A4M39GPU, Deep Learning Framework

Adam Volný

January 12, 2018

1 Problem Statement

The goal of this project is to create a basic deep learning framework with high level of modularity and generality. Its interface should be intuitive and efficient. It should encapsulate both CPU and GPU implementation in a meaningful manner so that either can be used. The performance of both implementations will later be tested against state-of-the-art frameworks.

First, let's introduce some basic theory on neural networks.

1.1 Artificial Neuron

An artificial neuron is a function which may have several inputs and a single output. The input vector $\mathbf{x}$ is fed to the *body* of the neuron through weight vector $\mathbf{w}$, we call that the potential (I call it activation in my code),

$$\zeta = \mathbf{w}^T\mathbf{x} + b$$

where b is a scalar bias. The potential is then fed through a single-real valued function, which is called activation function and the output of neuron is obtained as:

$$y = f(\zeta)$$

where the activation function can be of many kinds, it can be linear function, logistic sigmoid, *tanh*, or the recently used rectified linear unit (ReLU). Most of the time, nonlinearity is preferred as it allows us to meaningfully stack multiple layers of artificial neurons on top of each other.

1.2 Artificial Neural Networks

We can use artificial neurons to form columns called layers and then create multiple layers which are interconnected to create one large network consisting of many layers of neurons.

When the weights and biases in the network are set accordingly, an ANN can be used as an universal function approximator. It can be used for classification, regression and some other specific tasks. The critical question though is, how to tune those parameters in such a way that the resulting network indeed does

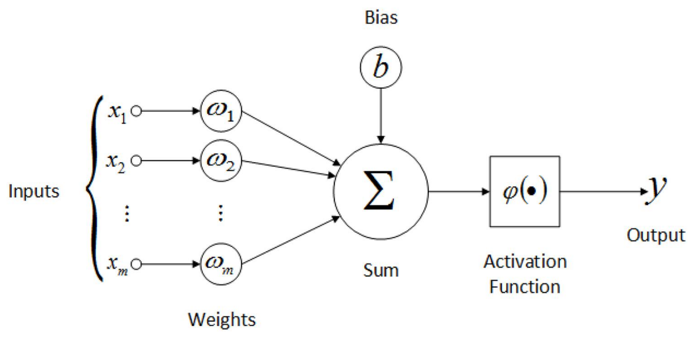

Figure 1: An artificial neuron

what we want it to do. To achieve this goal, a very straightforward algorithm was developed back in the 80s called the backpropagation algorithm.

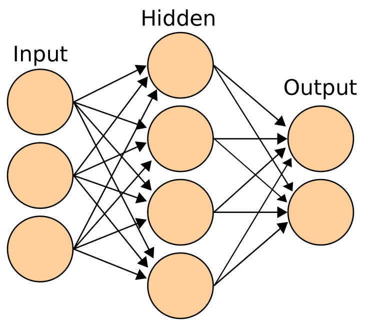

Figure 2: An artificial neural network

The whole idea is very simple. Let's consider a supervised case, where we have a set of inputs for the network and a set of correct outputs for each of those inputs. Now we have to define some measure of quality of our network. That's called *loss function*. For this task we can consider e.g. mean square error between the actual outputs of the network given the input and the correct output. Now we have a scalar value, which we are trying to minimize, and the backpropagation algorithm does this by iteratively updating each parameter a little bit according to the contribution of each parameter to the error. The contribution can be computed as partial derivative of the loss function w.r.t. the parameter. Let's look at it closer.

1.3 Backpropagating the error

Let $\mathbf{y}_o$ be the output of the network, let $\mathbf{t}$ be the desired output, then error can be written as,

$$E = \frac{1}{2}(\mathbf{t} - \mathbf{y}_o)^2$$

now let $\boldsymbol{\theta}$ be a vector of the network's parameters. The problem is that it is intractable to solve the minimization of E analytically, so we try to do so using the gradient descent algorithm. The idea is that we start with some initialization of parameters $\boldsymbol{\theta}$ and then we try to make little updates that try to lower the error. Problem with this method is that it gets stuck in local minima and has very low chance of finding the global optimum so in practice more sophisticated algorithms (e.g. using momentum) are used for the error minimization.

The update rule can be written as follows,

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \epsilon \frac{\partial E}{\partial \boldsymbol{\theta}}$$

where ϵ is a small real number called learning rate. Essentially, we are approximating the surface of the error function at point $\boldsymbol{\theta}$ by an affine function and then we take a small step in the direction of the steepest descent on this affine function. The assumption is that the error function's surface is *well behaved* in a small neighborhood around this point, by which I mean that steepest descent on the affine function that is approximating the error surface at the point $\boldsymbol{\theta}$ is most likely going to be close enough to the steepest descent on the actual error surface. Therefore, as long as the assumption holds, minimizing the error function itself.

