Framework Implementation Progress Report 0

Luc Alexander

September 28th, 2023

Contents

1 Abstract

The structural component of this software framework has been completed. This document includes descriptions of all implemented systems up to this point, including the graph structure, the compiler for the descriptino language which generates graph models, and both linear and limited vectorized versions of a sample of functions required for the neural network to run.

The included repository includes a brief TODO list, and the entire code base. This software is a work in progress. Upon the completion of this document will begin the next development phase,

wherein the functionality of the graph neural network will be implemented including forward passes, batches, epochs, gradient descent, and back propogation through time. The python user level module interface will also be completed. After this second development effort, documentation will be drafted to replace the current README.md file in the repository.

The next development phase will also include more work into optimizations, such as adding more SIMD architecture candidates for better vectorization where it is supported.

2 Model Description Language

The model description language (MDL) is a syntax for describing graph based neural networks. It is designed in a way which makes the structures in question easy to programatically mutate from one legal graph to another.

2.1 Language Structures

2.1.1 Layers

Layers are enclosed with parenthesis ( ), the first argument is always the name of this layer as it relates to the rest of the model. This is a unique identifier. The first argument of a layer, the layer width, is an integral order dependant parameter. The layer width controls how many nodes are in the layer which this node represents.

Layer
A typical layer expects a single other position independent argument; a parametric activation function.

Output
The output layer requires one extra position independent argument in addition to an activation function; a parametric loss function.

Input
The input node can be given an activation or loss function, but it requires neither, and neither argument being passed to it will have any effect on the behavior of the node or the model.

Parametric Function
The activation and loss functions in question are parametric function expressions. These are denoted by angle brackets <>, and require at least one position dependent argument; the name of the function. This is not a unique identifier and refers to a preexisting function within the framework which is categorized as either activation or loss. Each expression may be passed one optional floating point argument which controls the parameter of the function if applicable. Some activation or loss functions require an additional hyperparameter to work. If the given function requires one and it is not provided, 0 is assumed. If a function does not require one, yet one is provided, the argument is ignored.

2.1.2 Links

While links of some kind are required to exist between each layer in the output graph, they are not required elements in an MDL description. If two layers are found adjacent in the description, a divergent link with no extra paths will be inserted to maintain legality.

Divergence
Divergence nodes are denoted by square brackets [ ], and also require a unique identifier as their first argument. Subsequent arguments are separated by a pipe operator — as opposed to a comma for standard arguments. Each pipe separated expression represents a branch, and must be its own valid MDL description. No branches are required, and while the second argument of a divergence node is required, it may be left entirely empty.

Convergence
Convergence nodes are denoted by curly braces **{ }**, and similarly require a unique identifier as their first argument. They require two subsequent order dependent parameters. The first is the name of a unique identifier of another node. This identifier may refer to a layer or link of any kind, even another convergence. The final arugment is a non-parametric function name, which refers to preexisting convergence functions present in the framework.

2.1.3 Examples

A Feedforward Neural Network

```
(input, 768)
```

```
(L1, 1024, <sigmoid>)
(L2, 1024, <relu_parametric, 0.6>)
(L3, 1024, <softmax>)
(output, 2, <elu, 5>, <huber, 1.1>)
```

A Recurrant Neural Network

```
(input, 128)
{merger, link, average}
(a, 128, <relu>)
(b, 128, <swish, 5.9>)
[link,]
(output, 2, <sigmoid>, <mse>)
```

A Memory Gated Recurrant Network

```
(input, 128)
{gate, process_remember, multiplicative}
(a, 256, <relu>)
(b, 256, <swish, 5.9>)
[link,(remember, 64, <tanh>)(process_remember, 128, <tanh>)]
(output, 2, <sigmoid>, <mse>)
```

2.2 Compiler and Builder

MDL correctly compiles partway into an abstract syntax graph. This graph can be built into a graph model ready for simulation with a simple function call. When the compiler parses over the description it performs syntax checks. When the builder parses over the abstract syntax graph, it performs legality checks to make sure the output graph makes sense, is complete, and follows the rigid rules of the structure. There must be one input node, one output node, all required arguments must be fullfilled, etc. The compiler ignores whitespace and is built to be lenient, allowing some arbitrary omissions and argument reorderings to allow for comfort.

The user will be able to compile multiple models without having to use an exorbitant amount of memory to store the layers of each structure, only their syntax graphs.

If development continues on this language, I may include more structures such as convolutional layers, transformers, and attention mechanisms.

3 Vector Functions

Preparing for the simulation segment of the frameworks development, I wrote some base functinoality for layer processing and error calculation. The activation, loss, and convergence functions I included are not a complete coverage of the domain, but its close enough. There is much more work that could be done in regard to more complicated convergence functions, and one of the activation functions I was planning on implementing is double parametric which I did not handle the possibility of in the MDL compiler. These may come in the future.

