DRAGON: DIRECTED ACYCLIC GRAPHICAL OBJECTIVE-ORIENTED NEURAL-NETWORKS

KAI FAN[*] AND MENGKE LIAN[*]

CONTENTS

1 Introduction 2
2 Related Work 2
3 Model Description 2
3.1 Feedforward Neural Network 2
3.2 DRAGON model 4
3.3 Batched Stochastic Gradient Descent (B-SGD) Algorithm 6
4 Implementation Details 7
4.1 Serial/OpenMP Implementation 7
4.2 CUDA Implementation 7
5 Discussion and Results 8
6 Conclusion 9

ABSTRACT

We proposed a new directed acyclic graphical formulation of neural networks and implemented into objective-oriented parallel computing library with CUDA C++. This model enables to train not only standard feedforward neural networks but also some probabilistic neural networks, such as variational auto-encoder. The experimental results show our package is significantly faster than specifically implemented Matlab program.

* *Duke University, Durham, NC, USA*

1 INTRODUCTION

In machine learning and cognitive science, artificial neural networks are a family of models inspired by biological neural networks, including a hierarchical structure with tons of neurons in each layer. Universal approximation theorem [?] states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets under mild assumptions on the activation function. Thus using neural networks to estimate or approximate function in machine learning prediction is popular, especially for sufficient large training datasets. Recently, the trend of study in neural network favors very deep hierarchical or more complicated network structure [?], and these models show excellent performance in many areas, such as computer vision, natural language processing or recommendation systems. However, it is crucial that computation complexity grows drastically with more hidden layers and more neurons per hidden layer due to the high dimensional matrix operation and large datasets, thus motivating the parallelism implementation, like Caffee [?], Theano[1] and TensorFlow[2].

In this project, we extend the neural network from typical chain structure to direct acyclic graph (DAG) structure and introduce 'combine function' to merge outputs among several hidden layers, i.e. allowing multiple children and parents layers. This model can be used to train standard feed forward neural networks, recursive neural networks or some probabilistic neural networks, such as variational auto-encoder [?]. Beside, we encapsulate the model into C++ class so that it can be reused and further extended as a library. Three versions are implemented. Basically, the serial version and OpenMP version are slower than MATLAB because matrix operation are implemented without BLAS library; however, the CUDA version takes advantage of cuBLAS and runs 10 times faster than MATLAB in our experiment.

2 RELATED WORK

There are a couple of deep learning package implemented with GPU and CUDA. Caffe is a deep learning CUDA C++ framework made with expression, speed, and modularity in mind, which is mainly used for convolutional neural networks. Theano is a Python library that allows user to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently, which is particularly design for deep learning. TensorFlow is an open source software library for numerical computation using data flow graphs, which is similar to our work. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them.

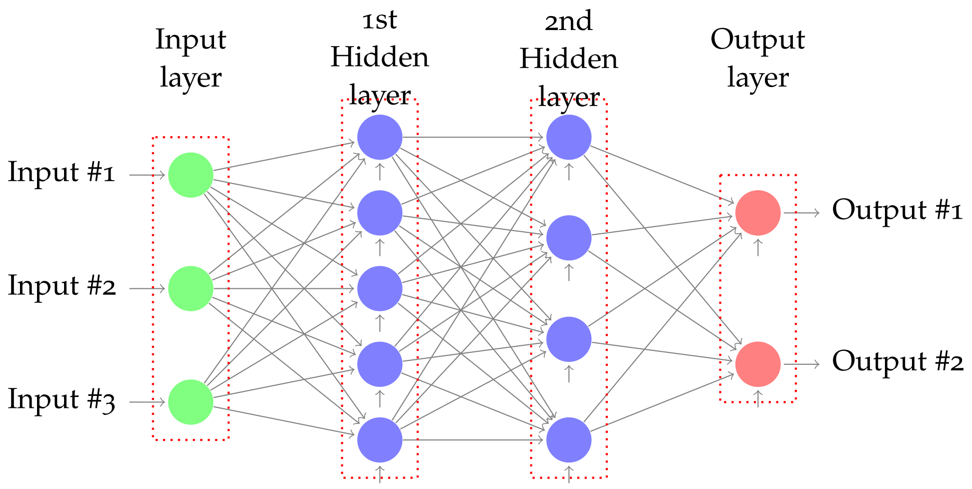
3 MODEL DESCRIPTION

3.1 Feedforward Neural Network

In supervised learning tasks, like classification and regression, feedforward neural network is widely used for prediction, and shows excellent performance. Our generalization is based on this fundamental model. If we regard each layer as a super node, then typical feedforward

1 http://deeplearning.net/software/theano/
2 https://github.com/tensorflow/tensorflow

neural network is a super node chain. Adjacent layers are completely connected, i.e. only consider the neurons of two adjacent layers, the induced graph is a complete bipartite graph.



Let L be the number of hidden layers (include the output layer). For symbolic consistency n_0 be the input dimension, and $\boldsymbol{\beta}_0$ be the input. For $l = 1, \dots, L$, set n_l be the number of neurons in hidden layer l, denote $\boldsymbol{\alpha}^{(l)}, \boldsymbol{\beta}^{(l)} \in \mathbb{R}^{n_l}$ are the linear output and activated result, let $\boldsymbol{\sigma}_l(\cdot)$ be the transfer or activation function of hidden layer l. Also, let $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}, \mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ be the bias and incoming weights of hidden layer l.

