Batch Pattern Parallelization Scheme of Neural Networks on Many-core Architectures

Dr. Volodymyr O. Turchenko
short name (Vlad)
Ph.D. in Computer Engineering
Associate Professor,
Fulbright Research Scholar,
Innovative Computing Laboratory, EECS,
The University of Tennessee,
1122 Volunteer Blvd., Knoxville, TN, 37996, USA
E-mail: volodymyr.turchenko@gmail.com
Web: http://uweb.deis.unical.it/turchenko/
<table>
<thead>
<tr>
<th>Outline</th>
<th>Slides</th>
</tr>
</thead>
<tbody>
<tr>
<td>➢ Results of recirculation NN parallelization on many-core architectures using MPI</td>
<td>2-14</td>
</tr>
<tr>
<td>➢ Development of an OpenMP version of the algorithm</td>
<td>15-19</td>
</tr>
<tr>
<td>➢ NN application for drop bandwidth prediction for Terabits project</td>
<td>20-26</td>
</tr>
<tr>
<td>➢ NN application for cloud spot price prediction</td>
<td>27-30</td>
</tr>
<tr>
<td>➢ Using Matlab Neural Network Toolbox: an example</td>
<td>31-33</td>
</tr>
<tr>
<td>➢ Conclusions &amp; future work</td>
<td>34</td>
</tr>
</tbody>
</table>
Neural Networks and Their Parallelization

**Multi-layer perceptron N-M-1**

Output value of the perceptron

\[ y = F_3 \left( \sum_{j=1}^{N} w_{j3} \left( F_2 \left( \sum_{i=1}^{M} w_{ij} x_i - T_j \right) \right) \right) - T \]

Sigmoid activation function of hidden layer neurons

\[ F(x) = \frac{1}{1 + e^{-x}} \]

Goal of training: \( E(t) \rightarrow \min \)

Modification of weights and thresholds

\[ \Delta w_{ij}(t) = -\alpha \frac{\partial E^p(t)}{\partial w_{ij}(t)} \]

\[ \Delta T_j(t) = -\alpha \frac{\partial E^p(t)}{\partial T_j(t)} \]

**Levels of parallelism**

- **connection** parallelism (parallel execution on sets of weights)
- **node** parallelism (parallel execution of operations on sets of neurons)
- **pattern** (exemplar) parallelism (parallel execution of operations on set of NN’s training patterns)
- **example** (modular) parallelism (parallel execution of examples on replicated networks)

- First three levels are **fine-grain** parallelism & the fourth level is **coarse-grain** parallelism

- **Fine-grain** parallel algorithms require a lot of low-level communications, for example to combine the results of parallel calculation of weights of each neuron

- **Coarse-grain** algorithms carry out main computation operations on processors of parallel machine. The communications are rarely executed taking into account functional sense of the algorithm

V. Turchenko, ICL, 05/31/2013, Batch Pattern Parallelization Scheme of NNs on Many-core Arch.
Sequential Batch Pattern Training Algorithm for MLP

![Diagram of a three-layer perceptron](image)

The batch pattern BP training algorithm consists of the following steps:

1. Set $SSE = E_{\text{min}}$ and training iterations $t$;
2. Initialize the weights and thresholds;
3. Calculate for the training pattern $pt$:
   3.1. The output value $y^p(t)$;
   3.2. The error of the output neuron $\gamma_2^p(t) = y^p(t) - d^p(t)$;
   3.3. The error of hidden layer neuron $\gamma_i^p(t) = \gamma_i^p(t) \cdot w_{ij}(t) \cdot y^p(t) \cdot (1 - y^p(t))$;
   3.4. The delta weights and delta thresholds of all perceptron’s neurons and add the result to the value of the previous pattern
      
      \[
      \begin{align*}
      s\Delta w_{ij} &= s\Delta w_{ij} + \gamma_j^p(t) \cdot F_j(S_j^p(t)) \cdot h_j^p(t), \\
      s\Delta T &= s\Delta T + \gamma_j^p(t) \cdot F_j(S_j^p(t)) \cdot \dot{h}_j^p(t), \\
      s\Delta w_j &= s\Delta w_j + \gamma_j^p(t) \cdot F_j(S_j^p(t)) \cdot h_j^p(t), \\
      s\Delta T_j &= s\Delta T_j + \gamma_j^p(t) \cdot F_j(S_j^p(t)) \cdot \dot{h}_j^p(t),
      \end{align*}
      \]
      
