Building a Neural Network Library From Scratch using General-Purpose GPU Programming with CUDA C

Final Project Proposal

CSC-213: Operating Systems & Parallel Algorithms

Anh Vu

I. Introduction

i) Inspiration & Rationale

Neural networks (NN) has for long been one of my biggest topics of interest. Through taking courses like *STA-395: Introduction to Machine Learning*, I have been familiarized with the building blocks of NNs and the underlying mathematical concepts. Yet, my interactions with using NNs have so far been abstracted away by libraries like *PyTorch* and *Scikit-learn*, for better or worse.

Going into my latest summer internship, I got the chance to work in the ML realm again, now in the MLOps context. Although the experience gave me a new and profound appreciation for building efficiently automated pipelines for data engineering, model training and model deployment, it involved building such an architecture around a more-or-less black-box model pre-trained by a cloud service provider. The frustration came with the guesswork around how the model learns and performs, and the limited venues for improvement aside from hyperparameter tuning and data preparation.

Ultimately, this fueled my desire to build a NN from scratch, hoping to understand thoroughly and observe how NNs work under the hood. In building a NN, one of the most relevant concerns is speed, as NNs can contain tens of thousands of nodes, with millions of edges each requiring a mathematical operation. This performance problem aligns nicely with the topic of parallel algorithms introduced in *CSC-213* and gives rise to this project.

ii) Project Description

On a high level, the project involves creating a library for working with neural networks accelerated with general-purpose computing on graphics processing units (GPGPU). A user would be able to use API calls exposed by the library to create neural networks of a desired architecture, perform common operations for model training, and save/load their models for future use. Under the hood, the library would utilize GPUs to perform several parallelizable tasks such as node-level mathematical computations and automated hyperparameter tuning.

II. Integrated Concepts

i) Concept 1: Parallelism with GPUs

There are several parallelizable tasks using GPUs in the context of NNs.

Firstly, for any edge between nodes in the network, there is an associated series of mathematical operations that typically involves a linear combination of the weights on that edge and the previous nodes' values, followed by an application of an activation function. Each individual edge or node does not rely on one another, allowing them to be delegated to different threads on the GPU.

Secondly, a hyperparameter tuning job simply involves exhaustively trying out combinations of hyperparameters in a user-specified search space. This effectively spawns multiple unrelated versions of a model, which again can be handled by separate GPU blocks or threads.

ii) Concept 2: Thread Synchronization

Using parallel GPU blocks and threads naturally calls for synchronization.

When computing the intermediate values of nodes in a hidden layer in parallel threads, we need a way to determine when all threads have finished. We then need to synchronize the threads, and continue calculations for the next layer. A synchronization, therefore, needs to occur at every hidden layer of the network.

In hyperparameter tuning, we need to delegate different combinations of hyperparameters to parallel blocks or threads, and synchronously identify the combination that yields the best accuracy after all tuning jobs have completed.

iii) Concept 3: File and File Systems

There are multiple benefits to being able to save and load a model, including portability, reusability, and transfer learning. Achieving such a feature will likely call for devising an encoding scheme and a reasonable file format such that models' weights can be saved easily and loaded correctly.

III. Implementation

i) Library API

The following are high-level functions exposed to the user by the library's API:

a. `create_network(architecture, hyperparameters)`
Creates a network with a specified architecture and set of hyperparameters. There should also be helper functions to create nodes, hidden layers, initialize weights and biases, and set the activation functions for each layer.

b. `forward_propagate(network, use_gpu)`
Trains a network by propagating the input data forward until it reaches the final layer. Uses a flag that determines whether the GPU is to be used.

c. `back_propagate(network, use_gpu)`
Updates the weights and biases of the network through a process of calculating the gradient of the loss with respect to each of the weights and biases.

d. `train_network(network, use_gpu)`
Uses forward- and back-propagation to train the network through its set epochs.

e & f. `save_network(network, file)` & `load_network(network, file)`
Saves or loads a network's weights and biases to or from a file.

g. `get_inference(network, input_data)`
Get inferences from a trained network using some set of input data.

ii) Timeline

Week 1 (11/20–11/26): Thorough planning, research, and test-driven draft implementations of individual atomic components of the library.
Week 2 (11/27–12/03): Extensive implementation of the library, debugging of major issues, planning for the next sprint, and completion of some library components.
Week 3 (12/04–12/10): Completion of all library components, testing with example dataset, analysis of runtime, ensuring adherence to project requirements.
Week 4 (12/05–12/15): Touch-ups and preparation for in-class presentation.

iii) Testing & Demonstration

To get a sense of the accuracy of the implementation, I plan to use the popular MNIST dataset of handwritten digits to train a classification model and compare its performance to example neural networks trained on it publicly available online. Moreover, an interesting question to explore is how much time is actually saved by using the GPU instead of the CPU. I wish to answer this question through testing and present my findings in class, along with a live demonstration involving training a model on a dataset and evaluating its inferences if time allows.

V. Possible Complications & Solutions

In terms of the three integrated concepts, I anticipate that complications should only arise due to my lack of understanding of the domain since they are already commonly used methods. Were extra efforts to research and follow example implementations to fail to unblock development, I would alter my project to focus on one small and simple NN architecture instead of providing a general solution.

One of the most concerning problems with my implementation is with floating point precision. Completely inexperienced in this domain, I will study relevant mathematical equations thoroughly, and rely on online blogs and papers for guidance on proven techniques to minimize precision errors. In the worst-case scenario, I would have to settle on a low-precision implementation of the library.

Another major concern is the time constraint. I will frequently evaluate the progress of the project in regards to the remaining time to ensure project completion. If major time constraints arise, I would discard the less important parallelizable task that is hyperparameter tuning, and also of other nice-to-have components of the project such as the comparison between CPU and GPU performance.

One last minor logistical concern is that I do not have easy access to a machine that has GPUs. This might require frequent visits to the MathLAN machines or DASIL.

Should other complications arise, I will further consult with Professor Curtsinger for advice and guidance.