3.1: The hyper parameters of capsule networks are encoded in GA bit strings, where each set of bits corresponds to a point in a given range. The last four have a different step size, and they are as precise as double precision allows.

Variable Name	Range	Bits
cnInnerDim	[2-33]	5
cnOuterDim	[2-33]	5
cnNumTensorChannels	[1-32]	5
batchSize	[20-640], step 20	5
m_plus	[0.8, 1.0), step $\frac{1}{32}$	5
m_minus	(0, 0.2], step $\frac{1}{32}$	5
lambda	[0.4, 0.6), step $\frac{1}{32}$	5

to 1, implying the significance of error when the desired class exists.

3.3.2 NSGA-II Algorithm

An initial random population of individuals, P_0, generates an auxiliary population, Q_0 using traditional binary tournament selection, single point crossover, and mutation. The two populations are combined to create R_0, and partitioned into separate Paredo fronts, $\mathcal{F}$. Consequently, all individuals in $\mathcal{F}_i$ dominate all individuals in $\mathcal{F}_{i+1}$. From here, the first Paredo fronts that comfortably fit into N go into P_1. For the solutions in the Paredo front needed to fill N, the individuals are sorted using a novel crowding-comparison operator, $\succ_n$ which helps favor solutions that compromise between multiple objective functions. This main loop continues to generate P_t from P_{t-1} but the binary tournament selection used thereon uses the crowding-comparison operator instead of rank alone [4]. This algorithm is illustrated in Figure 3.3.

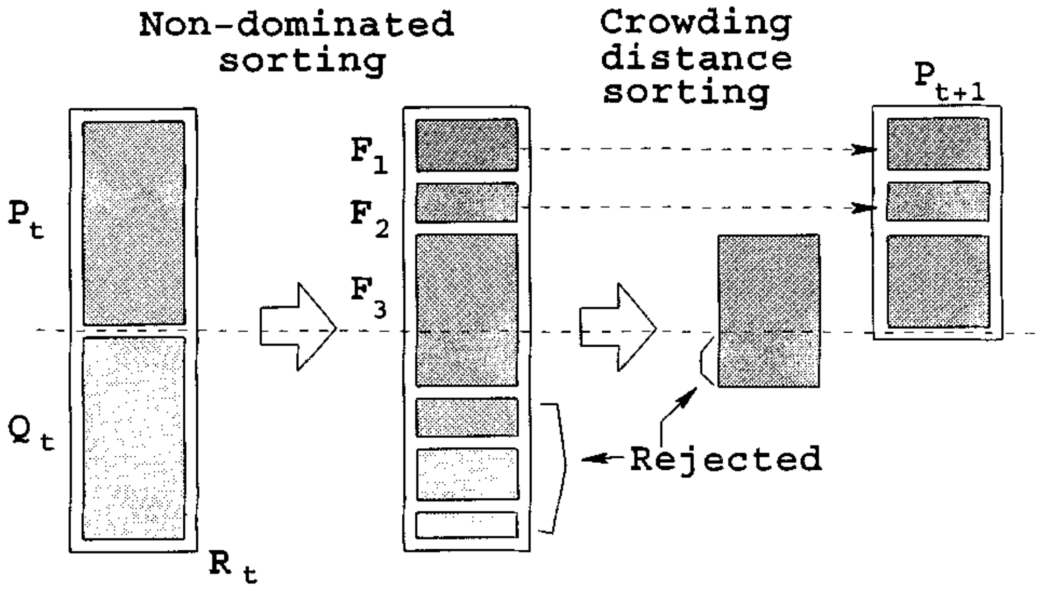

Figure 3.3: The main loop of the NSGA sorts P_t and Q_t by Paredo fronts and put as many as it can into P_{t+1}. The odd front out is sorted by a distance-based crowding operator [4].

Fast Non-Dominating Sort

To sort a population to their Paredo fronts, each individual is compared to all others, making sure to keep track of how many individuals it dominates and the individuals that dominate it. All dominating solutions go into the first front and removed from the initial population, and the process is repeated until the original population is exhausted. Although this algorithm, detailed in Procedure 3.3, is $O(|P|)$, it is still considered fast because of programmer-determined population size.

Crowding-Distance Assignment

To promote a sparser assortment, the distance parameter of an individual is incremented by the average distance to its neighboring individuals. This is done for all separate objective functions, $\mathcal{M}$, sorting a given population by their fitness function outputs each time. Individuals at extremes ranges for each fitness functions are given infinite or near-infinite values to favor high-density, utopian-bound solutions. This distance parameter, updating in Algorithm 3.4 is the second part of the crowded-comparison operator mentioned earlier.

Crowded Comparison Operator

During all but the first GA generation, binary tournament selection uses the crowded comparison operator rather than rank. This helps promote diversity within Paredo fronts while preserving elitism from closer fronts. The operator $\succ_n$ can be defined as:

$$i \succ_n j \textbf{ if } (i_{rank} < j_{rank}) \textbf{ or}$$
$$((i_{rank} = j_{rank}) \textbf{ and } (i_{distance} < j_{distance}))$$

3.4 Use Case Modeling

The use cases of this software will be focused from two perspectives: a user-configurable, terminal-run application, or a lightweight developer API. Users with no interest to upload input images can run an user-specified MOEA with GPU-accelerated capsule

Algorithm 3.3 Fast Non-Dominating Sort

```
1: procedure PAREDO-SORT(P)
2:     for all p ∈ P do                          ▷ Determining the first front, F_1
3:         S_p ← ∅                               ▷ Individuals dominated by p
4:         n_p ← 0                               ▷ Individuals that dominate p
5:         for all q ∈ P do
6:             if p ≻ q then
7:                 S_p ← S_p ∪ {q}               ▷ Add q to the solutions dominated by P
8:             else if q ≻ p then
9:                 n_p = n_p + 1                 ▷ Increment domination counter for q
10:            end if
11:        end for
12:        if n_p = 0 then
13:            p_rank = 1
14:            F_1 ← F_1 ∪ {p}
15:        end if
16:    end for
17:    i = 1
18:    while F_i ≠ ∅ do
19:        Q = ∅                                 ▷ Helper population for the next front
20:        for all p ∈ F_i do
21:            for all q ∈ S_p do
22:                n_q = n_q − 1                 ▷ Decrement the domination counter for q
23:                if n_q = 0 then
24:                    q_rank = i + 1
25:                    Q = Q ∪ {q}               ▷ This belongs in the next front
26:                end if
27:            end for
28:        end for
29:        i+ = 1
30:        F_i = Q
31:    end while
32: end procedure
```

Algorithm 3.4 Crowding-Distance Assignment

procedure CROWDING-DISTANCE-UPDATE($\mathcal{I}$)
  $l = |\mathcal{I}|$
  for $i = 0$ **to** l **do**
    $\mathcal{I}[i]_{distance} \leftarrow 0$
  end for
  for all $m \in \mathcal{M}$ **do** ▷ Updated from perspective of all objective functions
    $\mathcal{I} \leftarrow sort(\mathcal{I}, m)$
    $\mathcal{I}[1]_{distance} = \inf$ ▷ Set the extremes to inf
    $\mathcal{I}[l]_{distance} = \inf$
    for $i = 2$ **to** $(l - 1)$ **do** ▷ Distance incremented by average distance
      $\mathcal{I}[i]_{distance} = \mathcal{I}[i]_{distance} + \frac{f_m(\mathcal{I}[i+1]) - f_m(\mathcal{I}[i-1])}{f_m^{max} - f_m^{min}}$
    end for
  end for
end procedure

network evaluation. The application can also construct and train single capsule networks with user-defined architecture. The developer API holds simple, homemade primitives allocated with CUDA Unified Memory. Alongside it, a library of CUDA kernels specific for Capsule Network, with an example usage of a GPU capsule network and the object-oriented, sequential counterpart.

Section 3.4.1 defines the use cases for the Linux program command line interface (CLI); in Section 3.4.2, the use cases for a development based API.

3.4.1 Application Use Cases

A use case diagram describing the high level operations permitted from the terminal are detailed in Figure 3.4.

Show Options: This will show the options on how to set parameters and run. There will be two modes, a single capsule network mode, and a genetic algorithm mode.

Set Capsule Network Parameters: This will parse user command line input to configuration parameters used for a single Capsule Network. These include: lower

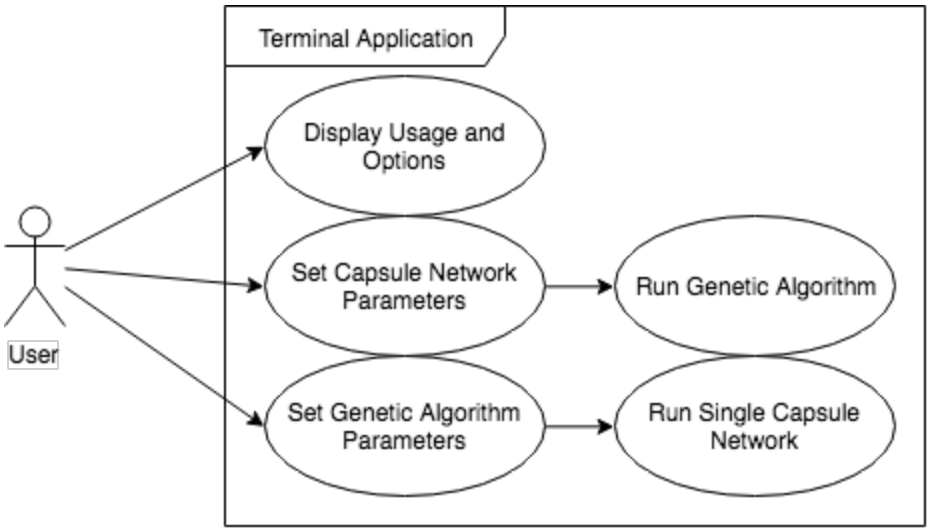

Figure 3.4: The terminal interface provides options for running a single Capsule Network or the entire genetic algorithm. It also allows configuration settings for the user.

level dimension, higher level dim, m_plus, m_minus, lambda, batch-size, and number of tensor channels.

