MLP Neural Net Parallel Implementation Using Nvidia CUDA

Szilagyi Ervin

December 4, 2016

Sapientia EMTE

Contents

1 Introduction

1.1 Problem description

Neural networks are a computational approach which is based on a large collection of neural units loosely modeling the way a biological brain. Neural networks are widely used in applications like pattern recognition and computer vision. However there can be found examples of neural net usage in other types of application. Their popularity hugely increased with the release of hardware such as graphical cards which support massive parallelization. In the year of 2007 nVidia first introduced their parallel computing platform and application programming interface named CUDA. This allowed engineers to run data parallel applications using thousands of threads maximizing computation throughput. A neural network is usually represented as interconnected neurons which basically are modeling the axons from a human brain. There are many kinds of neural network models hat have been proposed to solve some real life problems. Feed-forward neural network and Restricted Boltzmann Machine (RBM) are two of the most popular neural network models but the latest years the popularity of deep networks increased significantly.

1.2 Fully-connected feed-forward neural MLP net

A neural net is called fully-connected when every neuron in a given layer has a link with every other neuron. A neural net can have one or more layers depending on the complexity of the given problem. The first layer in the net is called input layer, the last layer is called output layer. The layers between the input and output layer are called hidden layers. The output value of each neuron from the hidden layer and from the output layer as well is calculated based on the output value of previous neurons. A graphical representation of an multi-layer perceptron neural net can be seen in Figure 1. The representation of a single neuron can be seen in Figure 2. In a fully-connected layer every neuron connects to every inputs. Every connection link has an arbitrary weight. Furthermore the neuron is represented by it's activation function which can vary depending on what kind of problem needs to be solved. Most often used activation functions are logistic, hyperbolic and rectifier. The output of a neuron can be computed following the above formula:

$$y_k = \sum w_i * x_i \tag{1}$$

2 Solution

2.1 Related work

There are several open source libraries for implementing neural networks, most of them are optimized for convolutional neural networks and deep learning networks. Some libraries which can be used are cuDNN by NVidia, tensorflow and Caffe. This are huge

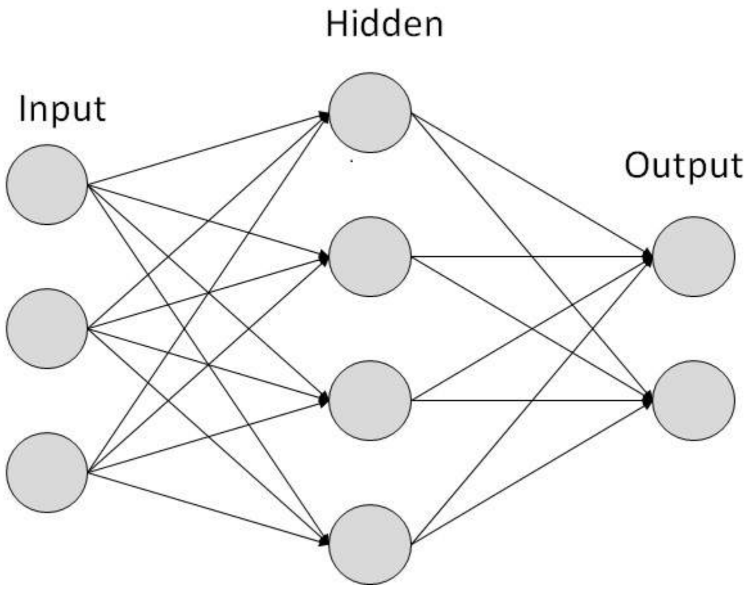

Figure 1: Fully-connected neural net.

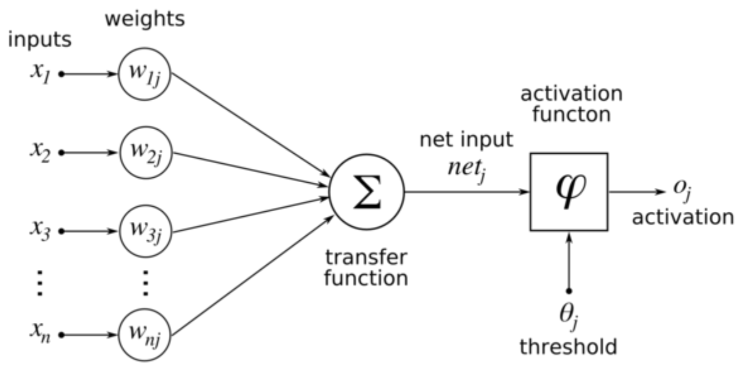

Figure 2: Neuron.

libraries focusing mainly on deep learning but they can be used for fully-connected networks as well.