Each neuron receives the output from the previous layer, computes the activation, the biased weighted sum according to its bias and incoming edge weights, and then passes the activation through its transfer function to get its output. Hence, the feed forward process can be described by the following equations

$$\begin{aligned}
\boldsymbol{\alpha}^{(l)} &= \mathbf{W}^{(l)}\boldsymbol{\beta}^{(l-1)} + \mathbf{b}^{(l)}, \quad \forall\, l = 1, \dots, L \\
\boldsymbol{\beta}^{(l)} &= \sigma_l(\boldsymbol{\alpha}^{(l)}), \quad \forall\, l = 1, \dots, L
\end{aligned}$$

The objective loss function $\mathcal{L}(\mathbf{t}, \mathbf{y})$ measures the distance between final output $\mathbf{y} \triangleq \boldsymbol{\beta}_L$ and the target value $\mathbf{t}$. To train the neural network, back-propagation algorithm is applied:

$$\begin{aligned}
\delta_i^{(L)} &\triangleq \frac{\partial \mathcal{L}(\mathbf{t}, \mathbf{y})}{\partial \alpha_i^{(L)}} = \frac{\partial \mathcal{L}(\mathbf{t}, \mathbf{y})}{\partial \beta_i^{(L)}} \frac{\partial \beta_i^{(L)}}{\partial \alpha_i^{(L)}}, \quad \forall\, i = 1, \dots, n_L \\
\delta_i^{(l)} &\triangleq \frac{\partial \mathcal{L}(\mathbf{t}, \mathbf{y})}{\partial \alpha_i^{(l)}} = \sum_{j=1}^{n_{l+1}} \frac{\partial \mathcal{L}(\mathbf{t}, \mathbf{y})}{\partial \alpha_j^{(l+1)}} \frac{\partial \alpha_j^{(l+1)}}{\partial \beta_i^{(l)}} \frac{\partial \beta_i^{(l)}}{\partial \alpha_i^{(l)}}, \quad \forall\, l = 1, \dots, L-1 \text{ and } i = 1, \dots, n_l
\end{aligned}$$

The above equations can be written into matrix form, where $\odot$ operator denotes matrix element-wise multiplication

$$\begin{aligned}
\boldsymbol{\delta}^{(L)} &= \frac{\partial \mathcal{L}(\mathbf{t}, \mathbf{y})}{\partial \boldsymbol{\beta}^{(L)}} \odot \sigma_l'(\boldsymbol{\alpha}^{(L)}) \\
\boldsymbol{\delta}^{(l)} &= \sigma_l'(\boldsymbol{\alpha}^{(l)}) \odot \left(\left[\mathbf{W}^{(l+1)} \right]^T \boldsymbol{\delta}^{(l+1)} \right), \quad \forall\, l = 1, \dots, L-1
\end{aligned}$$

Finally, compute the partial derivative w.r.t. weights and bias by

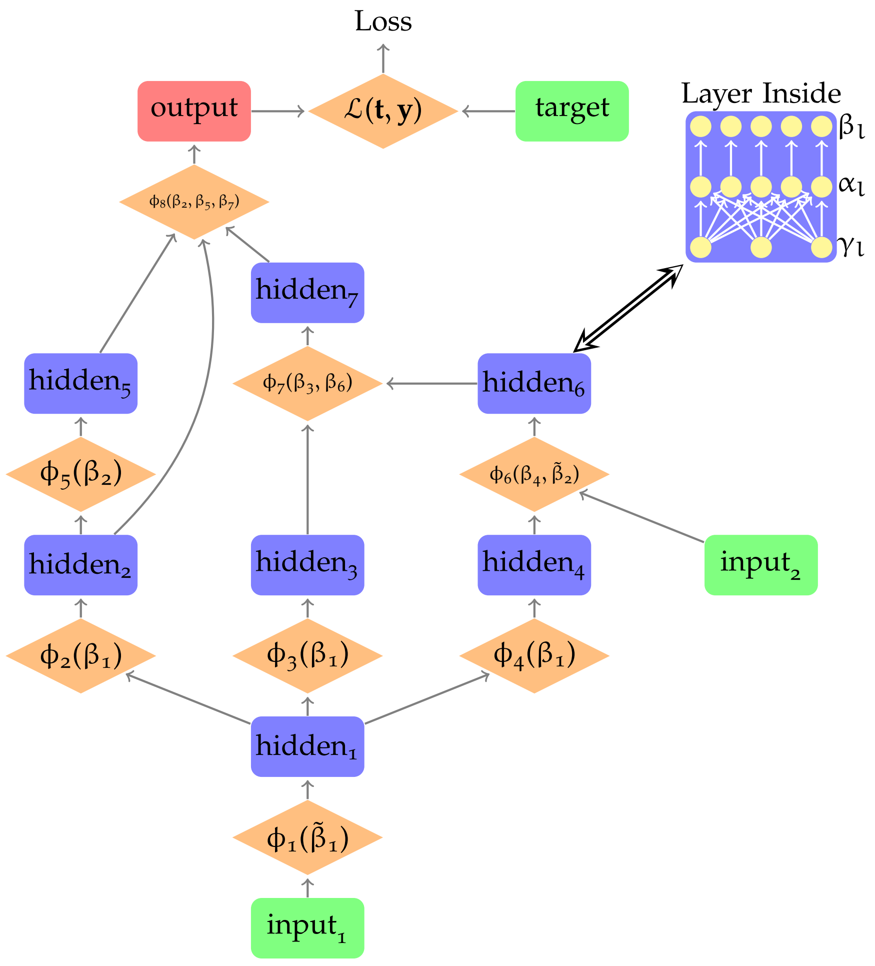
$$\frac{\partial \mathcal{L}(\mathbf{t}, \mathbf{y})}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)} \left[\boldsymbol{\beta}^{(l-1)} \right]^T, \quad \frac{\partial \mathcal{L}(\mathbf{t}, \mathbf{y})}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}, \quad \forall\, l = 1, \dots, L$$