Set Genetic Algorithm Parameters: This will parse user command line input to configuration parameters used for a genetic algorithm. These include: population size, number of generations, probability of mutation, probability of crossover, and database hostname.

Run Capsule Network and Display Progress: This will construct a single capsule network and train it with 300 generations with the MNIST image set [21]. A terminal progress-bar will display progress.

Run Genetic Algorithm and Display Progress: This will run the NSGA-II algorithm for capsule networks and display the parent chromosomes in each generation.

3.4.2 API Use Cases

A detailed use case diagram of the API-based functionalities of this project may be found in Figure 3.5.

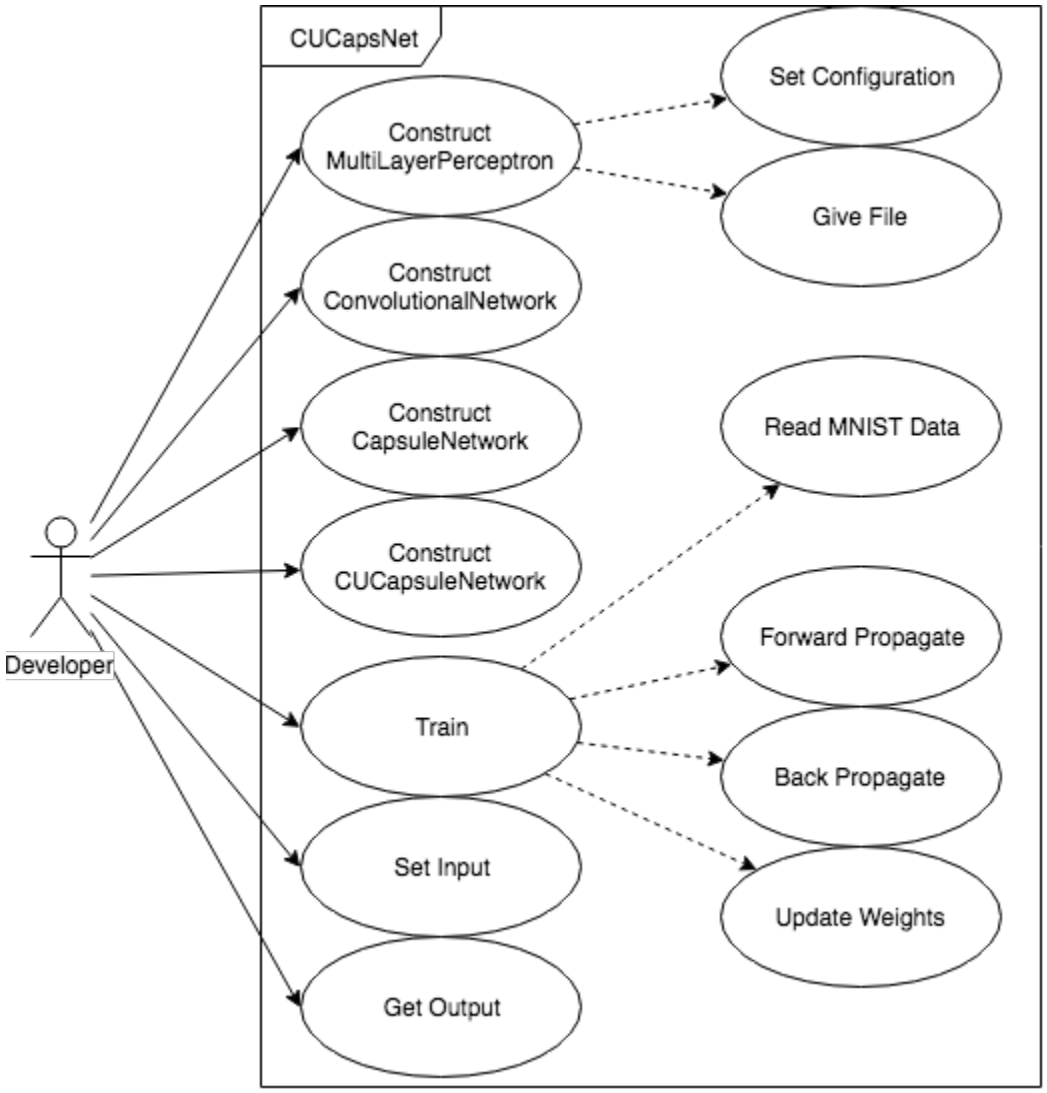

Figure 3.5: The API provides high level abstractions for simple construction and training of Capsule Networks.

Read MNIST Image Data: This will parse and hold MNIST images with their respective labels. This may also translate the data into arrays of doubles.

Construct a Multi-Layer Perceptron from Configuration: This API will construct a multi-layer perceptron (MLP) given just the input layer size, output layer size, and hidden layer size(s).

Construct a Multi-Layer Perceptron from a file: Given a file with trained weights from a given, construct a MLP with the same architecture with those weights.

Construct a Convolutional Network: This will create a convolutional network (CNN) made up of convolutional and pooling layers and an MLP, with set parameters.

Construct a Sequential Capsule Network: This will create an OOP Capsule Network made from an array of capsules; a collection of matrices and vectors specific for capsule feature computation, with set parameters. This holds weights for a sequential convolutional layer followed by two capsule layers and optional reconstruction MLP layer. This will back propagate both margin loss error and reconstruction error.

Construct a Parallel Capsule Network: This will create a SOA Capsule Network made from Unified Memory arrays called Blobs. This holds weights for a SOA convolutional layer followed by two capsule layers.

Forward Propagate Input : Given an image index or a raw input as a vector of doubles for a network, return and/or display the output.

Back Propagate Error: Given an raw error as a vector of doubles for a network, back propagate towards the input layer while holding and not applying error deltas. Return the new error gradient from the input (in case this network is used as an addition to another model).

Update Weights: This will apply the delta weights for a given network using momentum learning [5, 22].

Train Network Model: This will autonomously input a MNIST image for forward propagation, compute Euclidean distance error gradient, back propagate the error, and periodically update the weights.

Set Input: This will allow an array of doubles to be set as input for any of the network models.

Get Output: This allows the developer to get the perceptron of the given network.

Chapter 4

Implementation

4.1 Capsule Network API and Example

This project focuses on the parallel speed up of Capsule Network and the role it plays in making time-infeasible heuristics feasible. Thus, an object-oriented, sequential approach and a pure CUDA implementation are needed for comparison. The lack of CUDA libraries or neural network primitives grant privileges on low-level memory management and architecture otherwise accompanied by overhead. A small implementation of primitives such as multi-layer perceptrons and convolutional layers were created while following object-oriented programming principles (OOP) for sequential-implementation realism. In this sequential implementation, linear algebra is accelerated using a single-threaded, light weight yet optimized CPU library: Armadillo. [27] The OOP Capsule Network class in this project contains an array of "Capsule" class instances for the DigitCaps layer. Each "Capsule" produces feature detection as vectors, a factor of $\mathbf{W}$ matrices and a remapped tensor cube from a sequential convolutional layer.

From a data allocation acceleration point of view, an unabstracted view of the data managed follows the Struct-of-Arrays principle instead of the Array-of-Structs principle. For human imaginative capabilities, a 2D grid of potentially multidimensional values allocated in a single grid best fit this model. This does not create any data redundancy created by other CUDA library accelerations. Each array exists for specific roles for the individual capsules and each kernel for every major mathemat-

ical operation. For the convolutional layer, simple, unoptimized kernels forward and back propagation algorithms were implemented; back propagation algorithms even use CUDA atomic operators.

4.1.1 Sequential Neural Networks

In Figure 4.1, the class diagram features a dispersed design pattern concentrates on interfaces for future improvements. All sub classes and implementations of the `Network` class seen in Figure 4.1 for maximizing future development and modularity, key components of object oriented programming. Although the Multi-Layer Perceptron network uses the naïve perceptron model rather than the much faster linear algebra based approach, this scales nicely when multithreaded.

For absolute speed-up fairness, mentioned in Chapter 5, sequential code has minimal memory movement; accomplished by things constant-reference parameters. The object-oriented based code includes the reconstruction multi-layer perceptron in the Capsule Network implementation. This is omitted in the parallel implementation for Capsule Layer speed up scrutiny. In the CUDA implementation, eighteen kernels were made for individual mathematical operations, each taking shared memory reduction and combining kernels optimizing steps where applicable.