For a single component θ_i it is usually non-trivial to calculate the derivative $\frac{\partial E}{\partial \theta_i}$ and it requires some diligence to do it properly. Probably the easiest way to derive the formula for given weight is using the chain rule. Let's assume the network consists of a single column of layers. Then we can decompose the error derivative for parameter θ_i in layer j as follows:

$$\frac{\partial E}{\partial \theta_i} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial \theta_i}$$

where θ_k is one of the parameters within layer j. The second part $\frac{\partial y_j}{\partial \theta_k}$ is specific to the given layer type and tells us how is the parameter θ_k influencing the output of layer j, so this formula is going to be the same for all the layers of the same type. What remains is computing the first part. For the output layer that's quite straightforward, that would be just,

$$\frac{\partial E}{\partial \mathbf{y}_o} = \mathbf{t} - \mathbf{y}_o$$

and for every intermediate layer, this can be decomposed again as,

$$\frac{\partial E}{\partial \mathbf{y}_j} = \frac{\partial E}{\partial \mathbf{y}_i} \frac{\partial \mathbf{y}_i}{\partial \mathbf{y}_j}$$

where layer j is predecessor of layer i, once again the derivative $\frac{\partial \mathbf{y}_i}{\partial \mathbf{y}_j}$ is specific to the given layer and can be computed easily, then when computing the derivative of layer before j, we just pass the computed value of $\frac{\partial E}{\partial \mathbf{y}_j}$ to it. So we propagate the error signal backwards, from the output to the input, that's why it's called backpropagation.

2 Implementation

The reason why the GPU implementations of deep learning frameworks are so widely used is simple, they are fast. Usually much faster than the CPU implementations. In the above theoretical introduction we consider single input vector and single output vector but it turns out that it is much better to compute the gradients in batches of samples because that drastically reduces the variance and leads to a faster and more stable convergence. So instead of single vector let's say we compute the output and gradient for 256 vectors. Now the input vector becomes a matrix, as does the output vector and all the intermediate calculations for the forward and backward pass become essentially different matrix calculations. And matrix calculations can be easily parallelized.

The central data structure is *tensor*. It is simply a generalization of vector and matrix. So it is an n-dimensional array with regular shape.

2.1 CPU implementation

The CPU implementation serves mainly as a sanity check for the whole project, the point of it is not to be as fast and as optimized as possible but to confirm that my derived math was indeed correct and that the structure of the code is working properly. However, there would be a straightforward way to optimize the code. When I was coming up with the API design, I got my inspiration from the python scientific computing library called NumPy. Its tensor implementation is very powerful thanks to the fact that you don't have to write a single *for* loop and you can in fact do all the mathematical operations using vectorized functions upon the tensors which are implemented efficiently in C in the library backend.

Therefore the tensor structure in this project works in a very similar way and all the mathematical transformations that are performed on the data during computing output and gradients is done using vectorized operations. Therefore, if the tensor functions were written more efficiently, e.g. using more than single thread or using SSE instructions, there could be a significant speedup of the CPU code.

The fact that all the operations are vectorized makes implementing the math a much less daunting task as when one's sure that the tensor implementation is working correctly, the only mistake can be in the math. However if all the operations were done individually, the code would drown in *for* loops and it would be very hard to both read and debug.

```
// get the output of previous layer
tensor_ptr in = (tensor_ptr) parents[0]->get_output();
// compute the dot product between the input and weight matrix
activation_tensor = weight_tensor->dot(in);
// add bias tensor which is repeated for each batch sample
activation_tensor->add(bias_tensor->repeat(1, batch_size));
// feed the resulting activation to the activation function
activation->compute_activity(activation_tensor);
```

Code snippet 1: Tensor interface example, computation of output of a hidden layer

One of the main ideas was to setup the API structure in such a way, that it could be reused by the GPU code. That means that the CPU/GPU distinction is abstracted away from the user of the framework who only has to specify which device to use during initialization but the usage of the framework doesn't change when using GPU instead of CPU.

2.2 GPU Implementation

Once I got the CPU implementation up and running, I was confident enough to proceed to designing the GPU code. My intention was to make the GPU code optimized, unlike the CPU code. In this case I actually didn't go for the vectorized operations for a simple reason; lot of the mathematical operations that are chained together in single layer could be performed upon shared memory in one go, omitting the need for redundant reads and writes to the global memory. But I have written some vectorized functions for tensors on GPU too because they proven useful in some cases (e.g. computing mean of a tensor over given dimension).

