Part One: Framework Implementation Details

Luc Alexander

September 11th, 2023

Contents

1 Abstract

The software implementation described in this document constitutes the first of three major project sections for a Neural Architecture Search (NAS) system. This first portion is comprised of a neural network framework with a rigid implementation specialized for flexible use in the later project sections. It will be abstracted into a user level module for ease of use and general applicability.

This user level abstraction will be documented.

2 Feature overview

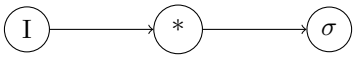
The first part of this software project will consist of a specialized machine learning development framework that allows the maleable design of a neural network through the view of a directed nodal graph. Within this graph, nodes may constitute a layer, a branch

of information flow, or a convergence of information flow. All information flow must converge to the main branch before reaching the necessarily singular terminal node. Branches may merge ahead of their own node, or behind it to create a memory link, or gated memory unit. Branching nodes exist between all layer nodes regardless of a branch, and hold information about the connection between involved layers. Information flows in the direction of the directed edges.

This framework will include a structural system, and basic simulation features as they are required by the overall project. The structure of a network will be mutable by the user. Users will be able to change the shape of the graph dynamically and tune the hyperparameters. After solidifying a structure they will be able to load data sets and train the model with them.

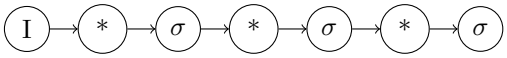
2.1 Structure

The most basic network graph requires three distinct nodes.



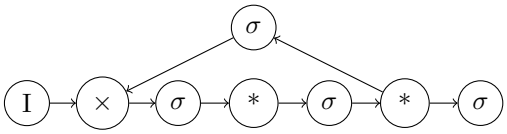
The input node I is where input data is loaded for any given pass of the network. The output node σ is a standard layer node, like any hidden layer, but is marked as the terminal node. σ indicates a sigmoid activation function on this layer node. This is where the loss function will be computed and where backpropogation will begin. The intermediate node, $*$, is an information flow node. These nodes must exist between any two layer nodes, and also serve as the node responsible for splitting branches.

Any number of hidden layers may be included between the input and output nodes, of course separated by information flow nodes.

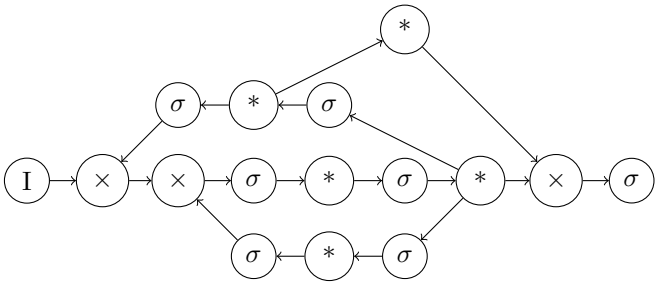


Information flow nodes may branch indefinitely, and those branches must converge to another branch eventually. All branches must converge with the main branch leading to the output node. Convergence points require an information flow node, specialized to

specify the convergence operation. $\times$ denotes point-wise multuplication for the vectors involved.



Convergence information flow nodes can only converge two branches, however information flow nodes can be infinitely chained, so you may converge infinitely many times between two layers.



2.1.1 Nodes

Nodes will operate through a state machine where an enumerated state determines the type and therefore the behavior and mutable parameters. All nodes have a pointer to at least the next node and the previous node in the graph along their personal main branch.

Input

The input node has a buffer for holding the initial data for a pass.

Information Flow

Information flow nodes contain a pointer to the previous data buffer, whether it may be from the input node, the most recent hidden layer in the chain, or the most recent convergence node buffer. There are two variations of this node type.

Divergent

Divergent nodes contain additional next-node pointers to keep track of all branches.