3.2 DRAGON model

In this section, we describe the extended DRAGON model in details. If considering the model in the perspective of super node (hidden layer), the structure of neural network is a directed acyclic graph instead of a chain. Each super node represents a layer (may be input layer, hidden layer or output layer), denoted as input node, hidden node and output node respectively. In DRAGON model, multiple input nodes and hidden nodes are possible, but only one output node is allowed. This setting makes sense since the loss function or objective function is usually unique.

Essentially, combine functions are introduced for merging output of parent nodes to generate input of its associate nodes. In other words, each hidden node and output node is associated with a combine function, taking the arguments from its parents outputs or inputs.

The following figure provides an example of DRAGON model. It can be seen that the network structure is a DAG, one hidden layer may have multiple parent layers and multiple child layers. In addition, this can be represented by a factor graph, where factor nodes are combine functions and variable nodes are layers. Within each hidden layer, the inside structure is the same as feed forward neural network: first proceeding a linear transformation and then passing the result through an activation function for current layer output.



We first define the notations of DRAGON model. Differently, we introduce the number of input sources the for notation clarity. Let $\tilde{L}$ be the number of input layers. For $i = 1, \ldots, \tilde{L}$, , denote $\tilde{n}_i$ be the i-th input dimension, and $\tilde{\beta}^{(i)}$ be the i-th input layer I_i. Similarly, let L be the number of hidden layers (include the output layer). For $l = 1, \ldots, L$, set m_l, n_l be the input dimension and the number of neurons in hidden layer l, denote $\gamma^{(l)} \in \mathbb{R}^{m_l}, \alpha^{(l)}, \beta^{(l)} \in \mathbb{R}^{n_l}$

are the input, activation and output, let $\sigma_l(\cdot)$ be the transfer function of l-th hidden layer H_l. Also, let $\mathbf{b}^{(l)} \in \mathbb{R}^{n_l}, \mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times n_{l-1}}$ be the bias and incoming weights of hidden layer l.

To concretely describe the DAG structure, for $l = 1, \ldots, L$ define $\mathsf{Pa}(l), \mathsf{Ch}(l)$ as the set of parent layers and child layers with respect to hidden layer l, let ϕ_l be the combine function with arguments of parent layers outputs.

$$\begin{aligned}
\mathsf{Pa}(l) &= \left\{ H_k \,\middle|\, \boldsymbol{\beta}^{(k)} \text{ is one of } \phi_l\text{'s arguments}, k \in \{1, \ldots, L\} \right\} \\
&\quad \cup \left\{ I_i \,\middle|\, \tilde{\boldsymbol{\beta}}^{(i)} \text{ is one of } \phi_l\text{'s arguments}, i \in \left\{1, \ldots, \tilde{L}\right\} \right\} \\
\mathsf{Ch}(l) &= \left\{ H_r \,\middle|\, \boldsymbol{\beta}^{(l)} \text{ is one of } \phi_r\text{'s arguments}, r \in \{1, \ldots, L\} \right\}
\end{aligned}$$

Since the combine function are not necessary to be symmetric, the order of the input arguments matters. For simplicity, we define $\boldsymbol{\beta}_{\mathsf{Pa}(l)}$ as the output tuple of H_l's parent layers, with the order rearranged according to the arguments of ϕ_l; particularly, $\boldsymbol{\beta}_{\mathsf{Pa}(l)}$ includes the layer outputs

$$\left\{ \boldsymbol{\beta}^{(k)} \,\middle|\, H_k \in \mathsf{Pa}(l) \right\} \cup \left\{ \tilde{\boldsymbol{\beta}}^{(i)} \,\middle|\, I_i \in \mathsf{Pa}(l) \right\}$$

For the feed forward process, each hidden layer first receives outputs of all its parent layers and passes them through the combine function to get its input, then the rest is same as typical neural network. Hence, the feed forward process can be described by the following equations

$$\begin{aligned}
\boldsymbol{\gamma}^{(l)} &= \phi_l(\boldsymbol{\beta}_{\mathsf{Pa}(l)}), \quad \forall\, l = 1, \ldots, L \\
\boldsymbol{\alpha}^{(l)} &= \mathbf{W}^{(l)} \boldsymbol{\gamma}^{(l)} + \mathbf{b}^{(l)}, \quad \forall\, l = 1, \ldots, L \\
\boldsymbol{\beta}^{(l)} &= \sigma_l(\boldsymbol{\alpha}^{(l)}), \quad \forall\, l = 1, \ldots, L
\end{aligned}$$