Another central idea of the framework is that when using GPU, all of the computations happens on GPU. The only transfer between the host and the device is when transferring the input to neural network or when transferring the output back to RAM. So the existing structure for handling the computation graph for CPU is reused and instead of calling the CPU operations, kernel wrappers are called. Also 100% of the necessary GPU memory is allocated during the network initialization and then it is completely reused during training. So during the computation itself, no allocation on GPU is happening.

2.2.1 GPU algorithms

There are 2 main algorithms used in the code. Parallel reduction and tiled matrix multiplication. Most of the kernels perform some sort of parallel reduction and occasionally tiled matrix multiplication is done.

For the parallel reduction I used the code from the lectures and modified it slightly so that it can handle arbitrary input sizes, not only powers of two.

For the tiled matrix multiply I used code that we developed during a tutorial. Some of the matrix operations require one of the matrices to be transposed, this was solved by simply loading each tile into shared memory as transposed.

Furthermore, the width and height of each tile is 32 so in order to avoid shared memory bank conflicts, the shared memory is allocated as 32x33. This delivers slight but still noticeable improvement in performance.

```
int prev_len = blockDim.x;
int step = (prev_len + 1) / 2;
while( true ) {
  if(th_ix + step < prev_len && glob_ix < input.shape.len)
  cache[th_ix] += cache[th_ix + step];
  __syncthreads();
  if(step == 1)
    break;
  prev_len = step;
  step = (prev_len + 1) / 2;
}
```

Code snippet 2: Sum using parallel reduction

2.2.2 CUDA C++ Integration

One of the challenges during the development was figuring out, how to integrate CUDA into an existing C++ project structure. The issue is that the CUDA source code cannot be directly accessible from the C++ code as that would mean that the C++ compiler would try to compile it which would result in a failed build. So the CUDA code needs to be hidden away and compiled separately by the NVCC compiler. In order to achieve this, none of the header files contain any CUDA code, all of the CUDA code is located within `.cu` files, where each kernel is accompanied by a wrapper function which calls the kernel. All that the C++ code calls are those wrappers as can be seen in the following code snippet.

```
///// in header file (tensor_gpu.h)
void w_reduce_sum(cu_tensor t, cu_tensor result);

///// in source file (tensor gpu.cu)
// kernel that simply fills a tensor with single value
__global__ void k_tensor_fill(cu_tensor t, float val) {
  int glob_ix = blockIdx.x*blockDim.x + threadIdx.x;
  if(glob_ix < t.shape.len)
  t.array[glob_ix] = val;
}

// wrapper for the kernel above
void w_tensor_fill(cu_tensor t, float val) {
  int gridDim = (t.shape.len + BLOCK_SIZE - 1) / BLOCK_SIZE; //ceil
  k_tensor_fill<<<gridDim,BLOCK_SIZE>>>(t, val);
}
```

Code snippet 3: Simplest example of kernel with its wrapper

2.3 Framework API Components

To see example usage of the framework, please see examples located in `main.cpp`.

2.3.1 Tensor

As mentioned before, the central data structure is tensor, that corresponds to a structure `tensor` for the CPU computations and to a structure `cu_tensor` for the GPU computations. The difference is that the `tensor` holds a pointer to data on RAM as well as contains all the methods for manipulating it, whereas the `cu_tensor` contains only a pointer to GPU memory and the shape of the tensor.

2.3.2 NeuralNet

The main interface for using this framework is class `NeuralNet` which wraps the neural network model and provides methods for building, initialization, training and predicting of the network.

First, layers are added to the network, then it is initialized with an optimizer and then it can be trained using the function `fit`. Later the trained network can be used for prediction using the function `predict`. Also, method `validate_gradients` can be called to validate the analytic gradients, see next section for more details. This is useful when implementing new types of layers to confirm the correctness of the underlying math.

2.3.3 Layer

Next core building block is the abstract class `Layer` which defines the interface for each layer class created for the network. Each layer can be seen as a node in the computation graph, so it has to handle forward computation and backward computation for both CPU and GPU. An important abstract subclass is `LossLayer` which defines the interface for implementing the loss function to be minimized w.r.t. the parameters of the model. It is beneficial to implement this as a layer because it can be treated as such, it also has forward and backward computation and integrates into the whole scheme nicely.