      \[
      F_j(S_j^p(t)) = y^p(t) \cdot (1 - y^p(t)), \\
      F_j'(S_j^p(t)) = y_j^p(t) \cdot (1 - y_j^p(t)).
      \]
   3.5. The $SSE$ using $E^p(t) = \frac{1}{2} \left( y^p(t) - d^p(t) \right)^2$;
4. Repeat step 3 for each $pt$, $pt \in \{1, ..., PT\}$, $PT$ is the size of the training set;
5. Update the weights and thresholds $w_j(PT) = w_j(0) - \alpha(t) \cdot s\Delta w_j$, $T_j(PT) = T_j(0) + \alpha(t) \cdot s\Delta T_j$;
6. Calculate the total $SSE$ $E(t)$ on the training iteration $t$ using $E(t) = \sum_{pt=1}^{PT} E^p(t)$;
7. If $E(t)$ is greater than the desired error $E_{\text{max}}$ then increase the number of training iteration to $t + 1$ and go to step 3, otherwise stop the training process.

V. Turchenko, ICL, 05/31/2013, Batch Pattern Parallelization Scheme of NNs on Many-core Arch.

We calculated the delta weight and thresholds for all training patterns sequentially according to 3.4 and then update all weights by one operation in point 5 - this is a precondition for parallelizing of this algorithm.
Sequential Batch Pattern Algorithm for RNN

\[ y = F_3 \left( \sum_{j=1}^{M} w_{j3} \left( F_2 \left( \sum_{i=1}^{N} w_{ij} x_i + \sum_{k=1}^{M} w_{kj} h_k(t-1) + w_{3j} y(t-1) - T_j \right) \right) \right) - T_0 \]

The batch pattern BP training algorithm consists of the following steps [13]:

1. Set the desired error (Sum Squared Error) \( E_{\text{sum}} \) and the number of training iterations \( T \);
2. Initialize the weights and the thresholds of the neurons with values in range (0...0.5) [13];
3. For the training pattern \( pt \):
   3.1. Calculate the output value \( y^p(t) \) by expression (1);
   3.2. Calculate the error of the output neuron \( e_j^p(t) = d^p(t) - y^p(t) \), where \( y^p(t) \) is the output value of the perceptron and \( d^p(t) \) is the target output value;
   3.3. Calculate the error of the neurons of the hidden layer \( e_j^p(t) = e_j^p(t) \cdot F'_j(S_j^p(t)) \), where \( S_j^p(t) \) is the weighted sum of the output neuron;
   3.4. Calculate the delta weights and delta thresholds of all neurons and add the result to the value of the previous pattern \( \Delta w_{j} = \Delta w_{j}(t) + e_j^p(t) \cdot F'_j(S_j^p(t)) \cdot h_j^p(t) \), \( \Delta T_j = \Delta T_j(t) + e_j^p(t) \cdot F'_j(S_j^p(t)) \cdot h_j^p(t) \) , \( \Delta w_{j} = \Delta w_{j}(t) + e_j^p(t) \cdot F'_j(S_j^p(t)) \cdot h_j^p(t) \) , \( \Delta T_j = \Delta T_j(t) + e_j^p(t) \cdot F'_j(S_j^p(t)) \cdot h_j^p(t) \) , where \( S_j^p(t) \) and \( h_j^p(t) \) are the weighted sum and the output value of the \( j \) hidden neuron in the current moment of time \( t \) respectively;
   3.5. Calculate the SSE using \( E^p(t) = \frac{1}{2} (y^p(t) - d^p(t))^2 \);
4. Repeat the step 3 above for each training pattern \( pt \), where \( pt \in [1, PT] \); \( PT \) is the size of the training set;
5. Update the weights and thresholds of neurons using \( w_j(PT) = w_j(0) - \alpha(t) \cdot \Delta w_j \), \( w_{ij}(PT) = w_{ij}(0) - \alpha(t) \cdot \Delta w_{ij} \), \( w_{j3}(PT) = w_{j3}(0) - \alpha(t) \cdot \Delta w_{j3} \), \( T_j(PT) = T_j(0) + \alpha(t) \cdot \Delta T_j \), \( T_0(PT) = T_0(0) + \alpha(t) \cdot \Delta T_0 \), where \( \alpha(t) \) is the learning rate;
6. Calculate the total SSE \( E(t) \) on the training iteration \( t \) using \( E(t) = \sum_{p=1}^{PT} E^p(t) \);
7. If \( E(t) \) is greater than the desired error \( E_{\text{sum}} \) then increase the number of training iteration to \( t+1 \) and go to step 3, otherwise stop the training process.