The objective loss function $\mathcal{L}(\mathbf{t}, \mathbf{y})$ measures the distance between final output $\mathbf{y} \triangleq \boldsymbol{\beta}_L$ and the target value $\mathbf{t}$. To train the neural network, the same back-propagation algorithm is used. However, because of the DAG structure it becomes more complicated than the typical case:

$$\begin{aligned}
\delta_i^{(L)} &\triangleq \frac{\partial \mathcal{L}(\mathbf{t}, \mathbf{y})}{\partial \alpha_i^{(L)}} = \frac{\partial \mathcal{L}(\mathbf{t}, \mathbf{y})}{\partial \beta_i^{(L)}} \frac{\partial \beta_i^{(L)}}{\partial \alpha_i^{(L)}}, \quad \forall\, i = 1, \ldots, n_L \\
\delta_i^{(l)} &\triangleq \frac{\partial \mathcal{L}(\mathbf{t}, \mathbf{y})}{\partial \alpha_i^{(l)}} = \sum_{c \in \mathsf{Ch}(l)} \sum_{j=1}^{n_c} \frac{\partial \mathcal{L}(\mathbf{t}, \mathbf{y})}{\partial \alpha_j^{(c)}} \frac{\partial \alpha_j^{(c)}}{\partial \gamma_j^{(c)}} \frac{\partial \gamma_j^{(c)}}{\partial \beta_i^{(l)}} \frac{\partial \beta_i^{(l)}}{\partial \alpha_i^{(l)}}, \quad \forall\, l = 1, \ldots, L-1 \text{ and } i = 1, \ldots, n_l
\end{aligned}$$

The above equations can be written into matrix form, where $\odot$ operator denotes matrix element-wise multiplication

$$\begin{aligned}
\boldsymbol{\delta}^{(L)} &= \frac{\partial \mathcal{L}(\mathbf{t}, \mathbf{y})}{\partial \boldsymbol{\beta}^{(L)}} \odot \sigma_l'(\boldsymbol{\alpha}^{(L)}) \\
\boldsymbol{\delta}^{(l)} &= \sigma_l'(\boldsymbol{\alpha}^{(l)}) \odot \sum_{c \in \mathsf{Ch}(l)} \left(\left[\mathbf{W}^{(c)} \right]^T \boldsymbol{\delta}^{(c)} \odot \frac{\partial \phi_c(\boldsymbol{\beta}_{\mathsf{Pa}(c)})}{\partial \boldsymbol{\beta}^{(l)}} \right), \quad \forall\, l = 1, \ldots, L-1
\end{aligned}$$

Finally, compute the partial derivative w.r.t. weights and bias by

$$\frac{\partial \mathcal{L}(\mathbf{t}, \mathbf{y})}{\partial \mathbf{W}^{(l)}} = \boldsymbol{\delta}^{(l)} \left[\boldsymbol{\gamma}^{(l)} \right]^T, \quad \frac{\partial \mathcal{L}(\mathbf{t}, \mathbf{y})}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}, \quad \forall\, l = 1, \ldots, L$$

There are two cases for the output layer in the DRAGON model and are treated differently in back-propagation algorithm:

- Pure Output Layer: The network output is equivalent as $\boldsymbol{\gamma}^{(L)}$, in this case $\mathbf{W}^{(L)}$ is identity matrix and $\mathbf{b}^{(L)}$ is all zero. Although this layer is trivial and the network output can be obtained once computation in ϕ_L is finished, we add the dummy layer for structure consistency. In back-propagation $\mathbf{W}^{(L)}$ and $\mathbf{b}^{(L)}$ are not updated and remain unchanged.

- Hidden Output Layer: The network output is not equivalent as $\boldsymbol{\gamma}^{(L)}$. In back-propagation $\mathbf{W}^{(L)}$ and $\mathbf{b}^{(L)}$ are updated.

One big advantage of DRAGON model is that the combine function can be random sampling, for example $\phi_l(a, b) = z$ where $z \sim \mathcal{N}(a, b^2)$, which can be naturally embedded into many non-deterministic setups. It is worth noting that if the DAG is chain structure and all combine functions are identity function, the DRAGON model degenerates into typical neural network.

3.3 Batched Stochastic Gradient Descent (B-SGD) Algorithm

Except for parallelism of matrix operation, another scalable trick can be batched stochastic gradient descent for optimization. Since neural networks are usually trained on large dataset for less overfitting performance, computation for gradients or back-propagation on whole dataset is extremely time and space consuming. Thus, we take the advantage of stochastic gradient descent with mini-batch setting. Basically, it means for each feed-forward and back-propagation iteration, only part of samples are evaluated for computing gradients, where the samples are commonly randomly drawn from the whole dataset. In general, the B_SGD algorithm follows the pseudo-code below:

```
Input: training data $\mathcal{D} = \left\{(x^{(1),i}, \dots, x^{(\tilde{L}),i})\right\}_{i=1}^{N}$, target $\mathcal{T} = \left\{t^i\right\}_{i=1}^{N}$, max epoch number $E$
Output: Trained network after $T$ epoch using training data and target
Initialize all weights randomly, usually normal with zero-mean and tiny variance;
Let batch size to be $B$, then the number of batch $n_B = \lceil N/B \rceil$;
for $\ell = 1, \dots, E$ do
    Permute and divide training data and corresponding into batches $\{(\mathcal{D}_r, \mathcal{T}_r)\}_{r=1}^{n_B}$;
    for $r = 1, \dots, n_B$ do
        Apply feed-forward according to $\mathcal{D}_r$ to train to compute batched output $\mathcal{Y}_r$;
        Apply back-propagation according $\mathcal{T}_r, \mathcal{Y}_r$ to compute partial derivative $\left(\frac{\partial \mathcal{L}}{\partial \theta}\right)_{\ell,r}$;
        Compute the update step size $\epsilon_{\ell,r}$;
        Update every parameter by $\theta \leftarrow \theta - \epsilon_{\ell,r}(\theta) \cdot \left(\frac{\partial \mathcal{L}}{\partial \theta}\right)_{\ell,r}$;
    end
end
```

