Framework Implementation Progress Report 1

Luc Alexander

November 5th, 2023

Contents

1 Abstract

This section of the project finishes the base functionality of the framework required to finish the exploration portion of the neural architecture search system. We can now compile intricate graphic models which can perform batches of forward passes, followed by optimizing backward passes to update learnable parameters in any model which is legal to the description language.

The differential supergraph union, required for the exploitation technique, has not been implemented yet.

2 Initialization

Before a compiled model can be used, its default learnable parameters must be initialized. To initialize weights the following known functions have been implemented:

Xavier (Glorot)
$w \sim U\left(-\sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}}+n_{\text{out}}}}\right)$

He
$w \sim U\left(-\sqrt{\frac{6}{n_{\text{in}}}}, \sqrt{\frac{6}{n_{\text{in}}}}\right)$

Lecun
$w \sim U\left(-\sqrt{\frac{3}{n_{\text{in}}}}, \sqrt{\frac{3}{n_{\text{in}}}}\right)$

Uniform
$w \sim U\left(a, b\right)$

Normal
$w \sim \mathcal{N}\left(\mu, \sigma^2\right)$

To initialize biases, the following known functions have been implemented:

Zero
$b \sim 0$

Constant Flat
$b \sim a$

Constant Uneven
$b \sim \mathcal{N}\left(a, b\right)$

The model description language compiler has been updated to support a header which defines how the learnable parameters will be initialized, supporting all of the implemented functions. The syntax for this header is parsed before the rest of the model. The syntax is

$/name_{weight}$,$name_{bias}$ $param_1$ $param_2$/

This can be seen in some of the example models used for testing.

lstm-model

```
/xavier,const_flat 0.5/
(input, 4)
{a, lastrecur, additive}
[b,
    (include, 4, <sigmoid>)
```

```
|
    (process, 4, <tanh>)
    {pi, include, multiplicative}
|
    (add, 4, <sigmoid>)
]
(forget, 4, <sigmoid>)
{state0, prevstaterecur, multiplicative}
{state1, pi, additive}
[prevstate,[prevstaterecur,]]
(statep, 4, <tanh>)
{lastc, add, multiplicative}
[last,[lastrecur,]]
(output, 4, <tanh>, <mse>)
```

big-model

```

/xavier,const_uneven 0.1 0.3/


(input, 4)
(a, 4, <sigmoid>)
{c2, g, additive}
(b, 4, <relu>)
[b1,
    (d, 4, <softmax>)
    {standby, stalerecur, additive}
    (e, 4, <softmax>)
|
    (f, 4, <relu>)
    [stale,[stalerecur,]]
    (g, 4, <relu>)
]
(c, 4, <sigmoid>)
{c1, e, additive}
(output, 4, <sigmoid>, <huber_modified, 2.4>)
```

3 Forward Passes

3.1 calculation

Given X is the matrix of inputs for a layer, W is the weight matrix of the current layer, b is the bias vector of the current layer, and f is the activation function fo the current layer; for a contiguous

array of layers a forward pass is trivial and can be treated as a feed forward network. We derive the actiivated output vector of the layer as

$$A = f(XW + b)$$

For convergence nodes, there is no activation or forward propogation other than the convergence operation c of the two independent input vectors X, and Z as

$$A = c(X, Z)$$

For divergence nodes, no additional operation occurs, but the activated output is propogated as the input of each divergent branch.

For the output node, the standard pass calculation has an extra step for loss calculation $L(expected)$ as

$$L = l(E) \circ f(XW + b))$$

Convergence from a looping layer cannot occur for the first pass of a batch, but occurs for every other pass. This is the only nuance posed by memory systems for forward passes.

The forward pass was very easy to multithread, allowing me to take full advantage of the computers resources to calculate the result of a model given an input with maximum concurrency.

3.2 Batches

The preactivated outputs, activated outputs, model loss outputs, model inputs, and expected outputs are all recorded per pass per batch. A batch of data will be run consecutively without intermediate backward passes. One backward pass will run at the end of each batch using all collected state data from the entire batch of forward passes.

4 Backward Passes

4.1 Functions

For every corresponding function in a forward pass which in any way mutates the forward flow of data vectors from input to output, we must take the partial derivative with respect to the loss so that we can calculate the n dimensional gradients of each learnable parameter we want to update. For most functions this is trivial, just integrate with respect to loss and propogate the chain rule backwards for the calculation of functions further back in the network until you reach the input node. This includes activation functions, loss functions, and convergence functions. There is one exception, which is stepwise functions such as binary step. These functions have undefined derivatives, so using them will prompt a warning that they shouldn't be used with a gradient optimizer.