2.2 Serial implementation

The training of an MLP neural net consists in two basic phases: the feed-forward step when some input is given to the net which should flow through it and by the activation of the neurons some output is generated; the secondary step is the backpropagation step where the generated output is compared to the target output resulting in some kind of an error value. This error is backpropagated through the net calibrating the weights. The feed-forward step can be implemented as it follows:

```
Parameter: Layers (vector of layers), Neuron (vector of neurons
in each layer), Inputs(vector of inputs), IN(number of inputs)
Variable: neuron (value of each neuron), i (value of each
input), layer(value of each layer), sum(placeholder)

for each layer in Layers
  sum := 0
  for each neuron in Neurons
    for i := 0 to IN
      sum += weight[i] * Inputs[i]
    end for
  neuron.activation = activationFunction(sum)
  end for
end for
```

The backpropagation step is solved using gradient descent method. The chain rule of the gradient descent can be implemented as following:

```
Parameter: Layers (vector of layers), Neuron (vector of neurons
in each layer), Outputs(vector of outputs), ON(number of outputs)
Deltas(vector of deltas)
Variable: neuron (value of each neuron), i (value of each
input), layer(value of each layer)

for each layer in Layers
  for each neuron in Neurons
    for i := 0 to ON
      gradients[i] += Deltas[i] *
        activationFunctionDerivative(Outputs[i])
    end for
  end for
end for
```

For the output layer, the delta values can be calculated by with a subtraction of the output value from the target value in case of every neuron. In case of hidden layers the delta value for every neuron in a layer is calculated by summing up the product of the weights and gradient values of every neuron from the following layer on which it has impact.

```
Parameter: myNeuronIdx (the neuron for which we compute the delta
values), N (number of neurons in the nextLayer)
for i := 0 to N
  deltas[i] = weights[i][myNeuronIdx] * gradient[i]
end for
```

After the gradients are computed, the last step is to update the weights in every layer. The weights are updated with this formula:

$$weight+ = trainRate * activatonRes * gradient + momentum * oldDelta \quad (2)$$

2.3 Parallel implementation of the feed-forward step

Every layer could have one ore more inputs. This inputs can be grouped together in a vector.

$$\begin{pmatrix} i_0 & i_1 & i_2 & ... & i_N \end{pmatrix}$$

Because of the nature of the fully-connected layer, every neuron has to have a connection with every input. The weights of a neuron can be represented as a column in a matrix. We can create having the width the number of neurons present in the layer and the height being the number of the inputs.

$$\begin{pmatrix} W_{00} & W_{01} & W_{02} & ... & W_{0M} \\ W_{10} & W_{11} & W_{12} & ... & W_{1M} \\ ... & ... & ... & ... & ... \\ W_{N0} & W_{N1} & W_{N2} & ... & W_{NM} \end{pmatrix}$$

Where:
N - number of inputs
M - number of neurons in the layer

In this case if we multiply the vector of inputs with the matrix of the weights, we get a result which applied to the activation function gives the output of the layer. Basically the feed-forward step of every layer can be done by solving a vector matrix multiplication. Using parallel programming this can be implemented with the usage of shared memory following a so called tiled approach. First of all we allocate two buffers in the shared memory, the first one is a single dimension array, the second one is a two-dimensional array. In the first buffer we will cache the input values, in the second array the weights are copied. Multiplying this arrays together we get a partial result. Next the remaining tiles are loaded into the shared memory. The partial results are summed up at the end. The final result will be a one-dimensional array of which values will be applied to the activation function. This step also is done when the results are gathered together.

2.4 Parallel implementation of the backpropagation step

Each delta value can be individually calculated at the same time. They don't have cross-dependencies. For the output layer, the parallelization is simple, we start a big number of threads, every thread should calculate a single value. At the ending this values are copied to the global memory of the GPU. The parallel implementation for the hidden layer gradients has an inner loop which calculates sum. The simple solution for this is to have a loop in every thread. Because every thread will calculate the delta value using the same number of neurons, it is guaranteed that the threads wont encounter branching and none of the threads will introduce bottlenecks. Also, this sum can be calculated by using regression but this wont really have an impact if the number of neurons are not huge. Furthermore, this implies using dynamic parallelization which wont be achievable without proper hardware support. Having the delta values already calculated the gradient values also can be easily computed launching a single kernel where every thread calculates a single gradient value. Finishing up this step, the update of the weights can be done following the same logic. It is important to know that computing the delta values, computing the gradients and updating the weights are steps that depend on each other. This excludes having them run in simultaneously. Even if we would be in the situation of having the independent steps, this would result in task parallelism for which is not advisable to use CUDA.

2.5 Memory management - Naive approach

The naive approach to do the parallelism of an MLP neural net would be that net is stored inside RAM memory. When the parallel kernels are executed, the necessary data will be transferred to the GPU memory. It turns out that this approach can massively slow down the training process because the RAM - VRAM communication is not negligible at all. The speed gain from the parallel sections is not comparable with the slow-downs occurring copying data.

2.6 Memory management - Optimized approach

The optimized approach is to allocate the memory at the beginning in the VRAM and use this memory until the training is completed. For this solution implementation of RAII (Resource acquisition is initialization) classes are the way to go. In the constructor of our net and layer class we allocate the needed memory and we will free up when the classes are destructed. This means we need to have enough GPU memory for keeping the entire net in it. Moreover, the training samples also have to be copied inside the GPU memory for a better speed gain and avoiding bottlenecks. In practice, the memory need for the training sample can exceed the memory space of the net. Using a modern GPU, the size of the memory should not be a problem, otherwise there are several methods for splitting up training samples and keeping only the necessary in the VRAM. We the memory is allocated, the CPU gets a pointer to it. This is very important looking back to the steps of the backpropagation. This way the CPU is the manager of what kernel should run at the specific moment. It also passes as an argument the pointer of the memory location which should be used.

3 Benchmarks

4 Conclusion

5 References