Examples of the vanilla `Layer` implementations are `InputLayer` and `DenseLayer`, where the former is used to feed the network with input data and the latter is the most basic of layers, which presupposes a scheme where two neighboring layers are fully connected with each other (each neuron in layer i is connected to each neuron in layer j).

Implemented subclasses of the `LossLayer` in this project are `MSE` which stands for Mean Squared Error, and `CrossEntropyError`. Both are widely used in deep learning, where MSE is very basic but generally useful, whereas the cross entropy error is more specific and better suited to classification applications.

Activation functions of neurons within layers are implemented separately in the `Activation` class. This is beneficial as there are multiple different types of layers that all can use various activation functions and in general those can be independent from the layer itself, so the forward and gradient computation can be handled separately therefore making the layer implementation simpler and more general. It also provides an interface for trainable parameters of the activation function. For example with logistic sigmoid activation function, it is possible to also train its parameter λ which determines the steepness of the function around zero.

2.3.4 Optimizer

An essential part of each deep learning framework are optimizers. Those are implementations of the optimization algorithms that are used for the parameter updates based on the computed gradients. By design, optimizers in this framework are meant to be completely *layer-agnostic*. The optimizer doesn't need to know anything at all about the layer in which the parameters are located. Each layer simply exposes all of its trainable parameter tensors and also their derivative w.r.t the error function. So each optimizer in each step simply loops over all the layers in the network and over all of its trainable parameter tensors and updates them regardless of what parameter it is.

The abstract class `Optimizer` is parent to two implemented optimizer instances, `SGD` and `RMSProp`. SGD stands for stochastic gradient descent which is the simplest of all gradient descent algorithms (Also doesn't work well in practice). RMSProp is a more complicated algorithm that utilizes momentum. So it maintains a momentum value for each trainable parameter and updates it in each step as well as the parameter itself. It is nowadays widely used in practice thanks to its simplicity and empirical results. For details see [1].

2.4 Gradient validation

One big issue with developing neural networks is that one can have a mistake either in the code or in the math and it can be very hard to tell those apart. Fortunately there is a way to validate whether the computation of the gradient is correct and that would be comparing to numerical gradients.

A derivative of function $f : \mathbb{R} \to \mathbb{R}$ is defined as follows,

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

this can be interpreted as that derivative of function f at point x is a slope of the function at given point. And there is a very simple way to approximate the slope of a function around a given point as follows, let ϵ be a small real number, then,

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Notice that the formula is almost identical to the definition of derivative only we replaced the variable which is supposed to end up in zero by a small real value. So instead of taking 2 points that are infinitely close to define the tangent line, we use 2 points that are finitely close to approximate the tangent. The slope computed this way may or may not be very close to the actual value but this can serve as a sanity check of the gradients computed analytically. Applied to our framework, for each model parameter θ_i we are interested in validating the derivative $\frac{\partial E}{\partial \theta_i}$.

In my experience, $\epsilon = 0.1$ has proven as most accurate in approximating the gradients as with smaller values, the results are prone to errors caused by many consecutive float calculations. The error surface tends to behave nicely so approximating on a larger neighborhood leads to a better rather than worse stability. In the code, I compare the signs of the gradients, when about 90% or more of the analytical gradients match the numerical ones, one can be pretty sure that the math is correct.

3 Testing datasets

3.1 Simple XOR

One of the most simple machine learning non-trivial toy examples is xor function of 2 variables. It is not trivial because it is probably the simplest instance of a dataset which is not linearly separable. Dataset is linearly separable when there exists such a hyperplane that would separate *all* the data points into their correct classes (assuming there are only 2 classes). That is something that you cannot do with XOR, therefore it can be used as a simple test of correctness of the backpropagation algorithm.

A rather amusing observation is that even solving this problem can be quite non-trivial, even when all the calculations and algorithms are correct. The example code `main_xor()` works because it uses the RMSProp optimizer, if it were switched for the basic SGD optimizer, the algorithm would find optimal solution with a very low probability. That is the reason why debugging neural networks can be very challenging. It is not easy to see whether the model is not working because of an error in the implementation or if the model itself is flawed or if the hyper-parameters are set improperly (e.g. the learning rate) or if one is simply unlucky. All of the above can be the case. That is also why when things work as expected that is very reassuring that all is implemented properly because the chances of having a mistake in an implementation and at the same time have it working properly are very low as there are so many things that can go wrong.

The neural network used in this example has 2 input neurons, single hidden layer with 5 hidden neurons and 2 output neurons (one for each class). It uses the ReLU nonlinearity in the hidden layer.