4.2 Gradient Optimizer

The minimum viable product implementation only includes a default gradient optimizer, although eventually an Adam or RMSprop option would be optimal. Gradient optimization is usually split upon two strategies, Stochastic Gradient Descent(SGD) and Gradient Descent. The only real distinction between these two is how often the backward pass runs, whereing SGD there is a backward pass for every forward pass, although all this means for our structure is that the batch size for that model is one. So this implementation covers both concepts.

4.3 BPTT

Back Propogation Through Time (BPTT) involves unrolling memory links in a network upon which we can perform normal back propogation of gradients using our gradient optimizer. When we encounter a convergence point, we distribute the gradient to both branches, one of the branches being a memory loop means that this convergence point will be reached again during this pass, so we mark it to make sure that it only gets propogated through once. At divergent points, we must wait for all non loop divergent branches to finish their propogation before we aggregate them all so that the aggregatd gradients can be used in the next layers calculation. Looping branches at divergent points will naturally propogate through when they are calculated starting from their

respective convergent point.

The calculation itself involves calculating the derivative of any function involved in data flow mutation with respect to the loss (propogated to the layer in question through the chain rule of differential calculus) for every pass in the batch, these gradients are averaged together for a final gradient for every weight and bias, which multiplied by a predefined model learning rate allows loss gradient descension.

The backward pass with BPTT has also been multithreaded for each branch calculation.

5 User Level Module

The user level module in python can now be used to compile and train models.

5.1 Python

First import the module.

```
import neuromorph as nm
```

models can then be generated from any string MDL representation. We can also change the seed used for learning parameter initialization.

```
batch_size = 5
learning_rate = 0.001

model = nm.compile(
    '''
    /xavier,zero/
    (input, 128)
    (interm, 256, <tanh>
    (output, 4, <sigmoid>, <mse>)
    ''',
    batch_size,
    learning_rate
)

# Assuming lstm-model exists in the current
    directory
with open("lstm-model", "r") as infile:
    loaded = nm.compile(infile.read(),
        batch_size,
```

```
        learning_rate
    )

nm.seed(344387)
```

We can then build any models we have compiled, which will transfer the intermediate representation to a solidified graph representation, and initialize weights and biases. This also returns the ID of the model you passed.

```
nm.build(model)
```

Any built model can be safely trained by passing it a set of input vectors in the shape (samples, batch_size, model_input_length), and a set of expected output vectors in the shape (sample, batch_size, model_output_length)

```
samples = 100
batch_size = 5
input_size = 128
output_size = 4
generate_tensor = (lambda vector_size: [
    [
    [random.random() for i in range(vector_size)]
    for batch in range(batch_size)
    ]
    for k in range(samples)
]
)
input_data = generate_tensor(input_size)
expected_data = generate_tensor(output_size)

nm.train(
    model,
    input_data,
    expected_data,
    verbosity
)
```

Optionally you can release the associated model memory to prevent memory leaks

```
nm.release(model)
nm.release(loaded)
```

5.2 Documentation

Minimal Documentation for this module is available in the git repositories README.md.

6 Refactoring and Optimization

Locality
It would be possible to restructure the memory that holds the neuron buffers, weight matricies, and bias vectors in contiguous memory locations to maximize cache usage. This would happen after a model is compiled and built, and would still allow for vectorization and multithreading. This would be an optimization which sacrifices nothing. At this stage we could also implement memory tiling.

Threading
While I have made sure everything is which is currently multithreaded with posix pthreads has no concurrency issues, there are other oportunities (particularly in regard to backward propogation) where additional threading can be implemented.

SIMD
Functions related to the forward pass all have at least SSE SIMD targets, but no partial derivatives, nor the backward propogation calculation functions are vectorized yet. It would be advisable to fill this gap, and then further support AVX, CUDA, NEON, and RISC-V SIMD instruction sets.

Layer Dimensions
Currently there is a limitation imposed by the structure that any layers feeding into and coming out of any branching nodes, convergence or divergence, must have the same vector dimension. There are ways to lift this restriction, but they require more involved methods of aggregating and splittin gradients which did not fit within the scope of this project.

Alternate Optimizers
Ideally, Adam and RMSprop should be implemented for any useful machine learning backend. If development continues after the minimum viable product is finished, this is an essential step.

Multiparametric Support
Currently there is no support for activation functions which are multiparametric, simply because I did not think ahead

when writing the compiler. This would be a simple refactor, but is not necessary for the scope of this project.

Exceptions
Currently any C level error leads to fprintf(stderr, const char*) and not much else. In the future, proper pythonic errors and exceptions should follow.

Additionaly Module Functionality
Serializing trained models, loaded serialized models, and running models to get output as opposed to training has yet to be implemented.

7 Development Ecosystem

This framework as well as the greater project of this capstone project 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 is available here as well.