Figure 1. Structure of recurrent neural network

We calculated the delta weight and thresholds for all training patterns sequentially according to 3.4 and then update all weights by one operation in point 5 - this is a precondition for parallelizing of this algorithm.
The batch pattern BP training algorithm consists of the following:
1. Set the desired SSE = $E_{\text{min}}$ and training epochs $t$;
2. Initialize the weights and the thresholds of the neurons;
3. Calculate for the training pattern $pt$:
   3.1. The output value $\bar{x}_i(t)$;
   3.2. The errors of the output neurons $\gamma_i^{pt}(t) = (\bar{x}_i^{pt}(t) - x_i^{pt}(t))$;
   3.3. The errors of the hidden layer neurons $\gamma_j^{pt}(t) = \sum_{i=1}^{s} \gamma_i^{pt}(t) \cdot w_{ji}^{pt}(t) \cdot F'_j(S_i^{pt}(t))$;
   3.4. The delta weights and delta thresholds of all neurons and add the result to the value of the previous pattern $s\Delta w_j^{pt} = s\Delta w_j^{pt} + \gamma_j^{pt}(t) \cdot F'_j(S_i^{pt}(t)) \cdot y_j^{pt}(t),
     s\Delta w_i^{pt} = s\Delta w_i^{pt} + \gamma_i^{pt}(t) \cdot F'_i(S_j^{pt}(t)) \cdot x_i^{pt}(t)$;
4. Repeat the step 3 for all training patterns $pt, pt \in \{1, \ldots, PT\}$;
5. Update the weights and thresholds using $w_j^{pt}(PT) = w_j^{pt}(0) - \alpha_j(t) \cdot s\Delta w_j^{pt}$ and $w_i^{pt}(PT) = w_i^{pt}(0) - \alpha_i(t) \cdot s\Delta w_i^{pt}$;
6. Calculate the total SSE $E(t)$ on the training epoch $t$ using $E(t) = \sum_{pt=1}^{PT} E^{pt}(t)$;
7. If $E(t) > E_{\text{min}}$ then increase the number of training epochs to $t + 1$ and go to step 3, otherwise stop the training process.

We calculated the delta weight and thresholds for all training patterns sequentially according to 3.4 and then update all weights by one operation in point 5 - this is a precondition for parallelizing this algorithm.
Sequential Batch Pattern Training Algorithm for RBF

We calculated the delta \( w_{ij}, G_j \) and \( v_j \) for all training patterns sequentially according to 3.4 and then update all weights by one operation in point 5 - this is a precondition for parallelizing this algorithm.
### Implementation Details of Batch Pattern Scheme

