MLP neural net parallel implementation using CUDA

Szilágyi Ervin

Sapientia EMTE

November 24, 2016

Introduction

- This project's goal is to implement a multilayer perceptron neural net by using a parallel aproach. The most suitable framework for accomplishing this is the nVIDIA CUDA framework.

About MLP neural net

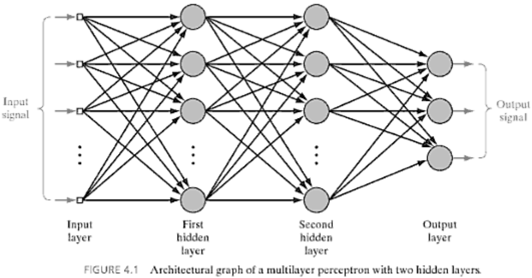

Figure: MLP Neural Net structure

MLP Neuron

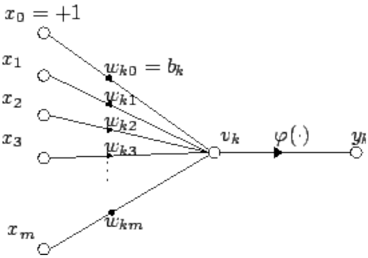

Figure: MLP neuron

MLP Neuron

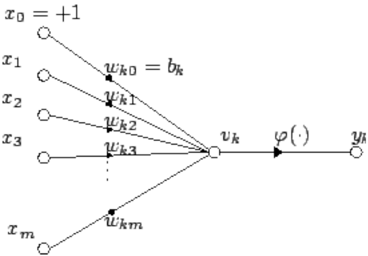

Figure: MLP neuron

Training the Net

```
For each layer
  do the feedforward step
end for each
  calculate the error
For each layer
  do the backpropagation step
  update the weights
end for each
```

Matrix representation of the weights

Inputs:

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

Weights:

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

Where:

N - number of inputs

M - number of neurons in the layer

Note:

The output can be calculated by applying the activation function to this product:

$$O = I * W$$

Backpropagation using gradient descent

```
//for the last layer
auto delta = d_targetVals[i] - d_activationResults[i];
d_gradients[i] = delta *
cuda_activeationFuncD(d_activationResults[i]);

//for hidden layers
d_gradients[i] = d_deltas[i] *
cuda_activeationFuncD(d_activationResults[i]);
```

Calculating the deltas for the hidden layer

```
for (auto i = 0; i < static_cast<decltype(i)>(sizeGrad); ++i)
{
   d_deltas[threadId] += d_weights[i * sizeOutput
   + threadId] * d_gradients[i];
}
```

Updating the weights

```
auto deltaWeight = trainRate * d_activationResults[idx]
   * d_gradients[idx] + momentum * oldDeltaWeight;
d_weights[i] += d_deltaWeights[i];
```

Normalizing the weights after update

- Problem: the activation functions usually accept values between 0.0 and 0.1 (ex: sigmoid) or -1.0 and 1.0 (ex: hyperbolic tangent).
- Solution: the weights need to be normalized to be inside these intervals.
- The minimum and maximum values needed for the normalization are calculated using reduction.

Implementation of the nerual net (Naive method)

- Keep the layers (weights, gradients) in the RAM.
- Parallelize key methods by writing cuda kernels.
- In every iteration send the values to the GPU, do the calculation, copy back the result.
- Very ineffective solution, the GPU is barely used, a lot of time is wasted by doing memory allocation and copying.

Implementation of the nerual net (Optimized approach)

- Keep the layers (weights, gradients) in the GPU memory. Use RAII classes, the GPU memory is freed up when the net is deleted.
- Parallelize every method by writing cuda kernels.
- The weights are initialized at the beginning and they are updated when an iteration is done. The inputs are sent in every iteration to the GPU.

Benchmarks

- The goal is to learn a sinus curve (20 points).
- The layer topology is: (1, 1), (1, 10), (10, 10), (10, 10), (10, 1)
- Avarage time using CPU: 13 milliseconds iteration
- Avarage time using GPU: 18 milliseconds iteration

References

- David Miller - Neural Net in C++ (https://vimeo.com/19569529)
- http://iamtrask.github.io/2015/07/27/python-network-part2/
- James A. Freeman - Neural Networks - Algorithms, Applications, and Programming Techniques

Source code and documentation of this project can be found here:
https://github.com/Ernyoke/cuda$_N N$