GPU-Accelerated Deep Neural Networks in TMVA

Simon Pfreundschuh
Supervisors: Sergei V. Gleyzer, Lorenzo Moneta

Google
Summer of Code

CERN

Outline

Introduction

Motivation

- Deep learning techniques have been revolutionizing the field of machine learning.
- Their success is closely related to the development of massively parallel accelerator devices, which allow for efficient training of machine learning models.
- Deep learning techniques have successfully been applied to problems in HEP[1].

[1] `http://arxiv.org/pdf/1402.4735v2.pdf`

Motivation

- Deep learning techniques have been revolutionizing the field of machine learning.
- Their success is closely related to the development of massively parallel accelerator devices, which allow for efficient training of machine learning models.
- Deep learning techniques have successfully been applied to problems in HEP[1].

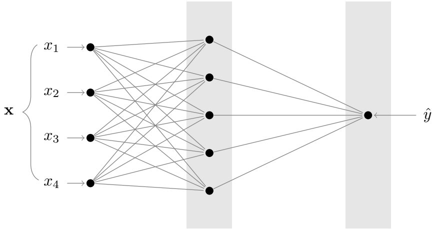
Aim
Provide an efficient and easy-to-use implementation of deep neural networks for the HEP community.

[1] `http://arxiv.org/pdf/1402.4735v2.pdf`

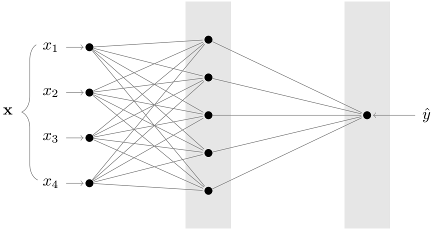
TMVA

- Toolkit for Multivariate Data Analysis with ROOT
- Root-integrated machine learning (ML) environment providing a training and test framework for a large number of ML methods.

Feed Forward Neural Networks

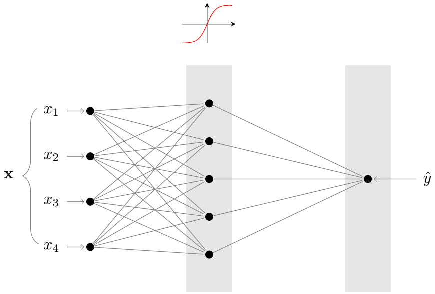


Feed Forward Neural Networks



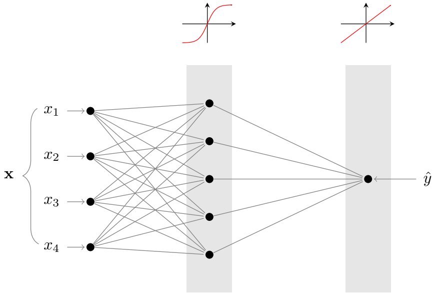
$$\mathbf{u}_1 = \quad \mathbf{W}_1\mathbf{x} + \boldsymbol{\theta}_1$$

Feed Forward Neural Networks



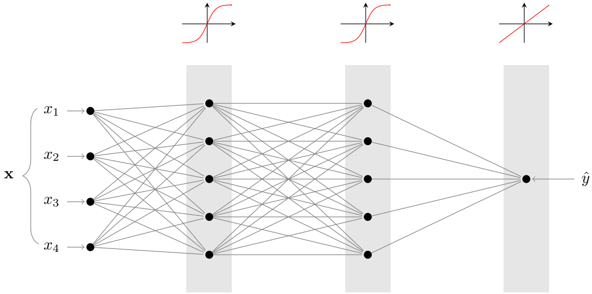
$$\mathbf{u}_1 = f\left(\mathbf{W}_1\mathbf{x} + \boldsymbol{\theta}_1\right)$$

Feed Forward Neural Networks



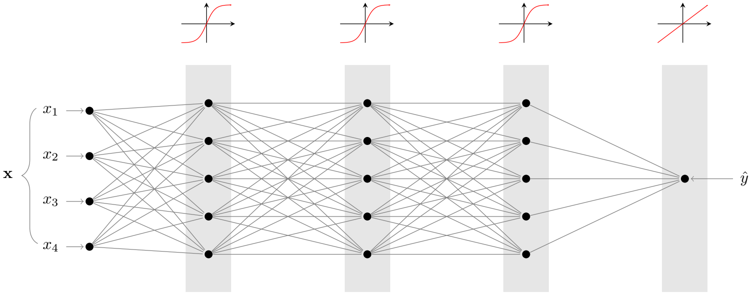
$$\mathbf{u}_1 = f\left(\mathbf{W}_1\mathbf{x} + \boldsymbol{\theta}_1\right) \quad \mathbf{u}_2 = f\left(\mathbf{W}_2\mathbf{u}_1 + \boldsymbol{\theta}_2\right)$$