Since deep-learning primitives in the API are expected to have the same operations, Java-style interfaces play a vital role in standardizing class functions. Neither interfaces, nor explicit abstract parent classes are natively supported in C++; therefore, this implementation uses simple class definitions. This enables future programmers from deriving their own unique models with the same functionality. The internal structure of the `Capsule` class include Armadillo primitives: vector and matrix [23, 27][1]. The `ILayer` and `Network` compound interfaces aggregate these and become key templates for the existing networks: `MultiLayer Perceptron`, `Convolutional Network`, and `Capsule Network`[2].

[1]The `Fileable` Interface, for time reasons, was not implemented for all Capsule Networks
[2]The `Network` interface is not explicitly declared in the code

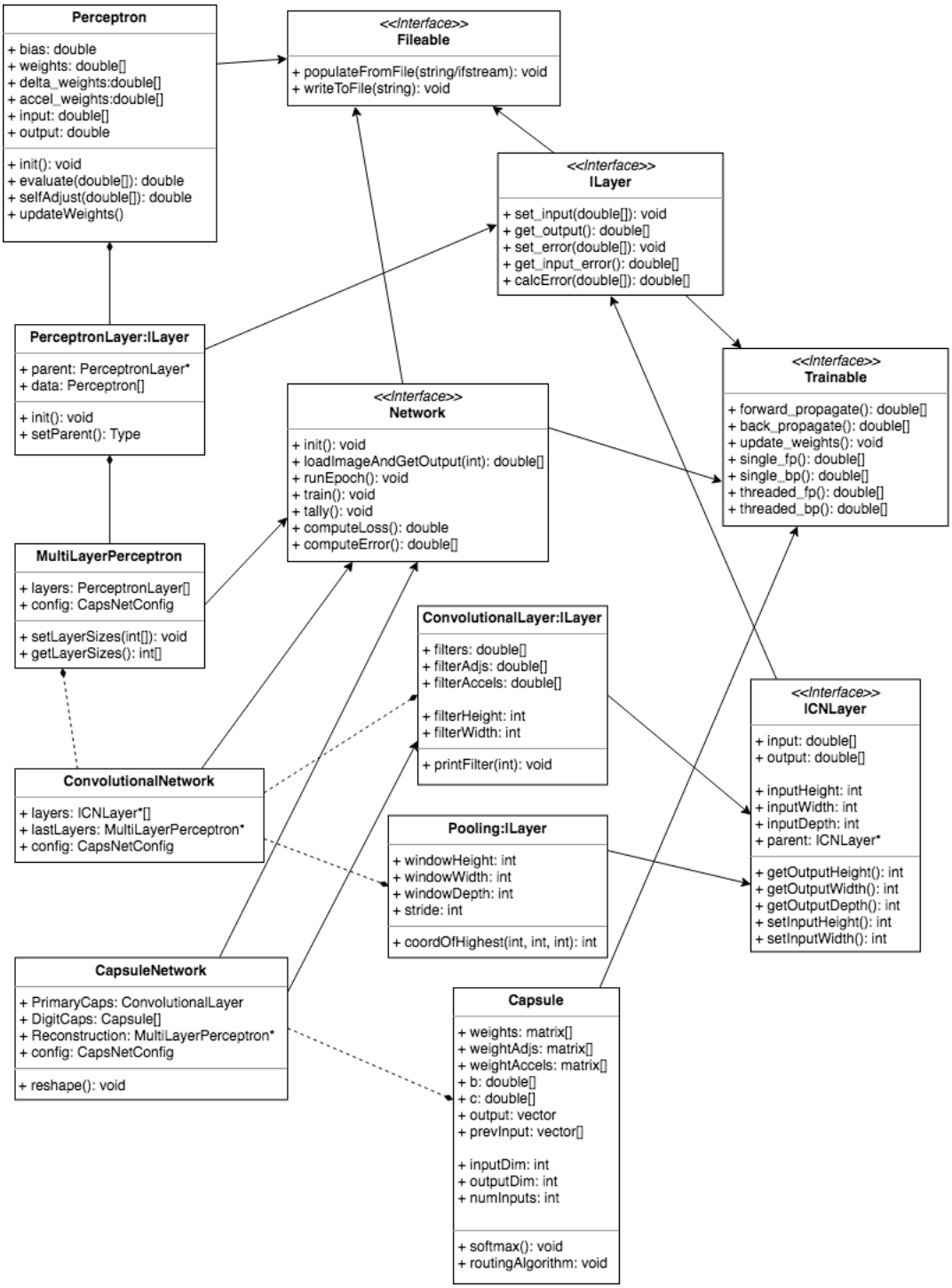

Figure 4.1: The sequential implementation of Capsule Networks and other neural network primitives, following OOP standards

Forward and back propagation operations on the Capsule Network differ from the other scalar based networks. In the Multi-Layer Perceptron and the Convolutional Layer, activation function inputs are scalars, computed by weight reductions of other scalars. Since the Capsule layer uses vectors, mapping a cube output from a convolution layer to vector list is vital. The Capsule Network then passes this list to each Capsule, which hold a specific matrix weight $\mathbf{W}$, softmax weight $\mathbf{b}$, and scalar vector weight, $\mathbf{c}$ for each list vector. An individualized routing algorithm, as discussed in Algorithm 3.1, continuously updates these softmax weights, and then passes the weighted reduction of these vectors to the vector activation function, concluding the capsule forward pass. The activated outputs are compiled by the Capsule Network for back propagation. The Capsule Network then gives each capsule their output back, transformed via an error gradient calculation detailed in Equation 2.7. The Capsule is faced with two tasks: computing a formatted error gradient for each original input given, and use the given error to calculate the weight delta for its matrices. Multiplying the original $\mathbf{c}$ value, with the transform of the weight matrix and the error produce a single vector in the input-sized vector list created. Since this project does not use mini-batching, using $\mathbf{c}$ values explain why back propagation must be computed immediately after each forward pass for an image. The Capsule Network aggregates this list from all higher-level Capsule and remaps it to a feature map cube. This is given back to the Convolutional Layer for normal back propagation processing. When multithreading, each forward and back propagation is performed for each independent Capsule. The number of higher-level Capsules is stable; it is the number of classes, k. A bottleneck arises in sequential back propagation is the vector-list aggregation of error due to its sequential addition.

Each weight in a network adds a dimension of complexity in solving finding a hyper-plane valley maximizing accuracy and minimizing loss. Given the momentum solver, all declarations with modifiable weights are accompanied with two same-sized variables. The adjustments keep track of the weight delta found through back propagation and the acceleration variables direct the weights with short term memory. Tra-

ditionally, in stochastic gradient descent, the weights are updated by the raw deltas, scaled down by a learning rate. In momentum learning, the acceleration weights help keep the network from being stuck in a local optima. If loss is consistently high, for example, the acceleration matrices will gradually increase. This will user the network towards a better solution, and in case local optimum is found along the way, the network will speed over it.

4.1.2 Parallel Capsule Network

An array of doubles was created for easy value assigning and printing using Unified Memory: `CUUnifiedBlob`. Unified Memory does not require explicit pushing and retrieving from the device. Instead, the GPU scheduler takes care of this. The master grids used for this Capsule Network implementation use these dynamic data structures.

The `CUCapsuleNetwork` class, implicitly implementing the `Network` Interface, manages the arrays inside itself, and its convolutional layer accompaniment, the `CUConvolutionalLayer` class. The latter focuses on forward and backward convolutional passes, holding solver-necessitated caches for its filters. Moreover, it focuses on remapping its activated and duplicated output to the **u** grid; the reverse mapping passes the error through back propagation.

Eighteen kernels are declared in the block at the bottom of Figure 4.2, and defined individually in Section 4.1.3. They take in these arrays as reference parameters, and call the similarly named CUDA kernel. The size of each call is different than one another, depending on the projection that helps shared memory reduction. For example, in the kernel `multiVectorReduction`, the **u** array is prepared for tensor-mapping by adding all vectors left wards. This project necessitates a block called per the lower-level capsule axis, per dimension, and one thread per each class allocated. In the `weightedReduceVectors` kernel, a block is allocated per class, per dimension, and one thread per each lower-level capsule axis. The convolutional kernels set up a block per input-filter configuration and a thread for each overlapping cell. In fact,