The update of scalar θ means for every element in $\mathbf{W}$ and $\mathbf{b}$, we follow the same update formula. Notice that the step size $\epsilon_{\ell,r}$ can be manually set to satisfy decreasing to infinitesimal. Alternatively, we implemented the adaptive step size B_SGD, i.e.,

$$\epsilon_{\ell,r}(\theta) = \frac{\epsilon}{\sqrt{\sum_{u=1}^{\ell-1} \sum_{v=1}^{r} \left[\left(\frac{\partial \mathcal{L}}{\partial \theta}\right)_{u,v}\right]^2}}$$

where $\left(\frac{\partial \mathcal{L}}{\partial \theta}\right)_{u,v}$ is the cumulative gradients to current epoch u and iteration v, with respect to parameter θ, and ϵ is a fixed and predefined learning rate. This setting makes less tuning requirement for the step size, allowing less cross validation computation.

4 IMPLEMENTATION DETAILS

4.1 Serial/OpenMP Implementation

For serial and OpenMP [?] versions, we implemented from the very bottom. The primary classes and the parallelism patterns are listed below:

- `BatchData` Class: basic row vector for batch input and output of `Neuron`, the arithmetic operation and basic math functions are paralleled by SIMD.
- `Neuron` Class: contains all weights and bias information corresponding to a neuron. A neuron in hidden layer H_l accepts m_l `BatchData` as input and generate one `BatchData` as neuron activation.

$$\boldsymbol{\alpha}_i^{(l)} = \sum_{j=1}^{m_l} \mathbf{W}_{ij}^{(l)} \boldsymbol{\gamma}_j^{(l)} + \mathbf{b}_i^{(l)}, \quad \forall i = 1, \ldots, n_l$$

  From the above equation we can see reduction pattern can be applied for this case.

- `Layer` Class: contains vector of `Neurons`, use map pattern to compute layer output from activation by $\boldsymbol{\beta}_i^{(l)} = \sigma_l(\boldsymbol{\alpha}_i^{(l)})$. Hence, the hidden layer H_l's output includes n_l `BatchData`.
- `Network` Class: A DAG where each node is `Layer`, layers can be computed in parallel under direct based parallelism. All divide, feed-forward and back-propagation subroutine are implemented inside `Network` class. Thus, folk-join and pipeline pattern are suitable for this case, however the parallelism at this scale is not implement in this project.

4.2 CUDA Implementation

After finishing OpenMP version, we found that it is still much slower than MATLAB, the main reason is that our self-implemented matrix operation is not as efficient as BLAS library. Hence, in CUDA version cuBLAS library is used for matrix operation to achieve best performance. The primary classes and the parallelism patterns are listed below:

- Thrust Library [?]: use `thrust::device_vector` as basic data structure. There are three advantage to use Thrust library than writing kernel functions directly. The first is that Thrust is objective-oriented and is more compatible with C++ language. The second is that by using Thrust one can get rid of massive device memory allocation and free code, which makes the code much more convenient and concise. The third is that unlike a kernel call, Thrust does not require programmer to specify the grid dimension and block dimension. Alternatively, it automatically chooses these best granularity parameters.
- `Layer` Class: since cuBLAS library is used for matrix operation, `batchData` and `Neuron` classes are obsoleted. Instead, all weights and bias are stored in `Layer` class as matrices directly. Map pattern is implemented by `thrust::for_each` and `thrust::transform`, matrix multiplication applies `cuBlasSgemm`.
- `Network` Class: A DAG where each node is `Layer`, layers can be computed in parallel under direct based parallelism. Map pattern is implemented by `thrust::for_each` and `thrust::transform`, reduce pattern is implemented by `thrust::inner_product`, matrix multiplication / addition applies `cuBlasSgemm` / `cuBlasSgeam`.
- Future Work: Implement directed based parallelism using CUDA streams.

Besides, in back-propagation process, once the computation for $\delta^{(l)}$ is finished, then $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}}, \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}}$ and $\delta^{(r)}$ where $r \in \mathsf{Pa}(l)$ can be computed simultaneously. Our guess is that the code may run 30% to 40% faster if this parallelism is implemented.