Convergent

Convergent nodes contain a single additional previous data buffer pointer, as well as an additional previous-node pointer to keep track of the branch which is being merged with the current branch defined by the original previous-node pointer. They also contain a pointer to a function with which to merge the data of two branches, as well as an additional data buffer to store the merged vector during a pass. This buffer can be referenced by other nodes.

Layer

Layer nodes have a referenceable data buffer just like the input node, but have additional buffers for weights and biases. They also have a pointer to an activation function.

Layer nodes can be specialized as the output node, which has an additional function pointer denoting the loss function, and must have an empty next-node pointer.

2.2 Simulation

Training a model on a loaded dataset involves running ϵ epochs. An epoch involves one complete forward and backward pass of all training samples, divided into batches of size β. Any single batch has all of its samples predicted and their losses computed. The average of these losses is used to perform gradient descent. The gradient of the loss is back propogated using Back Propogation Through Time (BPTT), and the resulting computed values are used to update the weights and biases of the network using optimization algorithm ω.

The training and evaluation of the function defined by these directed graphs is highly parallelizable. The computation of data buffers in layer nodes can be vectorized, and the computation of information flow in branches can be multithreaded.

Before any simulation begins, weights and biases must be initialzied using an initializer function ι.

Hyperparameters like ω and ι require a function implementations, such as activation functions, loss functions, and learnable param-

eter initialization functions. So these will be implemented as part of this framework.

2.2.1 Passes

Data buffers in layers are computed using the data buffes the layer contains, as well as the pointer to the previous relevant data buffer accessible through the previous node, which is necessarily an information flow node. Convergence nodes have their merged buffers computed based on the previous buffer pointer from the current branch, and the buffer pointer from the last sample pass. After the computation of a buffer, and present activation function or loss function is applied before the next node is calculated. A node can only be calculated if all previous nodes are ready, as this computation process must be parallelized to maximize efficiency. Each buffer calculation can be vectorizied for the same reason.

2.2.2 Optimization

After a batch is passed through the network during an epoch, the losses of each sample are averaged and the loss gradient is computed. This value is used to update te learnable parameteres of the model. Since branching can move back in time to create recurrant and gated memory functionality, BPTT must be used. Once back propogation ends, the chosen optimization function of the model uses the resulting computation to update the weights and biases to minimize the loss function by the learning rate λ.

2.3 Serialization

Models can be serialized, and loaded in order to save models with their trained learnable parameters. I may make it possible in the future to serialize or convert models to formats usable in other popular machine learning frameworks, such as tensorflow keras, or pytorch.

3 Tools

3.1 Framework Language

The framework itself will be implemented in C in order to ensure maximum control over performance. While most high leevel machine learning and data science is performed in Python, Python has a global interpreter lock (GIL) which disallows true multithreading

capabilities opting instead for multiprocessing, which while beneficial is not completely optimal. Python also has no way to do idiomatic vectorization of operations on functors. Any module that gives access to this feature, such as numpy, is implemented in a language lacking a GIL, usually C. The overall project will be implemented in a mixture of C for high performant operations such as this framework, and Python for user level ease of use.

Another consideration was made to use Chris Lattner's Mojo programming language, a superset of Python which allows for C like performance by allowing specialized Python code to take advantage of vectorization, memory tiling, static types, etc. However as of the date of this software's conception, this tool is not yet feature complete, and does not yet serve as a true representation of its final vision.

3.2 Product Language

The framework will be usable as a python module for future segments of the project, necessitating the usage of the Python C API. Documentation for this user level abstraction will be done in markdown.

3.3 Development Ecosystem

The entire scope of the project this document belongs to will be split among two Git repositories, one for developing and maintaining the C implemented framework, and one for the NAS. Markdown documentation will be included in each respective git repository's README.md document. All project development documents including implementation outlines, project segment reports, final thesis, documentation, and any extraneous documents written in relation to the development, testing, or usage of this software will be stored on a website. This website is yet to be developed, and will be developed concurrently to the neural network framework.

All repositories are yet to be created. The repository relevant to this segment of the overall project, as well as the website will be included in a later document.