<table>
<thead>
<tr>
<th>Model</th>
<th>Number of neurons in layers (Number of connections)</th>
<th>Message passing size, elements (bytes)</th>
<th>Mathematical expressions to modify weights and thresholds</th>
</tr>
</thead>
</table>
| MLP   | 40-40-1 (1681)                                     | 1681 (13448)                           | \[ s\Delta w_{ji} = s\Delta w_{ji} + \gamma_{ji}(t) \cdot F_j'(S_j(t)) \cdot h_j(t) \]  \
|       |                                                    |                                        | \[ s\Delta w_{ij} = s\Delta w_{ij} + \gamma_{ij}(t) \cdot F_i'(S_i(t)) \cdot x_i(t) \]  \
|       |                                                    |                                        | \[ s\Delta T_j = s\Delta T_j + \gamma_{ji}(t) \cdot F_j'(S_j(t)) \], \[ s\Delta T = s\Delta T + \gamma_{ji}(t) \cdot F_j'(S_j(t)) \] |
| RNN   | 40-40-1 (3321)                                     | 3321 (26568)                           | \[ s\Delta w_{ji} = s\Delta w_{ji} + \gamma_{ji}(t) \cdot F_j'(S_j(t)) \cdot h_j(t) \]  \
|       |                                                    |                                        | \[ s\Delta w_{ij} = s\Delta w_{ij} + \gamma_{ij}(t) \cdot F_i'(S_i(t)) \cdot x_i(t) \]  \
|       |                                                    |                                        | \[ s\Delta w_{ij} = s\Delta w_{ij} + \gamma_{ij}(t) \cdot F_i'(S_i(t)) \cdot h_i(t-1) \]  \
|       |                                                    |                                        | \[ s\Delta w_{ij} = s\Delta w_{ij} + \gamma_{ij}(t) \cdot F_i'(S_i(t)) \cdot y_i(t-1) \]  \
|       |                                                    |                                        | \[ s\Delta T_i = s\Delta T_i + \gamma_{ji}(t) \cdot F_j'(S_j(t)) \], \[ s\Delta T_j = s\Delta T_j + \gamma_{ij}(t) \cdot F_i'(S_i(t)) \] |
| RBF   | 40-40-1 (1680)                                     | 1680 (13440)                           | \[ s\Delta w_{ij} = s\Delta w_{ij} + \gamma_{ij}(t) \cdot v_j \cdot \exp \left(- \frac{|x_j - w_{ij}|^2}{2G_j^2} \right) \cdot \left( \frac{x_j - w_{ij}}{G_j^2} \right) \]  \
|       |                                                    |                                        | \[ s\Delta G_j = s\Delta G_j + \gamma_{ij}(t) \cdot v_j \cdot \exp \left(- \frac{|x_j - w_{ij}|^2}{2G_j^2} \right) \cdot \left( \frac{|x_j - w_{ij}|^2}{G_j^2} \right) \]  \
|       |                                                    |                                        | \[ s\Delta v_j = s\Delta v_j + \gamma_{ij}(t) \cdot y_j \] |
| RCNN  | 40-20-40 (1600)                                    | 1600 (12800)                           | \[ s\Delta w_{ji} = s\Delta w_{ji} + \gamma_{ji}(t) \cdot F_j'(S_j(t)) \cdot y_j(t) \]  \
|       |                                                    |                                        | \[ s\Delta w_{ij} = s\Delta w_{ij} + \gamma_{ij}(t) \cdot F_i'(S_i(t)) \cdot x_i(t) \] |

V. Turchenko, ICL, 05/31/2013, Batch Pattern Parallelization Scheme of NNs on Many-core Arch.
• The sum operations of delta weights and thresholds are independent of each other
• The *Master* divides all patterns in equal parts corresponding to number of the *Slaves*
• Then the *Master* sends to the *Slaves* the numbers of the appropriate patterns to train
• Each Slave calculates the partials of delta weights, thresholds and SSE only for its assigned number of training patterns
• The global operations of reduction and summation should be executed just after the synchronization point. Then the summarized values of the delta weight and thresholds should be sent to all CPUs working in parallel. We implement this by MPI_Allreduce()

  • The *MPI_Allreduce()* function includes an internal synchronization point – we have removed the function *MPI_Barrier()* from our code
  • Reduce operation was implemented by only one call of *MPI_Allreduce()* with pre-encoding of all data into one message before sending and after-decoding the data to appropriate matrixes after receiving
The code is eliminated, please contact the author
Experimental Environment

**Many-core system Remus** consists of two socket G34 motherboards RD890 (AMD 890FX chipset) connected each other by AMD Hyper Transport technology. Each motherboard contains two twelve-core AMD Opteron 6180 SE processors with a clock rate of 2500 MHz and 132 GB of local RAM. Thus the total number of computational cores is **48**.

Cluster **Dancer** consists of 16 nodes, each node has two four-core Intel(R) Xeon(R) processors E5520 with a clock rate of 2270 MHz and 12 GB of local RAM. The nodes are connected by Infiniband G20 interface. The total number of used computational cores is **48**.