5 DISCUSSION AND RESULTS

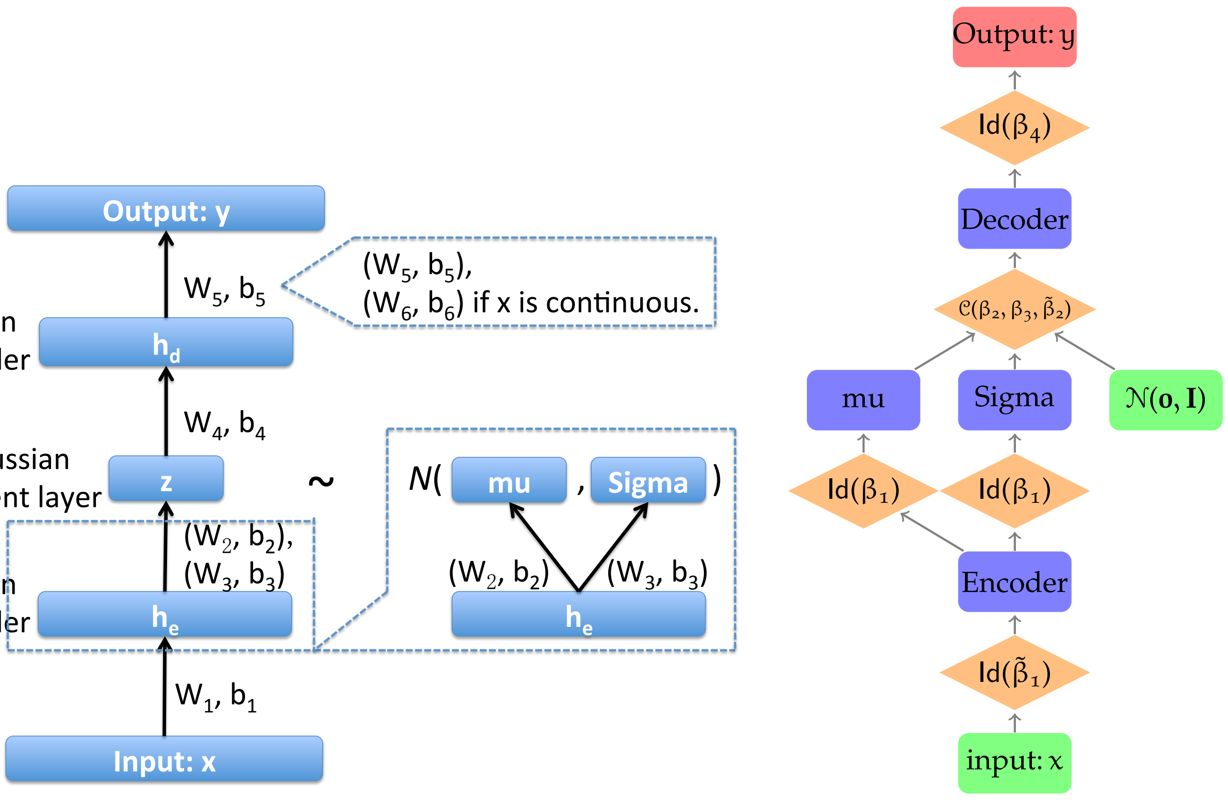


Figure 1: Left panel is the variational auto-encoder model. Right panel is its equivalent DRAGON model representation, where combine function $\mathcal{C}(\mu, \sigma, \epsilon) = \mu + \sigma\epsilon$ with $\epsilon \sim \mathcal{N}(0, 1)$.

The model we consider is a probabilistic neural networks, i.e. variational auto-encoder. The difference from standard neural networks is one of its hidden layer $\mathbf{z}$ is Gaussian random variables; in other words or from the perspective of DRAGON model, the Gaussian latent layer is the output of combine function including three arguments, where one of its arguments is from another input source.

The dataset we used is MNIST images, including 70,000 handwriting digits, 10,000 of which are used for testing. Each data is a 28×28 image, i.e., 784 dimensional inputs in the neural networks. The auto-encoder structure is $784 - 400 - (20, 20) - 400 - 784$, where the number of parameters is approximately 660,000. The mini-batch size for stochastic gradient training is 100. The predefined learning rate for adaptive SGD is 0.01. The platform or server to run our program is GNU/Linux 3.16.0-50-generic x86_64, 16 cores and 32 Processors(CPUs, two packages of Intel(R) Xeon(R) E5-2640 v3), Tesla K40C 12GB GPU.

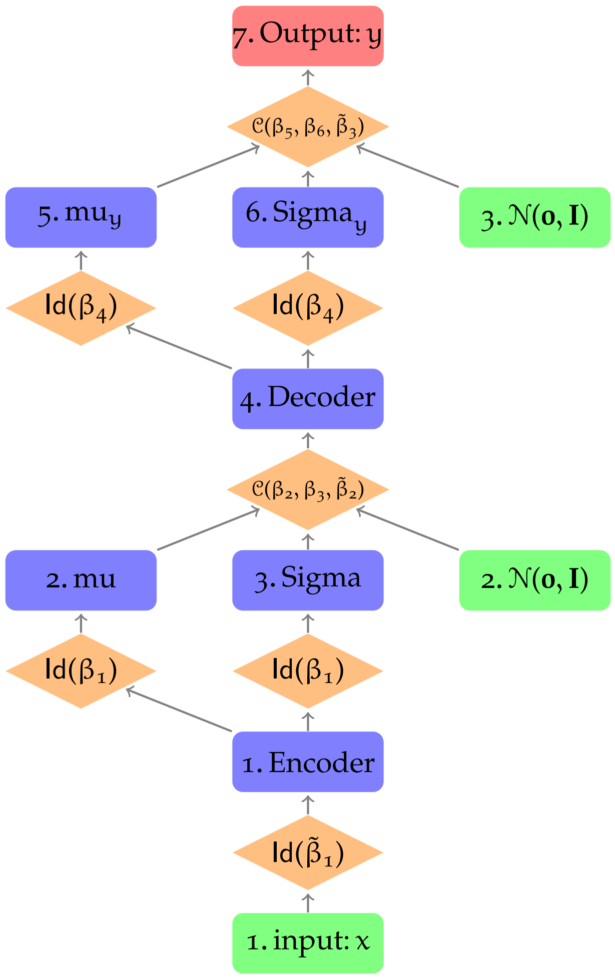
Serial C++	OpenMP C++	Matlab (personal)	CUDA C++ (personal)	CUDA C++
39s	7.5s	2.3s	0.72s	0.2s