Feed Forward Neural Networks



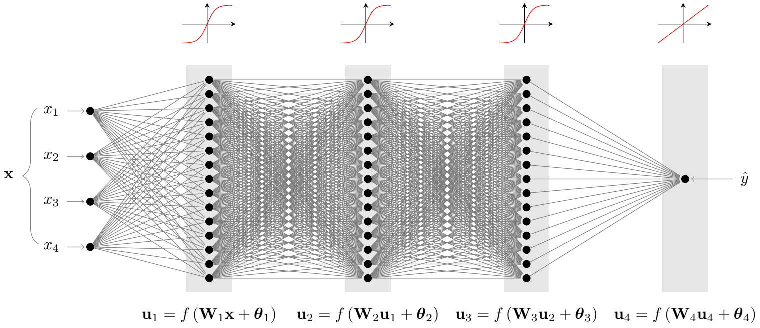
$\mathbf{u}_1 = f\left(\mathbf{W}_1\mathbf{x} + \boldsymbol{\theta}_1\right)$ $\mathbf{u}_2 = f\left(\mathbf{W}_2\mathbf{u}_1 + \boldsymbol{\theta}_2\right)$ $\mathbf{u}_3 = f\left(\mathbf{W}_3\mathbf{u}_3 + \boldsymbol{\theta}_3\right)$

Feed Forward Neural Networks



$\mathbf{u}_1 = f\left(\mathbf{W}_1\mathbf{x} + \boldsymbol{\theta}_1\right)$ $\mathbf{u}_2 = f\left(\mathbf{W}_2\mathbf{u}_1 + \boldsymbol{\theta}_2\right)$ $\mathbf{u}_3 = f\left(\mathbf{W}_3\mathbf{u}_2 + \boldsymbol{\theta}_3\right)$ $\mathbf{u}_4 = f\left(\mathbf{W}_4\mathbf{u}_4 + \boldsymbol{\theta}_4\right)$

Feed Forward Neural Networks



Feed Forward Neural Networks

- A feed forward neural network is defined by a set of layers $l = 1, \dots, n$, each with an associated weight matrix $\mathbf{W}_l$, bias terms θ_l and activation function f_l.
- **Feed forward**: Neurons of a given layer l are only connected to neurons of the layer $l + 1$
- A neural network may be viewed as a function

$$F(\mathbf{x}, \mathbf{W}, \boldsymbol{\theta}) = f_n\left(f_{n-1}(\cdots)\mathbf{W}_{n-1}^T + \boldsymbol{\theta}_{n-2}\right)\mathbf{W}_n^T + \boldsymbol{\theta}_n \qquad (1)$$

Feed Forward Neural Networks

- A feed forward neural network is defined by a set of layers $l = 1, \ldots, n$, each with an associated weight matrix $\mathbf{W}_l$, bias terms θ_l and activation function f_l.
- **Feed forward**: Neurons of a given layer l are only connected to neurons of the layer $l + 1$
- A neural network may be viewed as a function

$$F(\mathbf{x}, \mathbf{W}, \boldsymbol{\theta}) = f_n \left(f_{n-1}(\cdots)\mathbf{W}_{n-1}^T + \boldsymbol{\theta}_{n-2} \right) \mathbf{W}_n^T + \boldsymbol{\theta}_n \qquad (1)$$

- **Machine Learning**: Find parameters $\hat{\mathbf{W}}, \hat{\theta}$ so that $F(\mathbf{x}) = F(\mathbf{x}, \hat{\mathbf{W}}, \hat{\theta})$ approximates either a target function $G(\mathbf{x})$ (**Regression**) or a likelihood measure for a given class (**Classification**).

Neural Network Training

- **Supervised learning**: The network is trained using a training set consisting of inputs $\mathcal{X} = \mathbf{x}_0, \ldots, \mathbf{x}_n$ and outputs $\mathcal{Y} = y_0, \ldots, y_n$.
- The **loss function** or **error function** $J(y, \hat{y})$ quantifies the quality of a prediction $\hat{y}$ with respect to the expected output y.
- Learning as a minimization problem:

$$\underset{\mathbf{W},\boldsymbol{\theta}}{\text{minimize}}\ J_{\mathcal{X}} = \frac{1}{n} \sum_{\mathbf{x}} J(y, \hat{y}) \tag{2}$$

Neural Network Training (Contd.)