**MIC**: Intel® Xeon Phi™ Coprocessor 5110P (8GB, 1.053 GHz, **60 cores**, 7GB dedicated memory) – on **Lips** machine.

**Researched RCNN architectures**
The RCNN model **41-5-41** will be having 420 updating connections (multiple per 8 = 3280 bytes of communication message) and the RCNN model **41-30-41** will be having 2640 updating connections (19680 bytes of communication message).
Efficiency Comparison on Many-core and Cluster

Parallelization efficiency for RCNN minimum scenario 41-5-41 on many-core (left, r:Remus) and cluster (right, d:Dancer) architectures

Parallelization efficiency for RCNN maximum scenario 41-30-41 on many-core (left, r:Remus) and cluster (right, d:Dancer) architectures

V. Turchenko, ICL, 05/31/2013, Batch Pattern Parallelization Scheme of NNs on Many-core Arch.
Running MPI Parallel Routine on MIC Intel Xeon Phi

1: Copy the MPI libraries to the appropriate directories of the MIC:
[vtu ~]$ scp /mnt/scratch/sw/intel/impi/4.1.0.030/mic/bin/mpiexec mic1:/bin/mpiexec
[vtu ~]$ scp /mnt/scratch/sw/intel/impi/4.1.0.030/mic/bin/pmi_proxy mic1:/bin/pmi_proxy
[vtu ~]$ scp /mnt/scratch/sw/intel/impi/4.1.0.030/mic/lib/libmpi.so.4.1
    mic1:/lib64/libmpi.so.4
[vtu ~]$ scp /mnt/scratch/sw/intel/impi/4.1.0.030/mic/lib/libmpigf.so.4.1
    mic1:/lib64/libmpigf.so.4
[vtu ~]$ scp /mnt/scratch/sw/intel/impi/4.1.0.030/mic/lib/libmpigc4.so.4.1
    mic1:/lib64/libmpigc4.so.4
[vtu ~]$ scp /mnt/scratch/sw/intel/composer_xe_2013.3.163/compiler/lib/mic/libimf.so
    mic1:/lib64/libimf.so
[vtu ~]$ scp /mnt/scratch/sw/intel/composer_xe_2013.3.163/compiler/lib/mic/libsvml.so
    mic1:/lib64/libsvml.so
[vtu ~]$ scp /mnt/scratch/sw/intel/composer_xe_2013.3.163/compiler/lib/mic/libintlc.so.5
    mic1:/lib64/libintlc.so.5

2: Compile the routine on the host: mpiicc -mmic recnn2pmic.c -o recnn2pmic.mic
3: Copy compiled executable and necessary files to the MIC:
   scp ./recnn2pmic.mic mic1:/tmp/recnn2pmic.mic
4: Login to the MIC: ssh root@mic1
5: Run the parallel routine on the MIC: mpirun –np 60 /tmp/recnn2pmic.mic

V. Turchenko, ICL, 05/31/2013, Batch Pattern Parallelization Scheme of NNs on Many-core Arch.
Efficiency Comparison on Many-core, Cluster and MIC Intel Xeon Phi

V. Turchenko, ICL, 05/31/2013, Batch Pattern Parallelization Scheme of NNs on Many-core Arch.
OpenMP Implementation of Parallel Algorithm

The code is eliminated, please contact the author
This huge drop of parallelization efficiency is because we are using the shared arrays in the memory of one NUMA node and then trying to commit this shared array in that memory from all processes working on three other NUMA nodes.

Solutions: to initialize the memory closer to the threads in local NUMA node, cache re-use, to provide another reduction mechanism.
Improved OpenMP Implementation of Parallel Algorithm

Memory initialization
#pragma omp parallel
{
    The code is eliminated, please contact the author
}

Cache re-use examples
#pragma omp for schedule(guided)
{
    The code is eliminated, please contact the author
}

Another reduction mechanism
#pragma omp critical
{
    The code is eliminated, please contact the author
}
The development of an efficient OpenMP parallel implementation is a non-trivial task because we are dealing with shared arrays and variables and it is necessary to be very careful to avoid thread racing.

It is necessary to take into account the architecture of the target parallel system as much as possible.

V. Turchenko, ICL, 05/31/2013, Batch Pattern Parallelization Scheme of NNs on Many-core Arch.
The OpenMP implementation spent much more time on the reduction operation.
Terabits Project: End-to-end Data Transfer Orchestration

- Goal: Minimize the end-to-end transfer time
- Online predict the network capacity and adapt the bandwidth
The scheduling limit the goodput based on the amount of data to be transferred per link, not only on the total available bandwidth.
Under intense bandwidth stress random data streams will be affected, and their potential bandwidth drastically decreased (no fairness guaranteed).
Performance feed-back mechanism is used to learn about specific classes of perturbations and their impact (duration and weight) on the data transfers.
WAN: TCP Bandwidth Under Stress

Incertainties of the bandwidth

Neural Network analysis

perturbation detection

bandwidth drop: one stream

New perturbations will be predicted and classified. The training will continue during the entire transfer. Traces will be used to predict characteristics of the traffic based on temporal properties.