3.1 Activation

Sigmoid

$$f(\vec{x}) = [\frac{1}{1 + e^{-\vec{x}_i}}]$$

Softmax

$$f(\vec{x}) = [\frac{e^{\vec{x}_i}}{\sum e^{\vec{x}_i}}]$$

Tanh

$$f(\vec{x}) = [\tanh(\vec{x}_i)]$$

Binary Step

$$f(\vec{x}) = [\begin{cases} 1 & \text{if } \vec{x}_i \geq 0 \\ 0 & \text{otherwise} \end{cases}]$$

Linear

$$f(\vec{x}) = \vec{x}$$

ReLU

$$f(\vec{x}) = [\max(0, \vec{x}_i)]$$

ReLU Leaky

$$f(\vec{x}) = [\max(0.1\vec{x}_i, \vec{x}_i)]$$

ReLU Parametric

$$f(\vec{x}, \alpha) = [\max(\alpha\vec{x}_i, \vec{x}_i)]$$

ELU

$$f(\vec{x}, \alpha) = [\begin{cases} \alpha \times (e^{-\vec{x}_i} - 1) & \text{if } \vec{x}_i < 0 \\ \vec{x}_i & \text{otherwise} \end{cases}]$$

GeLU

$$f(\vec{x},\gamma) = [\frac{1}{2}\vec{x}_i(1 - \tanh(\sqrt{\frac{2}{\pi}}(\vec{x_i} + (\gamma\vec{x}_i^3))))]$$

Swish

$$f(\vec{x}) = [\frac{\vec{x}_i}{1 + e^{-\vec{x}_i}}]$$

3.2 Loss

Mean Squared Error

$$L(\vec{y},\vec{x}) = \sum (\vec{x}_i - \vec{y}_i)^2$$

Mean Absolute Error

$$L(\vec{y},\vec{x}) = \sum |\vec{x}_i - \vec{y}_i|$$

Mean Absolute Percentage Error

$$L(\vec{y},\vec{x}) = \sum |\frac{\vec{x}_i - \vec{y}_i}{\vec{x}_i}|$$

Huber

$$L(\vec{y},\vec{x},\alpha) = \sum \begin{cases} \frac{(\vec{x}_i - \vec{y}_i)^2}{2} & \text{if } |\vec{x}_i - \vec{y}_i| <= \alpha \\ \alpha|\vec{x}_i - \vec{y}_i| - \frac{\alpha^2}{2} & \text{otherwise} \end{cases}$$

Huber Modified

$$L(\vec{y},\vec{x}) = \sum \begin{cases} \max(0, 1 - (\vec{x}_i - \vec{y}_i)^2 & \text{if } \vec{x}_i - \vec{y}_i > -1 \\ -4(\vec{x}_i - \vec{y}_i) & \text{otherwise} \end{cases}$$

Cross Entropy

$$L(\vec{y},\vec{x}) = -\sum \vec{x}_i \ln(\vec{x}_i - \vec{y}_i)$$

Hinge

$$L(\vec{y},\vec{x}) = \max(0, 1 - (\vec{x} \times \vec{y}))$$

3.3 Convergence

Additive

$$f(\vec{x},\vec{y}) = [\vec{x}_i + \vec{y}_i]$$

Multiplicative

$$f(\vec{x},\vec{y}) = [\vec{x}_i \times \vec{y}_i]$$

Average

$$f(x,y) = [\frac{\vec{x}_i + \vec{y}_i}{2}]$$

4 SIMD Vectorization

The operations in the vector functions are independent element-wise, making them ideal candidates for vectorization. There are many vectorization architectures under C SIMD. Each processor architecture has many versions of instructions which can be used to optimize the vectorization of any operation of rangeing complexity.

4.1 x86_64

I decided that for the initial development of this project, I would only cover x86_64 processors. The module will still work for other processors, but it will have to use the linear implementation provided. The best available SIMD vectorization implementation is generated during the install.

SSE
Currently the only implementation is of base SSE. SSE has many versions which improve upon the instruction set for vectorization, allowing more optimizations. It is typically also backwards compatible, so a system running SSE4_2 will still benefit from an SSE implementation, even though the code was not written specifically for that version. I plan on supporting optimizations where prossible for processors which support the following versions of SSE.

- SSE
- SSE2
- SSE3
- SSSE3
- SSE4_1
- SSE4_2

AVX
SSE is the old standard, and while most x86 processors support it, they usually also have a version of the more modern AVX, or the even more recent AVX512. I plan on implementing support for these SIMD candidates, but this is not entirely urgent. I may do this after the completion of the larger project when I revisit the module, or if I find that I have spare time during development.

4.2 Other Architectures

Many of the target users for a machine learning tool do not use x86 processors. I plan on implementing some version of all of the

following SIMD instruction sets, but this is also not urgent for a proof of concept implementation, especially when the main focus of this thesis is larger than this framework.

ARM with NEON
Apple M1 and M2 processors have switched from x86 to ARM. SIMD provides support for these new processors with the NEON instruction set.

RISC-V
While not as common now, RISC-V is gaining popularity.

CUDA
Nvidia GPUs are a large portion of the hardware used to run machine learning tasks. Supporting CUDA is essential for a neural network framework of any kind.

5 Development Ecosystem

This framework as well as the greater project of this thesis are managed through public Git repositories hosted on Github. The repository for the framework described in this document can be found at

`https://github.com/LucAlexander/NeuroMorph`

Markdown documentation will be displayed here as well.