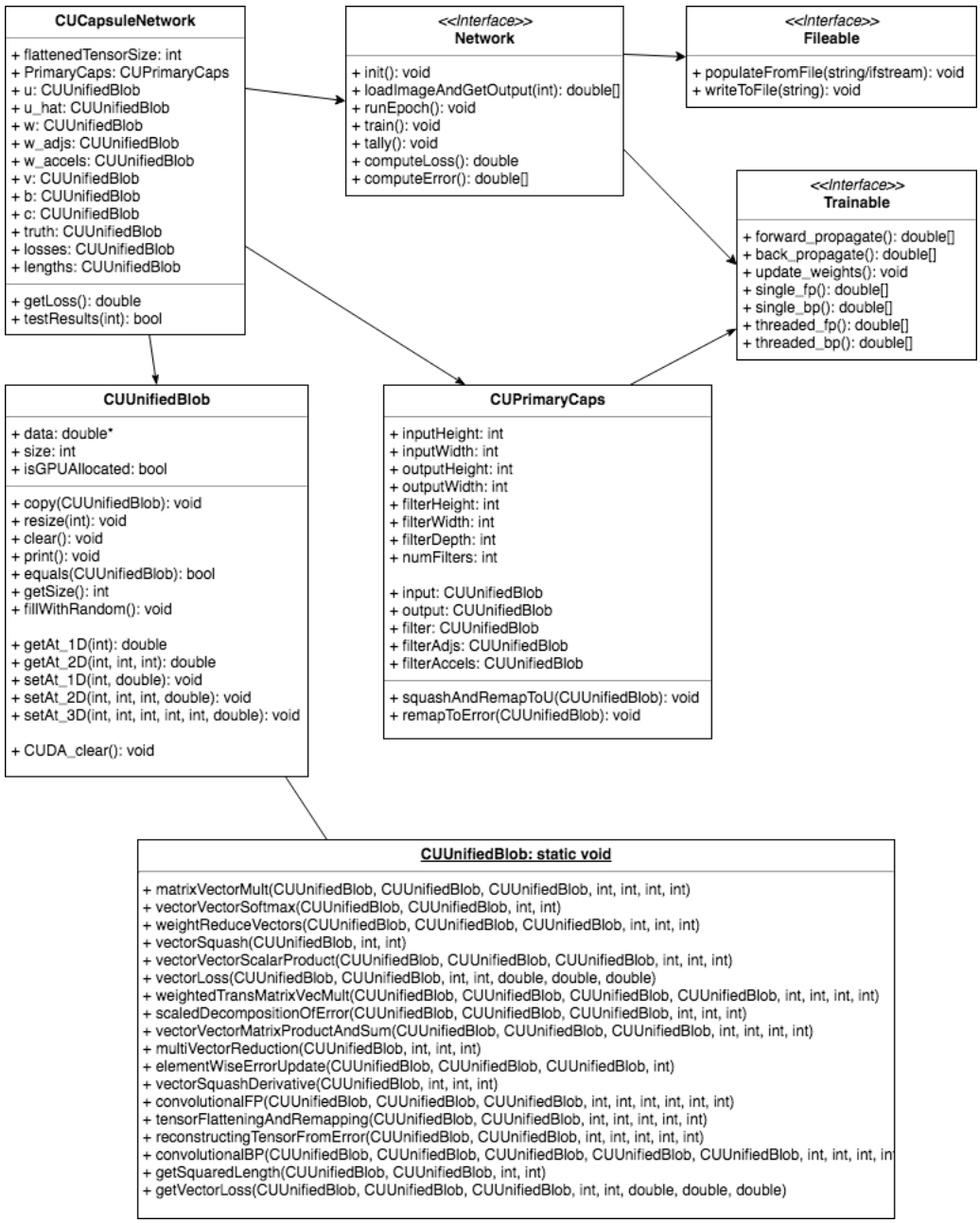

Figure 4.2: The parallel implementation of CUCapsuleNetworks and other GPU accelerated neural network primitives, following Struct-Of-Array Conventions. The bottom object is not an entity, but a list of unique kernels (wrappers). They are given the `CUUnifiedBlob` instances as reference parameters.

the convolutional back propagation kernels even use atomic CUDA calls.

4.1.3 Custom CUDA Kernels

Each kernel has different block and thread dimensions, depending on the projection that best leverages shared memory reduction. Whenever there are more values that need to be reduced (typically along the lower level capsule axis) than the maximum number of threads allowed per block (1024 for CUDA 7 and above [26]), threads will "wrap around" and stride by 1024 for all values. They are individually detailed below:

- **matrixVectorMultiplication** - This function performs individual matrix vector multiplication between $\mathbf{u}$ and $\mathbf{u}$ to produce $\hat{\mathbf{u}}$ This function allocates a block per lower-level capsule, per class. Each block has one thread for every cell in the higher-level vector

- **vectorVectorSoftmax** - This updates scalar values $\mathbf{c}$ from those in $\mathbf{b}$. There is one block per class, and a thread for every value.

- **weightReduceVectors** - Given a scalar weight, $\mathbf{c}$, for every vector in $\hat{\mathbf{u}}$, this function will reduce them vertically and populate $\mathbf{v}$. There is one block per class and dimension, one thread per lower-level capsule.

- **vectorSquash** - Sequentially annexed vectors will be scaled according to the vector activation function, detailed in Eq. 2.5. There is one block per vector, and one thread per dimension.

- **vectorVectorScalarProduct** - The significance values in $\mathbf{b}$ are incremented by the scalar output after dotting $\hat{\mathbf{u}}$ and $\mathbf{v}$. There is one block per lower-level capsule, per class, and one thread per dimension.

- **vectorLossFunction** - This performs the margin loss formula detailed in Eq. 2.7, producing $\delta\mathbf{v}$ A vector with respective T values for each k is also provided. There is one block per class, one thread per dimension.

- **scaledDecompositionOfError** - This distributes the error from $\delta\mathbf{v}$ down to $\delta\hat{\mathbf{u}}$ using the forward propagation weights, $\mathbf{c}$. There is one block per class, per lower-level capsule, and one thread per higher-level dimension.

- **weightedTransMatrixVecMult** - The error from a pre-weighted $\delta\hat{\mathbf{u}}$ is passed to its $\delta\mathbf{u}$ through the inverse, element-wise multiplication of $\mathbf{u}$. There is one block per class, per lower-level capsule, and one thread per lower-level dimension.

- **vectorVectorMatrixProductAndSum** - This will compute the matrix product of the previous input to the capsule layer, $\hat{\mathbf{u}}$ and the error, $\delta\mathbf{v}$. There is one block per class, per lower-level capsule, and one thread per cell in the matrix $(d_l \times d_{l+1})$.

- **multiVectorReduction** - This will reduce all the vectors from each class in $\delta\mathbf{u}$ to the left-most column. There is one block per lower-level capsule, and one thread per lower-level dimension.

- **elementWiseErrorUpdate** - During updating, this will perform momentum solving for $\mathbf{W}$ and clear out its filter cache, $\Delta\mathbf{W}$. There is one block, one thread per element.

- **vectorSquashDerivative** - Much like the inverse of "vectorSquash", this will scale the vectors by the operations defined in Eq. 2.9. This kernel is custom and assumes these vectors will be in the left-most column of an array used as $\delta\mathbf{u}$. There is one block per lower-level capsule, with one thread per dimension.

- **convolutionalDotProduct** - This provides the convolutional forward propagation operation for the convolutional layer. There is one block allocated for each output cell; one per filter, output feature map height, and output feature map width. For each block, there is a thread for each filter projection onto its portion of the input map; one for the input (and filter) depth, filter height,

and filter width. This is generalized for potentially three dimensional outputs, provided from similar convolutional layers or RGB images.

- **tensorFlatteningAndActivatedRemapping** - This kernel projects the output of a convolutional layer to the initial **u** grid. At the same time, it squashes the vectors according to Eq. 2.5. There is one block per virtual output vector, and one thread per each dimension; the grid x dimension and the block x dimension effectively cover all feature maps in the cube.

- **reconstructingTensorFromError** - As the inverse function of "tensorFlatteningAndActivatedRemapping", this function remaps the compiled error from **u** back to a tensor cube for convolutional back propagation. However, it does not scale the vectors back through a deactivation formula. Much like its counterpart, there is one block per virtual output vector and one thread per each dimension.

- **convolutionalBackPropFromError** - This kernel is for the back propagation step of a convolutional layer. As mentioned, the atomic function is required for a naïve implementation, given that a particular input cell affects multiple output cell through different filter and filter strides. The atomic function was explicitly created since double atomic functions were not supported[3]. Like the forward propagation kernel, there is one block per output of the convolution, and one thread per input-filter pairing for that block.

- **getSquaredLength** - This kernel simply goes through **v** and computes the length (to another vector) for easy debugging.

- **getVectorLoss** - This kernel computes the overall loss of an image from the guess **v** using solely Eq. 2.7.

[3]CUDA 8.0 and above supports atomic functions on other primitives, such as longs, doubles, and shorts. However, the explicit functionality was written for backwards compatibility; to older architectural GPU cards that cannot support the newer framework.

4.2 NSGA-II Implementation

The individuals from first generations of a GA have genetic diversity from the random initialization. As genes start to spread through each generation, duplicates arise and eventually dominate the population. Individual chromosome evaluation becomes infeasible for duplicates, given the time it takes to construct and train a neural network. The PostgreSQL database usage in this project offers speed up in the overall structure; an uncommon yet useful feature for MOEAs.

The C++ implementation of NSGA-II used in this project, however, is simple. The "Individual class is a child of a standard vector[4] of boolean values, and it decodes its bit-string to a specific capsule network configuration. A `Population` class, a child of a standard vector of `Individuals`, extracts statistics from its data, used by a higher-level `GA` class. The accompanying `ParedoFront` class is a small wrapper, using a pointer-only interface for faster sorting and crowding-distance calculations. The `GA` class sorts and produces generations of these populations through NSGA-II operations. This architecture is illustrated in Figure 4.3

NSGA-II requires the sorting of the solutions per their performance to each objective function, individually. This sorting updates crowding distance parameters, and thus is optimizable. The `ParedoFront` class abstracts a population as an array of pointers; they are more lightweight and easy to move. Each `ParedoFront` refers to a subset or the entire population. Moreover, a static function, `sortFastNonDominated` returns an array of these fronts, order by their dominance.

When a new individual is encountered, a Capsule Network (sequential or parallel) is constructed and trained for 100 and 300 epochs. Its results are stored in the database, with its bitstring serving as the primary key. Figure 4.4 shows the one entity ERD implemented in the PostgreSQL database for these individuals. When an evaluated individual is encountered, its statistics are retrieved from the database and applied to the instance.

