Lab 6
Parallel and Distributed Algorithms

Neural network training

Author: Adam Brzeski, Karol Draszawka
Date: 26.05.2017

1 Aims of the laboratory

The aim of the last laboratory is to practice distributed programming skills by implementing distributed neural network training. The implementation will utilize the provided java framework for simulating distributed system, which was already used in the previous classes.

Unlike the previous assignments, Lab 6 will take two laboratory classes. This instruction covers all tasks planned for both classes.

The code for this lab is in Lab06.zip file. Extracted folder contain a project for Netbeans IDE. The source can be however easily imported to other IDEs (Idea, Eclipse).

2 Neural network training

The assignment will will cover simple neural networks consisting of 3 layers: input layer, hidden layer and output layer. This kind of network in fact called a 2-layer network, since the input layer is not counted, as it only passes unchanged input vectors to the first hidden layer. The number of neurons in the input and hidden layers will be set in the constructor of the network, while the output layer will be fixed and will contain one neuron. The network will use ReLU activation functions and the training will use several weight update methods.

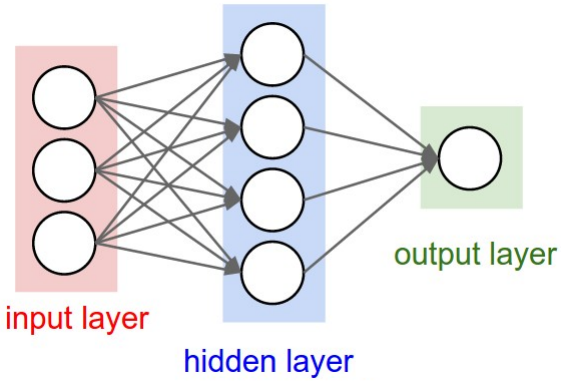

Figure 1: An example of a 2-layer neural network (the input layer is not counted) with a single neuron in the output layer.
Source: http://cs231n.github.io/neural-networks-1/

Training data for a neural network consists of a set of input vectors associated with expected output values. The set of input vectors form an input matrix, called *trainDataIn* in the laboratory code. The width of the matrix is equal to the number of vectors, while the height is equal to vector length, and equal to the number of input neurons. The set of output values form an output matrix (*trainDataOut*), which in fact is a vector (since in this case there is only one output neuron). The training data is usually split into batches, which are then subsequently used in training iterations. The training iterations consists of using data batch to perform 3 steps:

1. Forwards pass
2. Backward pass
3. Weight update

In the forward pass data batch is propagated through the network, generating some output values at the end of the network. The outputs are then compared to the expected output values and the differences are evaluated as an error rate. Then, knowing the output error rate acquired for the data batch, is propagated backwards (backwards pass), during which gradients are computed, denoting the impact of all of the weights of the given layer on the error rate. Finally, with gradients established for all layers (excluding input layer, which has no weights), the weights are updated using a given method (e. g. Rprop). It should be noted, that the weights (and biases) of each layer are also represented as matrices, therefore forward and backward passes are matrix operations.

3 Distributed training (DATA PARALLELISM)

In distributed training using data parallelism scheme, each node will keep the full model of the network (and all of it's weights) in the memory. The nodes will be however given different data batches to process. Each node will then perform forward and backwards pass. Then, synchronization between the nodes is required, which takes place in three steps:

1. accumulation of gradients (computing the mean of gradients acquired by each node for every single weight and bias over their batches)
2. updating weights in node 0
3. broadcasting updated weights to all nodes, which update their local copy of the model

4 Student's tasks

There are two tasks to complete, as listed below. The tasks require implementing proper methods in two classes representing parallel neural networks. As reference, the already implemented methods of sequential training can be used, provided in *FFNet* and *FFNetSimple* classes.

1. **Distributed neural network training – 6 pts**
    - requires implementing the following methods of the *neuralnets.FFNetParallel* class:

        void accumulateGrads(int wholeTrainDataSize)
        void broadcastWeights()

    - requires passing Test 1: `compareParallelToSequential()`

-

2. **Extending per node processing with multithreading – 4 pts**
   - requires implementing the following method of the *neuralnets.FFNetParallelDouble* class:

     Matrix forwardPass(Matrix input)
     void backwardPass(Matrix lastDelta)

   - requires passing Test 2: `compareDoubleParallelToParallel()`

5 Cheat Sheet

`FFNet._InW`	Matrix with weights of the connections between the input layer and the hidden layer
`FFNet._InBiasW`	Weights of the biases of the hidden layer
`FFNet._InWGrad`	Gradients of the weights of the hidden layer
`FFNet._InBiasWGrad`	Gradients of the weights of the biases of the hidden layer
`FFNet._LayerW`	Matrix with weights of the connections between the hidden layer and the output layer
`FFNet.LayerBiasW`	Weights of the biases of the output layer
`FFNet._LayerWGrad`	Gradients of the weights of the output layer
`FFNet._LayerBiasWGrad`	Gradients of the weights of the biases of the output layer
`FFNet._AFHidden`	Activation function of the hidden layer neurons
`FFNet._HiddenState`	Output of the hidden layer neurons

`BasicCommunication.reduceWithBarrier (node, data, operator)`	Reduces the distributed data parts to a single part by applying given operator. The result is returned only in the node 0. Also, at end of the operation all of the nodes are synchronized (like a barrier).
`BasicCommunication.broadcastWithBarrier(node, data)`	Broadcast the data from the node 0 to all other data. The return value contains the broadcasted data. Also, at end of the operation all of the nodes are synchronized (like a barrier).
`Node.synchronizeDS()`	Barrier for the nodes - forces synchronization. The method will finish when all of the nodes in the system reach the function.