where Matlab program is specifically implemented in personal laptop (CPU 2.5 GHz, RAM 16GB, 1600 MHz DDR, GPU 2GB NVIDIA GeForce GT 750M).

6 CONCLUSION

In this work, we manage to implement a GPU based parallel computing package for a more general neural networks. In our experimental results, the CUDA C implementation can achieve 3x speed up compared with Matlab on personal laptop GPU, even 10x speed up in Tesla K40C. One possible future work would be the fork-join pattern implementation or CUDA streams. Another one may rewrite OpmenMP version with MKL (Math Kernel Library).

APPENDIX: USER GUIDE FOR DRAGON LIBRARY


7. Output: y
$\mathcal{C}(\beta_5, \beta_6, \tilde{\beta}_3)$
5. mu$_y$
6. Sigma$_y$
3. $\mathcal{N}(\mathbf{0}, \mathbf{I})$
$\mathrm{Id}(\beta_4)$
$\mathrm{Id}(\beta_4)$
4. Decoder
$\mathcal{C}(\beta_2, \beta_3, \tilde{\beta}_2)$
2. mu
3. Sigma
2. $\mathcal{N}(\mathbf{0}, \mathbf{I})$
$\mathrm{Id}(\beta_1)$
$\mathrm{Id}(\beta_1)$
1. Encoder
$\mathrm{Id}(\tilde{\beta}_1)$
1. input: x

In the rest of this user guide we will use above structure as an example, variational auto-encoder with continuous inputs and outputs. Take MNIST as the example, the structure would be $784 - 400 - (20, 20) - 400 - (784, 784) - 784$. The corresponding DRAGON model is represented above. The most different layer is the output, which is also randomly drawn from Gaussian distribution. In this case, the final output layer is pure output layer, i.e., no weights and bias associated with this layer.

A. Network Construction

To construct a DRAGON neural network, user needs to provide all graphical structure and functional informations

```
neuralNetworkEx(vector<int> inputDims,
                vector<string> inputDists,
                vector<int> layerInputDims,
                vector<int> numNodesPerLayer,
                int _batchSize,
                vector<string> trsFuncName,
                vector<string> cmbFuncName,
                vector<vector<int>> I2L_edges,
                vector<vector<int>> L2L_edges,
                string objFuncName,
                float _epsilon,
                bool _lastLayerAsOutput)
```

`inputDims`: the input dimensions of all input layers, i.e. $\tilde{n}_i$ for $i = 1, \ldots, \tilde{L}$.

`inputDists`: the distribution of each input source, mainly used for random combine functions. If the input is deterministic, use `'fixed'` type, otherwise give the distribution name of random input (currently only support `'gaussian'` and `'uniform'`, user may implement other distributions).

`layerInputDims`: input dimension of each layer, i.e. m_l for $l = 1, \ldots, L$.

`numNodesPerLayer`: input dimension of each layer, i.e. n_l for $l = 1, \ldots, L$.

`batchSize`: the batch size in batch training, usually chosen to be 100.

`trsFuncName`: transfer function name of each layer, i.e. names for σ_l for $l = 1, \ldots, L$.

`cmbFuncName`: combine function name of each layer, i.e. names for ϕ_l for $l = 1, \ldots, L$.

`I2L_edges`: edges between input layer and hidden layer, in our implementation each hidden layer can have at most one input layer as parent. Also user needs to tell the input layer corresponds to which input argument of its child layer's combine function, by default it is the first input argument.

`L2L_edges`: edges between hidden layers, edges connected to one combine function must be added in the same order of its input arguments.

`objFuncName`: objective function name i.e. name for $\mathcal{L}(\mathbf{t}, \mathbf{y})$.

`_epsilon`: learning rate, usually chosen to be 0.01.

`_lastLayerAsOutput`: indicator for whether last layer is the pure output layer.