[4]The vector class comes from the C++ standard templated library (STL)

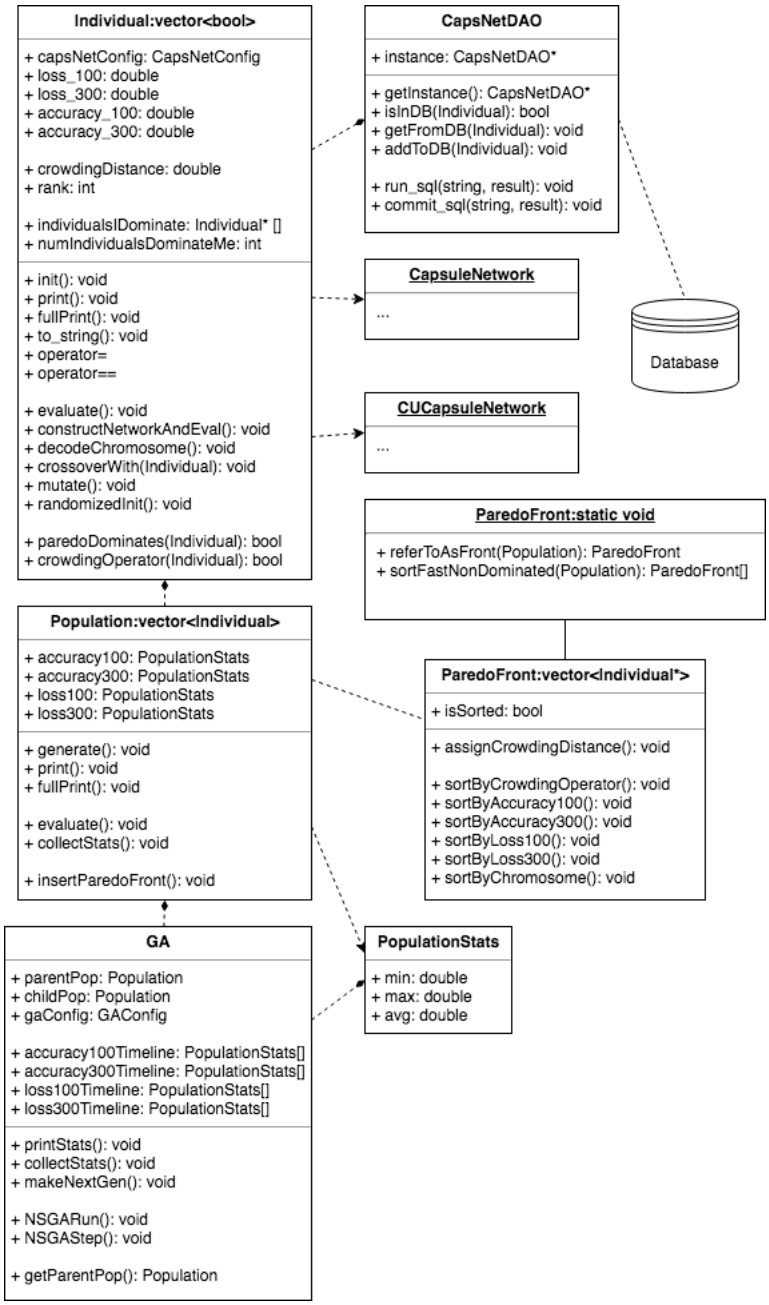

Figure 4.3: The NSGA-II implementation is simple; aggregating individuals into populations, and shorthanding populations into Paredo Fronts. A Paredo Front is abstracted as a pointer array for easily sorting for individual updating. The CapsNetDAO helps speedup Individual evaluation by creating SQL queries and checking for duplicate networks to prevent wasteful duplicate evaluation.

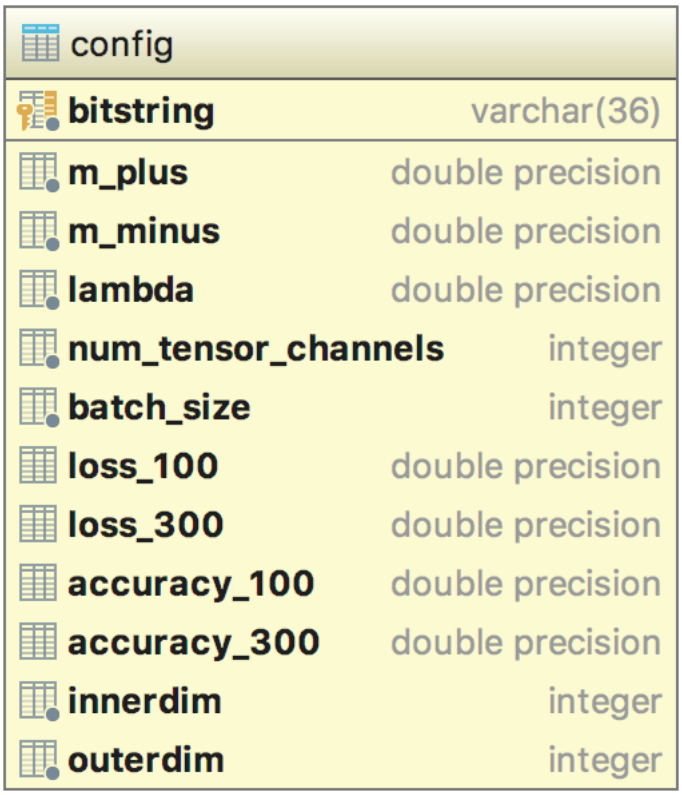

Figure 4.4: This PostgreSQL entity uses the individual bitstring as the primary key and contains decoded hyperparameters and Capsule Network metrics as other row attributes

Chapter 5

Results

5.1 Architectural Results

5.1.1 Single Capsule Network Results

Statistical-based learning models are trained using thousands of Epochs of a training set, and then tested on a separate test set. The second test set contains images the network has not seen, and only forward propagation is performed during testing to validate the progress of the training session. Such models are then evaluated using forward-propagation performance with the testing session, sometimes called a validation set. The performance of the neural network is measured through the total loss and the accuracy measured at each iteration. Figure 5.1 and Figure 5.2 shows the performance of two such Capsule Networks; a faithful recreation of the source Capsule Network theory paper, and the best chromosome from the MOEA. The source Capsule Network has an inner dimension of 8, an outer dimension of 16, batch size of 250, 32 tensor channels, $\lambda = 0.5, m^+ = 0.9, m^- = 0.1$. The Capsule Network yielded by the MOEA had an inner dimension of 22, an outer dimension of 2, batch size of 160, 2 tensor channels, $\lambda = 0.44375, m^+ = 0.8125, m^- = 0.18125$.

Although the evolved configuration outperformed the original architecture in terms of early accuracy, the loss calculations appear to be worse. In the evolved network, the loss starts at a much lower point, and drops but then slowly arises. Towards 200 iterations, however, a downwards parabolic shape emerges, but there are nevertheless not enough points to make the claim that loss appears to go downwards

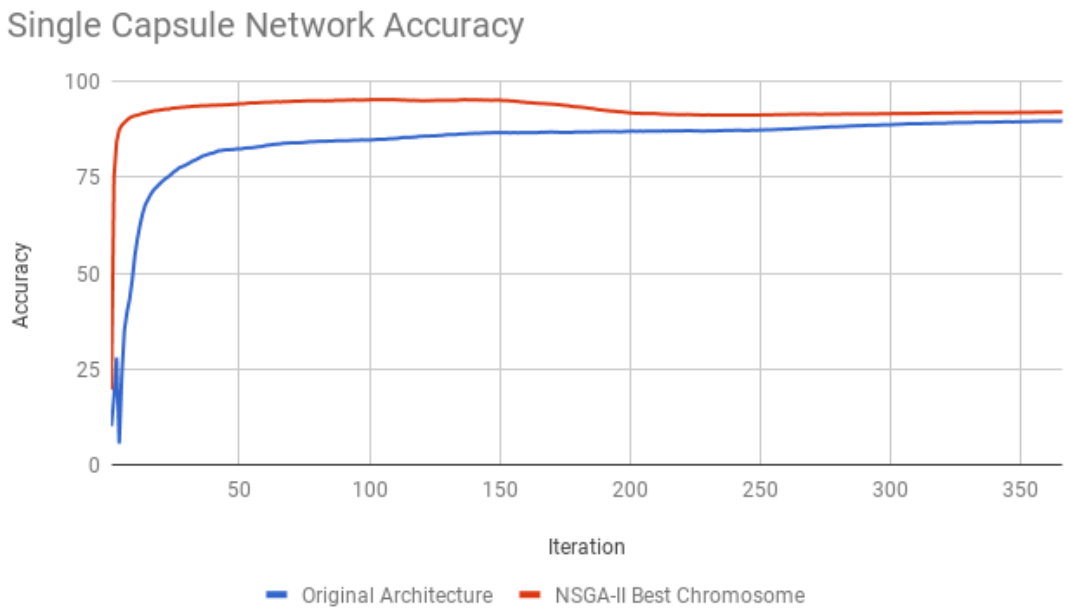

Figure 5.1: The Accuracy for the NSGA-II configuration outperforms the original architecture, despite both having a small dip just before 200 iterations, due to local minima in $\mathbf{W}$ space.