V. Turchenko, ICL, 05/31/2013, Batch Pattern Parallelization Scheme of NNs on Many-core Arch.
The bandwidth will be dynamically adjusted to satisfy the requirement determined by the Z-scheduler.
Experimental results

Input data: 5 events with 30 sec and 60 sec duration, totally 10 events

Input data: 103 vectors, 63 for the training and 40 for the testing

Ideal result of the NN: to classify types of the events only (5 events) and type & duration of the events (10 events) with a detection rate 100% ON the first timestamp (on the THIRD SECOND from the START of the EVENT)

Architecture of NN: Multi-layer perceptron, 10 input, 10 hidden and 1 output neuron, sigmoid activation

Training of NN: Back-propagation error, SSE=0.048, 400000 training iterations, moving simulation mode, 200000 training iteration of re-training, 14 seconds of re-training on Inter

Results on the training set (63 vectors):
- detection rate to classify only type of the event (5 events p1-p5) - 93.7%
- detection rate to classify type & duration of the event both (10 events p1-p5 with 30 sec and 60 sec) - 73.1%

Results on the testing set (40 vectors):
- detection rate to classify only type of the event (5 events p1-p5) - 40.0%
- detection rate to classify type & duration of the event both (10 events p1-p5 with 30 sec and 60 sec) - 25.0%

Obtained results showed good adaptability and good potential capabilities of NNs to solve this task.
Neural Network Description: how to use

Multi-layer perceptron N-M-1

- NNs can require a high computational load in the training phase
- High computational load and large training time make worse wide NN application for scientific research and real-time systems
- Therefore NN parallelization is one of the technologies, which could outperform this disadvantage

<table>
<thead>
<tr>
<th>k</th>
<th>$x_1^k$</th>
<th>$x_2^k$</th>
<th>$x_3^k$</th>
<th>$d^k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.81</td>
<td>5.84</td>
<td>6.27</td>
<td>6.50</td>
</tr>
<tr>
<td>2</td>
<td>5.84</td>
<td>6.27</td>
<td>6.50</td>
<td>6.80</td>
</tr>
<tr>
<td>3</td>
<td>6.27</td>
<td>6.50</td>
<td>6.80</td>
<td>6.95</td>
</tr>
<tr>
<td>4</td>
<td>6.50</td>
<td>6.80</td>
<td>6.95</td>
<td>7.20</td>
</tr>
<tr>
<td>5</td>
<td>6.80</td>
<td>6.95</td>
<td>7.20</td>
<td>7.28</td>
</tr>
<tr>
<td>6</td>
<td>6.95</td>
<td>7.20</td>
<td>7.28</td>
<td>7.40</td>
</tr>
<tr>
<td>7</td>
<td>7.20</td>
<td>7.28</td>
<td>7.40</td>
<td>7.80</td>
</tr>
<tr>
<td>8</td>
<td>7.28</td>
<td>7.40</td>
<td>7.80</td>
<td>8.34</td>
</tr>
</tbody>
</table>

Training set: 3-(M=3)-1

| Prediction set: | 1 | 7.40 | 7.80 | 8.34 |

V. Turchenko, ICL, 05/31/2013, Batch Pattern Parallelization Scheme of NNs on Many-core Arch.
Neural-based Spot Market Prediction for Cloud Computing


V. Turchenko, ICL, 05/31/2013,  Batch Pattern Parallelization Scheme of NNs on Many-core Arch.  28/34
Neural-based Spot Market Prediction for Cloud Computing

V. Turchenko, ICL, 05/31/2013, Batch Pattern Parallelization Scheme of NNs on Many-core Arch.
A moving simulation mode approach was used to remove elder data for NN retraining in order to improve a prediction accuracy of the model.

The experimental simulation modeling results on the Amazon EC2 spot instances showed high correlation accuracy of the proposed approach given that the average relative prediction error does not exceed 4% and the number of outliers is not more than 5% for the total number of the prediction results.