3.2 MNIST Handwritten Digits Dataset

The second test set is a widely used classification dataset which contains approximately 65 000 images of handwritten images with the corresponding labels. Each image is in gray-scale with dimensions 28x28. It is considered to be the *Hello World!* of deep learning, as it is quite straightforward to train even a small network meaningfully on this dataset.

The network in the example code `main_mnist()` has 784 input neurons (one per pixel), two single layers with 128 neurons each and an output layer with 10 neurons (one per digit class). The handling of loading this dataset is done using code from https://github.com/wichtounet/mnist.

Figure 3: The MNIST dataset sample

4 Evaluation

To evaluate the performance of the framework, I have chosen to train a neural network with two hidden layers on the MNIST dataset. The hidden layers are equal in size and the testing was done on several different sizes. It's the execution time per epoch that we're going to examine. As baselines for comparison I have also trained networks using the same methods but implemented in widely used deep learning frameworks TensorFlow and Theano.

The accuracy each model obtains during training is not really an important factor here. The subject of evaluation of this work is not comparing different mathematical models and their methods of initialization, it is comparing different implementations in terms of execution time. As long as the implemented math is correct, with equal initialization, both this framework and any other deep learning framework should deliver roughly the same solution. The quality of solutions of this framework is comparable to both of the used frameworks.

The evaluation was done on a machine with i5-6500 3.2 GHz quad-core, 8GB of RAM, Geforce GTX 1060 6GB. Some of the GPU properties are displayed in table 1.

Table 1: GPU specification

CUDA Driver Runtime Version	8.0
CUDA Compute Capability	6.1
Streaming Multiprocessors	10
Total CUDA Cores	1280
Max Threads per Block	1024

4.1 CPU Evaluation

The CPU implementation was tested against the CPU implementation in Theano. Obviously my CPU implementation does poorly because it is single threaded and not optimized.

Table 2: CPU implementation vs Theano

Hidden units	**My CPU [s]**	**Theano [s]**
64	90	0.8
128	186	1.12
256	412	1.95
512	1033	3.83
1024	3058	8.82
2048	10426	24.2

4.2 GPU Evaluation

The GPU implementation was tested against the GPU implementation in TensorFlow. Here the results are much more interesting and comparable. For the two smallest models, my implementation is even faster than TensorFlow because it's lightweight. For the bigger models, the optimized implementation in TensorFlow does better, my model implementation doesn't scale that well.

Table 3: GPU implementation vs TensorFlow

Hidden units	**My GPU [s]**	**TensorFlow [s]**
64	0.5	0.71
128	0.68	0.7
256	0.98	0.67
512	1.52	0.81
1024	2.96	1.47
2048	8.04	2.39

5 Conclusion

In this project I have implemented a deep learning framework which uses Nvidia CUDA for acceleration of the computation on GPU. The framework was tested on two datasets, first on a simple XOR dataset and second on a full fledged machine learning dataset, MNIST. It was benchmarked against implementations in Theano and TensorFlow. The CPU implementation compares poorly to the state-of-the-art framework Theano which was to be expected as the implementation is non-optimized and single threaded. A lot more optimization effort went into the GPU implementation and it is comparable to the state-of-the-art framework TensorFlow, execution time-wise. It performs better on smaller models due to the negligible overhead but scales worse to the big ones.

The result is still quite a success as all of the state-of-the-art frameworks take advantage of cuDNN which is a library designed and maintained directly by Nvidia for the purposes of use in deep learning. This library contains optimized methods for forward and backward for many different layers and therefore let's the developers take advantage of both their know how and their hardware. All the computations were written from scratch in this work, so worse results are to be expected.

A considerable part of this work, which is not displayed here (that would do for another report filled with formulas), was deriving the specific formulas used for the forward and backward computation. Dealing with matrix computation and derivatives of matrices is highly prone to error when done by hand, so it took non-trivial amount of time to derive all the formulas for all the layers, activation functions, loss functions. There are likely no major problems with the mathematical part as the analytical gradients were validated using the numerical approximation.

The main aim of this framework was to demonstrate, how such task as developing deep learning framework might be tackled, while using native CUDA for computational code. So the main added value is not in the performance but in the education. Both for me during the process and for anyone who is wondering what is under the hood of such framework and how it might be specifically implemented. Thanks to the minimal amount of code, it is possible to examine and see how various things are done without being overwhelmed with immense code base, which would be the case if one were to dig into any widely used deep learning framework.

6 Bibliography

References

[1] Geoffrey Hinton, *Neural Networks for Machine Learning course on Coursera.org, Lecture 6*, `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`