Figure 5.2: The Loss of the NSGA-II configuration starts out at a smaller pace, but after finding a local minima, starts to rise again at 200 iterations.

after a while. It is important to note both accuracy and loss were considered equally significant in the MOEA.

5.1.2 MOEA Results

The genetic algorithm was run with a population of 30 and ran for 50 iterations. Typically, to track the progress of a genetic algorithm, the maximum or average fitness of each populations is plotted against each iterations.

The performances of the MOEA are detailed in the following photos. Figure 5.3 shows the accuracy statistics. Within the first few iterations, the accuracy after 300 iterations are, on average, higher. However, they decrease to stabilize to roughly 3% lower (about 300 training images)

The generation of the child population is elitist due to the binary tournament selection and the non-dominating sorting. Due to this elitist nature, the graphs will converge sooner than a canonical genetic algorithm. To counter this behavior, crossover and mutation probabilities are set high, 0.5 and 0.01.

5.2 Parallel Capsule Network Results

The number of filters in the preceding convolutional layer must change accordingly. For example, for two tensor channels and an inner dimension of eight, the convolutional layer must have 16 filters; for forty tensor channels, 320 filters. This strongly influences the computation required by the entire network, justifying it as the independent variable. Varying the number of channels in the tensor also helps better profile the program better through its throughput measurements. Reported below are the time taken to perform forward propagation, backward propagation and epoch timings, along with the equivalent speed up and throughput calculations. All timings seen in Figure 5.5 and Figure 5.6 are an average 30 statistical runs, after removing the highest and lowest outliers. To ensure sequential optimization, the sequential version uses the Armadillo library for linear algebra operations (matrix-vector multiplication) [27].

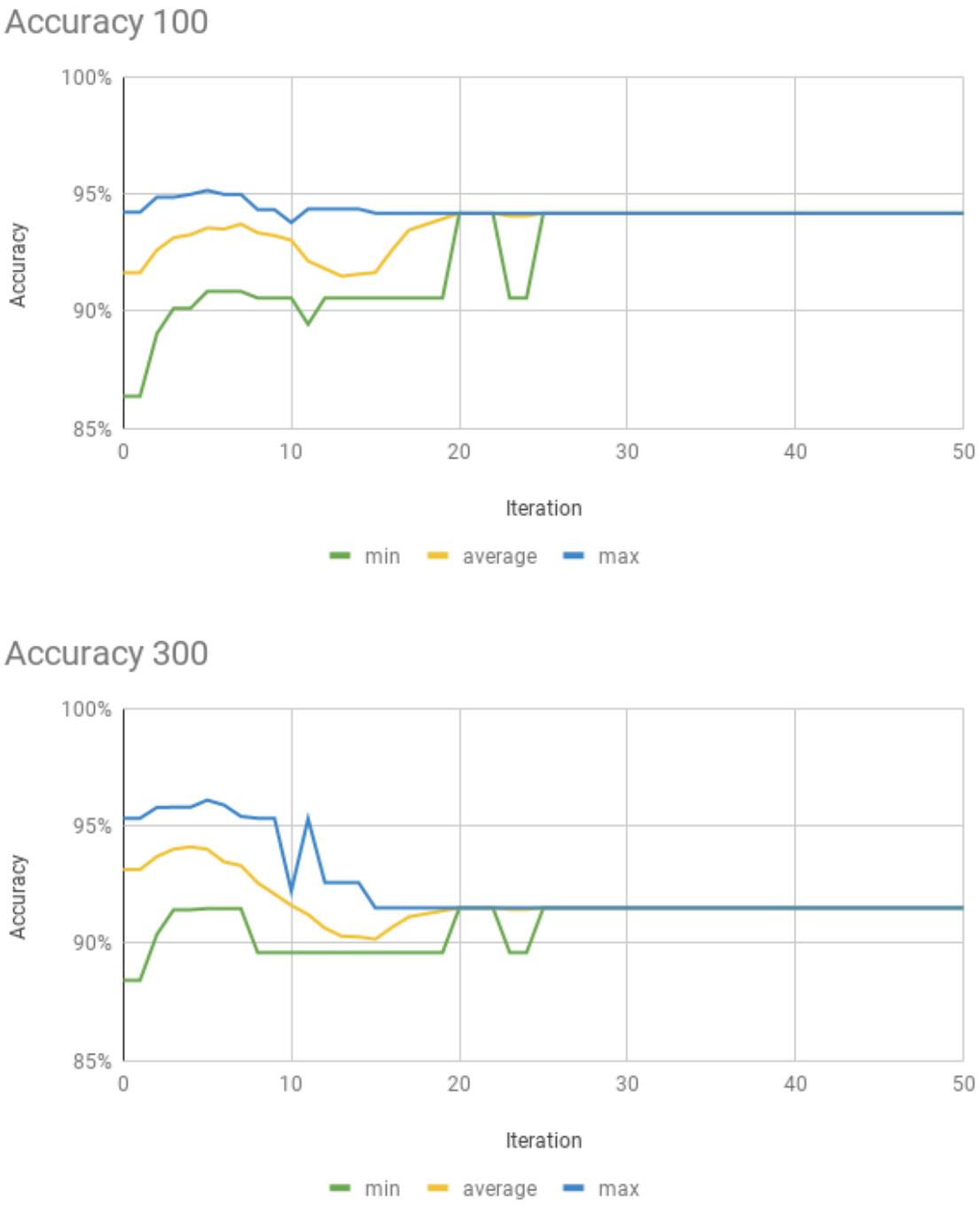

Figure 5.3: The minimum, average, and maximum accuracies of the population of the MOEA after 100 epochs (top) and after 300 epochs (bottom).

Figure 5.4: The minimum, average, and maximum loss values of the population of the MOEA after 100 epochs (top) and after 300 epochs (bottom).

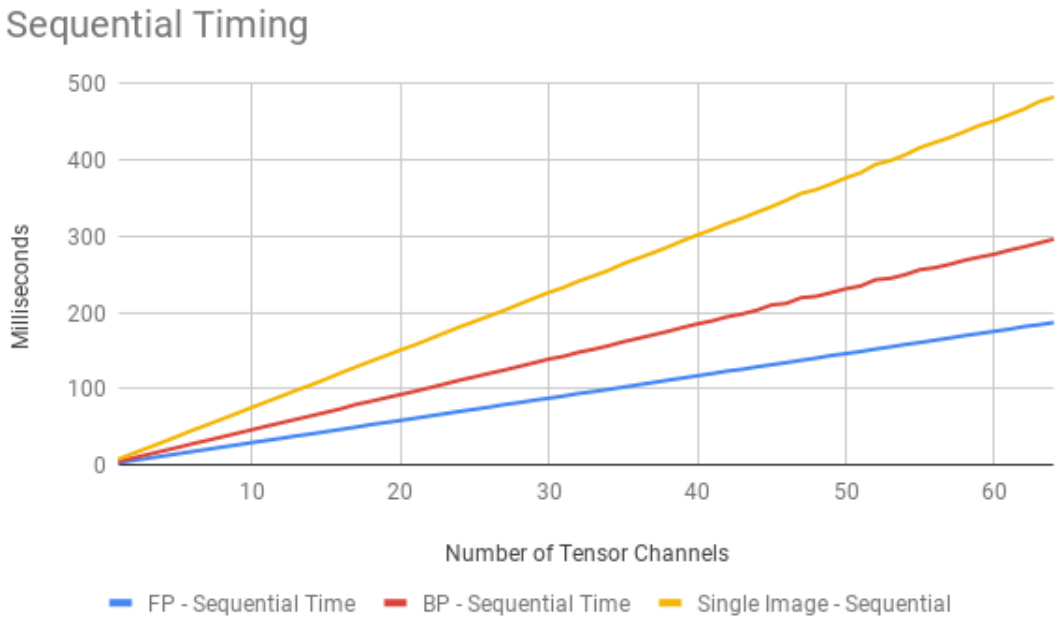

Figure 5.5: Forward propagation takes less time than back propagation in the sequential version of these methods since there is no data movement (despite the inner loop found in Dynamic Routing [25].) These methods are not multi-threaded and are compiled without compiler optimization flags.

In Figure 5.5, back propagation is shown to have a higher percentage of data computation rather than data movement, since data movement is not a hindering factor on CPU-based operations. Parallel timings in Figure 5.6 further illustrate the GPU bottleneck since back propagation is faster than forward propagation. The single image trend lines show the computation time of processing an entire image, combining forward and back propagation.

The gap between forward propagation and back propagation steadily increases over larger capsule layer sizes, despite the transferred data amount being the same. This indicates the back propagation problem scales better to GPUs than forward propagation. However, forward propagation also adds to this difference since it includes the time taken to copy the original photo information over to the GPU itself, an extra overhead not found in the sequential version.