This shows that NNs are well suited for prediction and are useful for users bidding on spot instance services.

V. Turchenko, ICL, 05/31/2013, Batch Pattern Parallelization Scheme of NNs on Many-core Arch.
Using MATLAB Neural Network toolbox

V. Turchenko, ICL, 05/31/2013, Batch Pattern Parallelization Scheme of NNs on Many-core Arch.
Main Functions to Work with NN in Matlab: NEWFF

Create feed-forward backpropagation network

Syntax

\[
\text{net} = \text{newff} \left(P, T, [S_1 \ S_2 \ldots S_{(N-1)}], \{T_{F1} \ T_{F2} \ldots T_{FN}\}, \\
BTF, BLF, PF, IPF, OPF, DDF\right)
\]

Description

\text{newff}(P, T, [S_1 \ S_2 \ldots S_{(N-1)}], \{T_{F1} \ T_{F2} \ldots T_{FN}\}, BTF, BLF, PF, IPF, OPF, DDF) takes several arguments:

- \( P \) \( R \times Q_1 \) matrix of \( Q_1 \) sample \( R \)-element input vectors
- \( T \) \( S_N \times Q_2 \) matrix of \( Q_2 \) sample \( S_N \)-element target vectors
- \( S_i \) Size of \( i \)th layer, for \( N-1 \) layers, default = \([]\). (Output layer size \( S_N \) is determined from \( T \).)
- \( T_{Fi} \) Transfer function of \( i \)th layer. (Default = 'tansig' for hidden layers and 'purelin' for output layer.)
- \( BTF \) Backpropagation network training function (default = 'trainlm')
- \( BLF \) Backpropagation weight/bias learning function (default = 'learngdm')
- \( IPF \) Row cell array of input processing functions. (Default =
  \{'fixunknowns', 'removeconstantrows', 'mapminmax'\})
- \( OPF \) Row cell array of output processing functions. (Default = \{'removeconstantrows', 'mapminmax'\})
- \( DDF \) Data division function (default = 'dividerand')
Main Functions to Work with NN in Matlab: TRAIN and SIM

Train neural network

**Syntax**

\[ \text{[} \text{net, tr, Y, E, Pf, Af}\text{]} = \text{train(} \text{net, P, T, Pi, Ai, VV, TV}\text{)} \]

**To Get Help**

Type `help network/train`.

**Description**

`train` trains a network `net` according to `net.trainFcn` and `net.trainParam`. `train(net, P, T, Pi, Ai)` takes

- `net` Network
- `P` Network inputs
- `T` Network targets (default = zeros)
- `Pi` Initial input delay conditions (default = zeros)
- `Ai` Initial layer delay conditions (default = zeros)

Simulate neural network

**Syntax**

\[ \text{[} \text{Y, Pf, Af, E, perf}\text{]} = \text{sim(} \text{net, P, Pi, Ai, T}\text{)} \]

\[ \text{[} \text{Y, Pf, Af, E, perf}\text{]} = \text{sim(} \text{net, (Q, TS), Pi, Ai, T}\text{)} \]

\[ \text{[} \text{Y, Pf, Af, E, perf}\text{]} = \text{sim(} \text{net, (Q, P), Pi, Ai, T}\text{)} \]

**To Get Help**

Type `help network/sim`.

**Description**

`sim` simulates neural networks.

\[ \text{[} \text{Y, Pf, Af, E, perf}\text{]} = \text{sim(} \text{net, P, Pi, Ai, T}\text{)} \text{ takes} \]

- `net` Network
- `P` Network inputs
- `Pi` Initial input delay conditions (default = zeros)
- `Ai` Initial layer delay conditions (default = zeros)
- `T` Network targets (default = zeros)
Conclusions and Future work

(i) the parallelization efficiency of the developed parallel batch pattern back propagation training scheme is high enough for its efficient use on general-purpose parallel computer systems

(ii) it is necessary to pay attention on the implementation details of the parallel algorithm in the framework of concrete parallelization library/system/technology because even well-designed algorithm (on an algorithmic level) may not show good parallelization speedup within its implementation

(iii) The development of parallel training algorithms of bio-inspired neural networks, in particular, Convolutional Neural Network is the next step of my research

I would like to thank very much Jack for inviting me and all people here at the ICL for the permanent support and help during my stay