Example code for construct a DRAGON network:

```
1  vector<int> srcDim({784, 20, 784});
2  vector<string> srcDist({"fixed","gaussian","gaussian"});
3  vector<int> inputDim({784, 400, 400, 20, 40, 40, 784});
4  vector<int> numNodes({400, 20, 20, 40, 784, 784, 784});
5  int batchSize = 100;
6  vector<string> ...
       trsFuncName({"tanh","tanh","identity","tanh","tanh","identity","sigmoid"});
7  vector<string> ...
       cmbFuncName({"null","null","null","NormRnd","null","null","NormRnd"});
8  string objFuncName = "CrossEntropy";
9  vector<vector<int>> I2L_edges({{0,0},{1,3,3},{2,6,3}});
10 vector<vector<int>> ...
       L2L_edges({{0,1},{0,2},{1,3},{2,3},{3,4},{3,5},{4,6},{5,6}});
11 float epsilon = 0.01;
12 bool lastLayerAsOutput = true;
13 neuralNetworkEx nn(srcDim, srcDist, inputDim, numNodes, batchSize, ...
       trsFuncName, cmbFuncName,
14  I2L_edges, L2L_edges, objFuncName, epsilon, lastLayerAsOutput);
```

Line 1 and 2 shows that there are three input sources, the first is fixed and the rest are Gaussian random.

Line 3 and 4 gives the input dimension and output dimension of every hidden layer.

Line 6 and 7 determines what transfer function and combine function are used for each hidden layer.

Line 8 provides the name of the objective function minimized in the back-propagation process.

Aware of the index starts from 0 in C++, line 9 tells that

1. The first hidden layer H_1 takes the first input layer I_1 as a parent layer, and $\tilde{\beta}^{(1)}$ is the first input argument of ϕ_1 (actually it is the unique input argument).
2. H_4 takes I_2 as a parent layer, and $\tilde{\beta}^{(2)}$ is the third input argument of ϕ_4.
3. H_7 takes I_3 as a parent layer, and $\tilde{\beta}^{(3)}$ is the third input argument of ϕ_7.

Similarly, line 10 tells that

1. H_2 and H_3 both has only one parent layer H_1.
2. H_2 and H_3 are parent layers of H_4. Note I_2 is also a parent layer of H_4 and uses the third argument of ϕ_4 and the rule that edges between hidden layers must be added in the same order of combine function's input arguments. Then it can be concluded that $\beta^{(2)}, \beta^{(3)}$ are the first and second input argument of ϕ_4.
3. H_5 and H_6 both has only one parent layer H_4.
4. H_5 and H_6 are parent layers of H_7. Similar with the case of H_4, $\beta^{(5)}, \beta^{(6)}$ are the first and second input argument of ϕ_7.

Line 12 tells that H_7 is a pure output layer, thus $\mathbf{W}^{(7)}$ is identity and $\mathbf{b}^{(7)}$ is all zero, we only use the result of ϕ_7 as network output.

B. Input Data Construction

Since DRAGON model accepts multiple input layers, the user needs to construct the input before using the neural network. The format of the input data is a 2D array of vectors: the first dimension is input source index (from 1 to $\tilde{L}$), the second dimension is index of samples (from 1 to N). The (i, n)-th element is a vector with length $\tilde{n}_i$ for deterministic input and length 0 for random input (since random inputs are generated on fly in divide process).

Here is the example code for constructing input data

```
1 vector<vector<vector<float>>> trainInput({train_image, ...
      vector<vector<float>>(train_image.size()), ...
      vector<vector<float>>(train_image.size())});
2 vector<vector<float>> trainTarget(train_image);
3 vector<vector<vector<float>>> testInput({test_image, ...
      vector<vector<float>>(test_image.size()), ...
      vector<vector<float>>(test_image.size())});
4 vector<vector<float>> testTarget(test_image);
```

Line 1 shows that the first input source is training images, the second and third input are random input (actually is Gaussian as previous section mentioned).

Line 2 shows that the target uses same training images since the example is a generative model and we want the final output of the network to be similar as input.

Line 3 and line 4 just simply do the same work as the first two lines.

Once both network and input data are correctly constructed, the user can easily train the DRAGON network and utilize the trained network by

```
int numEpoch = 100;
bool print_info = false;
nn.train(numEpoch, trainInput, trainTarget, testInput, testTarget, print_info);
vector<vector<float>> output = nn.apply(testInput, print_info);
```

C. User Extension

In this project, our DRAGON library supports two kinds of user extension. Firstly, user can create their own functions (transfer function, combine function and objective function). Secondly, user can write their own child classes inherited from our neural network class for some more specific tasks.

C1. Customize Function

Since all functions are used on device, they must have `__device__` modified in front of them. To obtain the device function pointers, a kernel called `get_address_of_device_(TYPE)_function` (here `TYPE` can be `trs`, `cmb` or `obj`) is used to store these device function pointers into an array for future use.

Also, a map type class named `(TYPE)FunctionMap` is created for finding function pointers given its name and vice versa. Therefore, customize functions must be registered in both the kernel and function map.

Therefore, the general procedure for customized function is

1. Implement the function and it (partial) derivatives with `__device__` modifier at head.
2. Register the function and all its derivatives in the kernel `get_address_of_device_(TYPE)_function`.
3. Register the function and all its derivatives in the `(TYPE)FunctionMap`.

C2. Class Inheritance

When using DRAGON model for some more specific tasks, it is better to write a subclass for code reuse. For example, after training process of auto-encoder, usually the lower half (encoder) and upper half (decoder) are used separately.

Another example is that in classifier, what we really care about is the error rate of the prediction, however this cannot be implemented by map + reduce (actually it is reduce-by-key + reduce). Hence, it is impossible to write an objective function directly computes the error rate and writing a subclass to do extra work is preferred.

For more implementation details, please refer to `Classifier.h` and `AutoEncoder.h`.