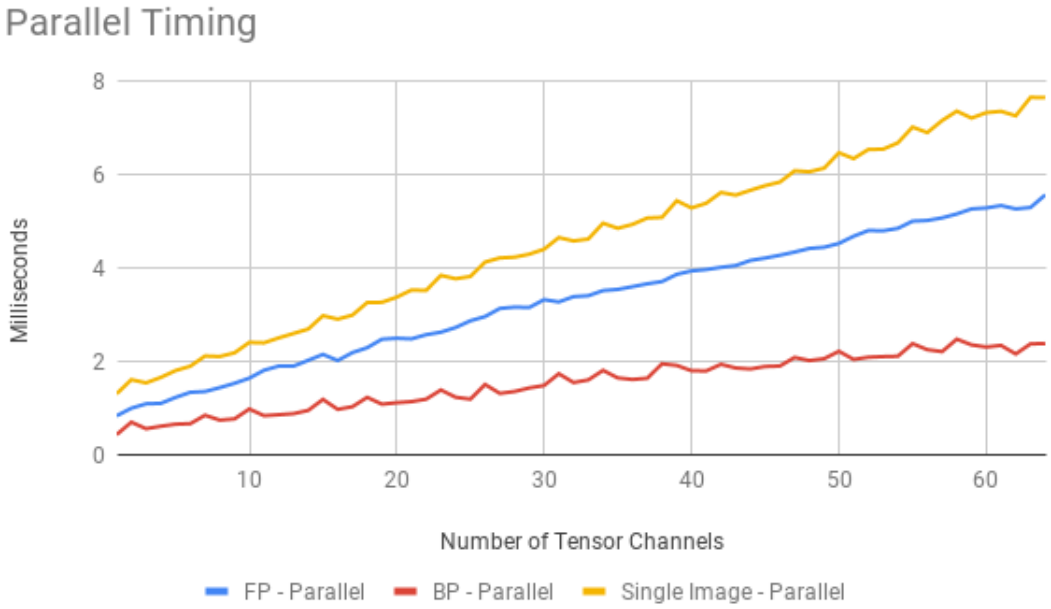

Figure 5.6: Back propagation is clearly faster than forward propagation due to lack of communication overhead in data movement. Note the scale as compared to Figure 5.5.

5.2.1 Speedup

The speedup of a program compares the sequential and parallel methodologies in high performance computing and performance validation.

Speedup is the proportion of the sequential time over the parallel time, $\frac{T_s}{T_n}$, where n corresponds to the number of processors used. However, n represents a variance of input size rather than processors, due to GPU hardware architecture. Nevertheless, theoretical limitations observed in CPU parallelism still apply. According to Amdahls Law, an asymptotic upper bound on the speedup improvement of any given parallel algorithm exists [30]. It hinges on the given overhead of the size of p, the parallelizable fraction of the application program. Speed up, S, is defined as

$$S = \frac{N}{(p * n) + (1 - p)} \tag{5.1}$$

To measure the speedup, capsule networks were generated with varying numbers of vector channels. This changes the lower level capsule layer size, dependent on the convolutional layer height and width, both set to 6 for the sake of referential integrity.

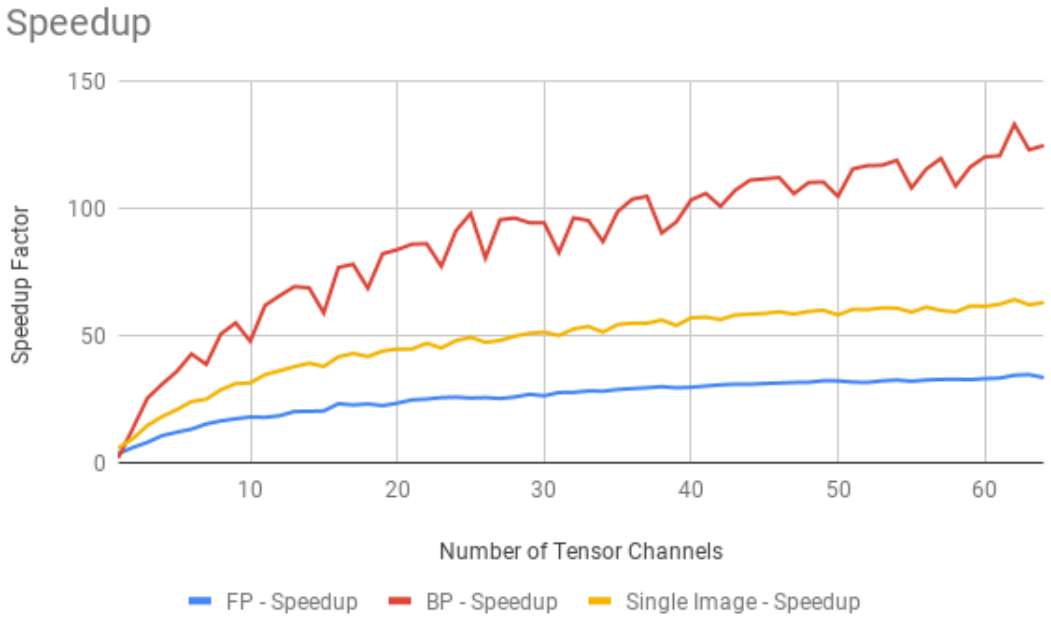

Figure 5.7: Speedup of these methods start to slow down between 10-15 tensor channels (360-540 lower level capsules) as these methods increase due to Amdahl's law. Note that back propagation, which only communicates resulting $\mathbf{v}_j$ errors back to the host, a constant $k \times d_{l+1} = 160$ values, has higher speedup than forward propagation, which requires the movement of a 28×28 sized image from the MNIST dataset. [21].

Thus, the number of rows in the $\mathbf{u}$ grid (and other grids) is based off the height (6), the width (6), and the tensor channel size.

For the equal tensor channel values found in [25], forward propagation obtained 32x speedup and back propagation obtained 116x speedup. These methods were able to obtain up to 33x speed up for forward propagation and 130x speed up for back propagation procedures alone.

5.2.2 Throughput

Traditionally, efficiency is calculated in multi-CPU application to measure resource and memory exploitation in distributed algorithms. However, GPU speedup is accompanied with throughput; how many floating point operations (FLOPS) are computed in a given amount of time. For GPUs, throughput focuses on the bandwidth

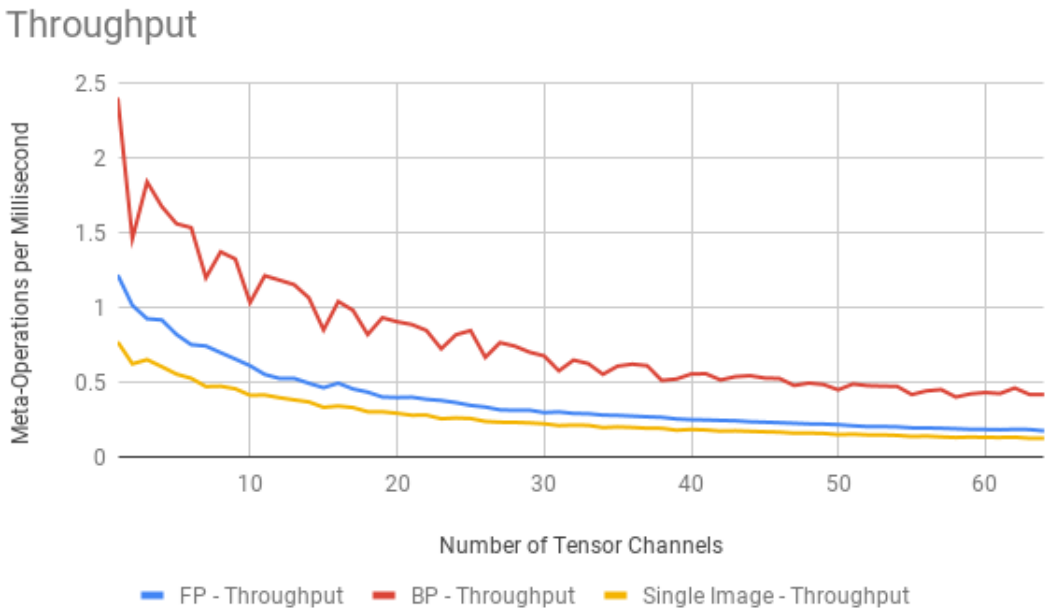

Figure 5.8: The throughput is measured by a factor the number of floating points required to compute (not the operations) at the variable layer divided by the amount of time taken to complete the meta-operation. These floating points are the ones for the interim layer only.

of the data flow rather than hardware architecture; an more appropriate, important distributed algorithm metric. For the graph seen in Figure 5.8, throughput was computed solely on the amount of computation used during the variable layer, the convolutional output to tensor, for easy comparison. In forward and back propagation, this equates to a $6 * 6$ grid, multiplied by the appropriate number of depth channels, divided by the processing time in seconds. This is similar for single image processing, where throughput is obviously slower due to the concatenation of these two operations. The warp-based valleys found during the speed up of the program also make an impact here; computation is less efficient when hardware resource allocation is not optimized. There is no surprise that back propagation can produce a higher throughput than forward propagation, even when the hidden layers are increased. Contributing to this advantage are lack of required CPU-to-GPU data communications and no iterative dynamic routing necessary.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The methods described in this thesis focus on providing a memory allocation map helping network evaluation feasible enough for practical use. The idea of this demonstration is to abstract forward and backward propagation operations to be used as a potential future CUDA library. Genetic Algorithms are an example of a meta-heuristic rendered infeasible without this device optimization. The entire project was written in vanilla C++ and CUDA code, with little to no third party libraries used, to showcase its effectiveness and standalone speedup. Although mostly a back-end project for use in more complicated machine learning libraries, a terminal-based application helps demonstrate these two domains while being verbose and displaying progress and results.

The advent of high performance computing, thanks to hardware development in the past few decades, accelerates the testing and validation of theoretical neural networks and other machine learning paradigms. Although they enable fast modeling and implementations for researchers coming up with new techniques, the machine learning libraries stress for being generalized introduces structural overhead. It is expected when researchers are especially looking towards challenging primitive data structures. This explains the proliferation of such deep learning techniques, not present during the infancy of artificial intelligence research. This project addresses such a problem for a newly-discovered, yet continuously evolving field of neuron grouping techniques;

Capsule Networks are only the stepping stone for multi-dimensional perceptron models.

6.2 Future Work

6.2.1 Preexisting Tools

This clean CUDA implementation of capsule networks and the surrounding MOEA model placed a majority of focus on low level memory management. Preexisting tools and libraries for CUDA, as well as other established techniques and models for MOEAs were ignored for the sake of integrity to this goal. The first convolutional layer found in a capsule architecture was implemented naively, effectively hindering overall performance with atomic functions, for example. In a future work, using existing primitives for these layers may greatly increase speedup.

Another possible improvement using less external CUDA libraries would be to use built-in vectors. Accompanying the Unified Memory acceleration found in CUDA 7 and above, host and device vectors mimic standard sequential vectors. With this method, simple assignment and other memory allocation subroutines can be optimized instead of being implemented from scratch.

6.2.2 Extra Evolutional Parameters

For the sake of referential intergrity, the height and width of the tensor cube output from the convolutional layer was fixed to 6. These values seemed arbitrary from the original paper, and could be included in the chromosome for systematic evolution. Limiting the lower capsule layer to a large filters (or small filters with large stride) from the convolutional layer might inhibit significant feature detection from the input.

Moreover, the architecture of capsule networks could be evolved. The current implementation in this paper evolved a fixed architecture; a convolutional layer, and two capsule layers. Additional capsule layers could be appended for more indepth feature detection. Nonintuitive architectural updates have been discovered with genetic

algorithms [15].

6.2.3 Concurrency

Currently all kernels run in CUDA are called sequentially, with host blocking code. This introduces kernel call overhead being fully expanded; including allocation of blocks, stack frames, and thread warps. Moreover, this does not let small enough kernels with mutually exclusive data from being called concurrently, thus taking less time. Kernel call optimization and methodical multi-threading could increase speed up.

6.2.4 Multi-GPU Implementations

There are two approaches that may be considered when distributing these memory methods between multiple devices: a naïve, per image computation, or a mini-batched, grid-dividing approach. Any attempt at multi-GPU distribution, however, must take into account the interdimensional dependencies of the presented grid formats and potential bottleneck latency from inter-device communication.

First, the naïve implementation is very simple but would maintain some redundant memory information amongst all devices in question. Every device would contain its own $\delta\mathbf{W}$ cache, updated during back propagation. However, each weight updating step must then involve a computationally-blocking, trans-device reduction step. This step collects all cache, reduces them, and then rebroadcasts the new values to everyone to continue computation. This satisfies inter-grid value dependencies found during soft-max and reduction operations. Even with mini batching techniques (adding depth to the memory grids), input based device independence would have communication overhead, despite optimized communication techniques.

A grid-dividing approach, on the other hand, would still have to face these two problems. If one divides along the higher-level capsule axis, there are memory synchronization requirements in two different scenarios. During forward propagation, the convolutional output (assuming to also be distributed with a pre-existing technique,

a problem in itself) would have to be broadcasted to all devices for mapping to $\mathbf{u}$ vectors. In the same spot during back propagation, vectors from all $\delta\mathbf{u}_j$ must be reduced to $\delta\mathbf{u}$ to give to the convolutional layer. Forward propagation is a much simpler task due to the repetitive nature of the vectors for each higher-level technique. However, both scenarios introduce potential communication latency.

If one devices along the lower-level capsule axis, then the problem is shifted to two different operations, soft-max of $\mathbf{b}$ to $\mathbf{c}$ and weighted vector reduction, $\mathbf{v}_j = \sum_i \mathbf{c}_{ij}\hat{\mathbf{u}}_{ij}$. It is the readerś responsibility to visualize the needed communication for this step.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016. arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467 (visited on 08/12/2018).

[2] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. Cudnn: efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014. arXiv: 1410.0759. URL: http://arxiv.org/abs/1410.0759 (visited on 08/12/2018).

[3] Adam Coates, Brody Huval, Tao Wang, David J. Wu, Andrew Y. Ng, and Bryan Catanzaro. Deep learning with cots hpc systems. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1337–III–1345, Atlanta, GA, USA. JMLR.org, 2013. URL: http://dl.acm.org/citation.cfm?id=3042817.3043086 (visited on 08/12/2018).

[4] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002. ISSN: 1089-778X. DOI: 10.1109/4235.996017.

[5] Wei Di, Anurag Bhardwaj, and Jianing Wei. *Deep learning essentials: your hands-on guide to the fundamentals of deep learning and neural network modeling*. English. Packt Publishing, Birmingham, UK, 1st edition, 2018. ISBN: 9781785887772;1785887777;

[6] Korry Douglas and Susan Douglas. *PostgreSQL*. New Riders Publishing, Thousand Oaks, CA, USA, 2003. ISBN: 0735712573. (Visited on 08/12/2018).

[7] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudk, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA. PMLR, 2011. URL: http://proceedings.mlr.press/v15/glorot11a.html (visited on 08/12/2018).

[8] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989. ISBN: 0201157675.

[9] David E. Goldberg. Genetic algorithms in search optimization and machine learning. *AI Magazine*, 12:102–103, 1989.

[10] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In GREGORY J.E. RAWLINS, editor. Volume 1, Foundations of Genetic Algorithms, pages 69 –93. Elsevier, 1991. DOI: https://doi.org/10.1016/B978-0-08-050684-5.50008-2. URL: http://www.sciencedirect.com/science/article/pii/B9780080506845500082 (visited on 08/12/2018).

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[12] Robert Hecht-Neilsen. Iii.3 - theory of the backpropagation neural network*. In Harry Wechsler, editor, *Neural Networks for Perception*, pages 65 –93. Academic Press, 1992. ISBN: 978-0-12-741252-8. DOI: https://doi.org/10.1016/B978-0-12-741252-8.50010-8. URL: https://www.sciencedirect.com/science/article/pii/B9780127412528500108 (visited on 08/12/2018).

[13] Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. Matrix capsules with EM routing. In *International Conference on Learning Representations*, 2018. URL: https://openreview.net/forum?id=HJWLfGWRb (visited on 08/12/2018).

[14] Kenneth A. De Jong. Genetic algorithms are not function optimizers. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms*. Volume 2, Foundations of Genetic Algorithms, pages 5 –17. Elsevier, 1993. DOI: https://doi.org/10.1016/B978-0-08-094832-4.50006-4. URL: http://www.sciencedirect.com/science/article/pii/B9780080948324500064 (visited on 08/12/2018).

[15] Yasusi Kanada. Optimizing neural-network learning rate by using a genetic algorithm with per-epoch mutations, July 2016.

[16] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Wu. An efficient k-means clustering algorithm: analysis and implementation. English. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, 2002.

[17] Andy J. Keane. Genetic algorithm optimization of multi-peak problems: studies in convergence and robustness. *Artificial Intelligence in Engineering*, 9(2):75 – 83, 1995. ISSN: 0954-1810. DOI: https://doi.org/10.1016/0954-1810(95)95751-Q. URL: http://www.sciencedirect.com/science/article/pii/095418109595751Q (visited on 08/12/2018).

[18] Christof Koch and Idan Segev, editors. *Methods in Neuronal Modeling: From Ions to Networks*. MIT Press, Cambridge, MA, USA, 2nd edition, 1998. ISBN: 0262112310.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf (visited on 08/12/2018).

[20] Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. ISSN: 0018-9219. DOI: 10.1109/5.726791.

[21] Yann LeCun and Corinna Cortes. MNIST handwritten digit database, 2010. URL: http://yann.lecun.com/exdb/mnist/ (visited on 08/12/2018).

[22] Ali Minai. Acceleration of backpropagation through learning rate and momentum adaptation. *Proceedings of the International Joint Conference on Neural Networks*:676–679, 1990. URL: https://ci.nii.ac.jp/naid/10008947031/en/ (visited on 08/12/2018).

[23] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[24] Dennis W. Ruck, Steven K. Rogers, Matthew Kabrisky, Mark E. Oxley, and Bruce W. Suter. The multilayer perceptron as an approximation to a bayes optimal discriminant function. *IEEE Transactions on Neural Networks*, 1(4):296–298, 1990. ISSN: 1045-9227. DOI: 10.1109/72.80266.

[25] Sara Sabour, Nicholas Frosst, and Geoffrey E. Hinton. Dynamic routing between capsules. *CoRR*, abs/1710.09829, 2017. arXiv: 1710.09829. URL: http://arxiv.org/abs/1710.09829 (visited on 08/12/2018).

[26] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010. ISBN: 0131387685, 9780131387683.

[27] Conrad Sanderson. Armadillo c++ linear algebra library, June 2016. DOI: 10.5281/zenodo.55251. URL: https://doi.org/10.5281/zenodo.55251 (visited on 08/12/2018).

[28] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013. ISBN: 0321563840, 9780321563842.

[29] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015. URL: `http://arxiv.org/abs/1409.4842` (visited on 08/12/2018).

[30] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004. ISBN: 0131405632.