- Use gradient-based minimization methods to minimize the error $\sum_{\mathbf{x} \in \mathcal{X}} J(y, \hat{y})$ over the training set:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \frac{dJ_{\mathcal{X}}}{d\mathbf{W}} \tag{3}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{dJ_{\mathcal{X}}}{d\boldsymbol{\theta}} \tag{4}$$

- **Batch gradient descent**: Instead of the whole training set, compute the gradient only for a small subset of it.
- Crucial for scalable training on large data sets.

Forward and Backward Propagation

Forward Propagation:

$$\mathbf{U}_n = f_n\left(\mathbf{U}_{n-1}\mathbf{W}_n + \boldsymbol{\theta}^T\right) \tag{5}$$

$$\mathbf{f}'_n = f'_n\left(\mathbf{U}_{n-1}\mathbf{W}_n + \boldsymbol{\theta}^T\right) \tag{6}$$

Backward Propagation:

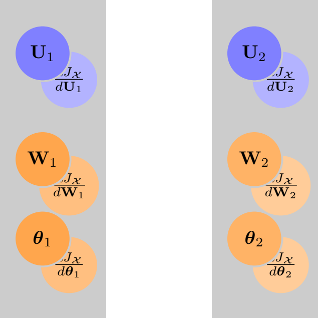
$$\frac{dJ_{\mathcal{X}}}{d\mathbf{W}_n} = \left(\mathbf{f}'_n \odot \frac{dJ_{\mathcal{X}}}{d\mathbf{U}_n}\right)^T \mathbf{U}_{n-1} \tag{7}$$

$$\frac{dJ_{\mathcal{X}}}{d\boldsymbol{\theta}_n} = \left(\mathbf{f}'_n \odot \frac{dJ_{\mathcal{X}}}{d\mathbf{U}_n}\right)^T \mathbf{1} \tag{8}$$

$$\frac{dJ_{\mathcal{X}}}{d\mathbf{U}_{n-1}} = \left(\mathbf{f}'_n \odot \frac{dJ_{\mathcal{X}}}{d\mathbf{U}_n}\right) \mathbf{W}_n \tag{9}$$

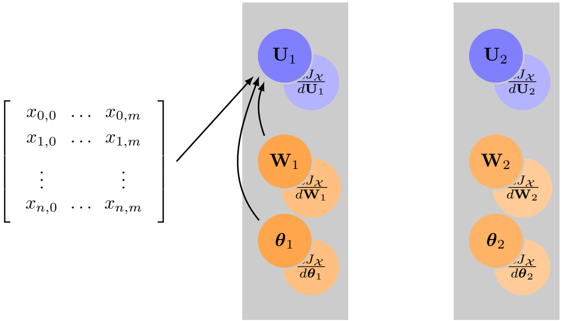
Forward and Backward Propagation

$$\begin{bmatrix} x_{0,0} & \dots & x_{0,m} \\ x_{1,0} & \dots & x_{1,m} \\ \vdots & & \vdots \\ x_{n,0} & \dots & x_{n,m} \end{bmatrix}$$



Forward and Backward Propagation

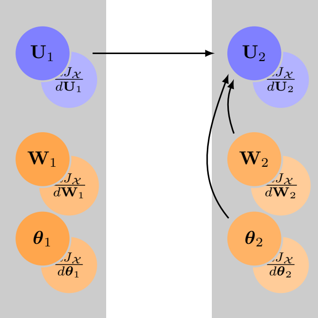
$$\mathbf{U}_1 = f_1\left(\mathbf{X}\mathbf{W}_1^T + \theta_1\right)$$



Forward and Backward Propagation

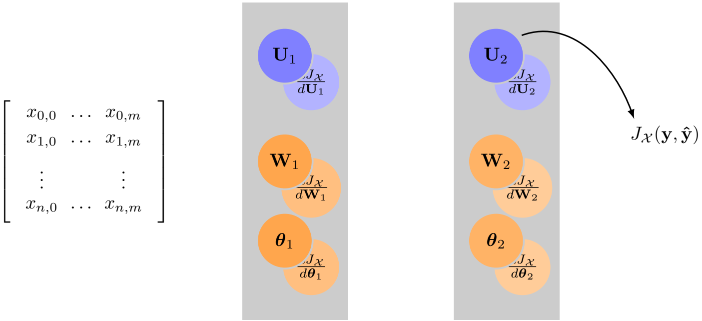
$$\mathbf{U}_1 = f_1\left(\mathbf{X}\mathbf{W}_1^T + \boldsymbol{\theta}_1\right) \qquad \mathbf{U}_2 = f_2\left(\mathbf{U_1}\mathbf{W}_2^T + \theta_2\right)$$

$$\begin{bmatrix} x_{0,0} & \dots & x_{0,m} \\ x_{1,0} & \dots & x_{1,m} \\ \vdots & & \vdots \\ x_{n,0} & \dots & x_{n,m} \end{bmatrix}$$

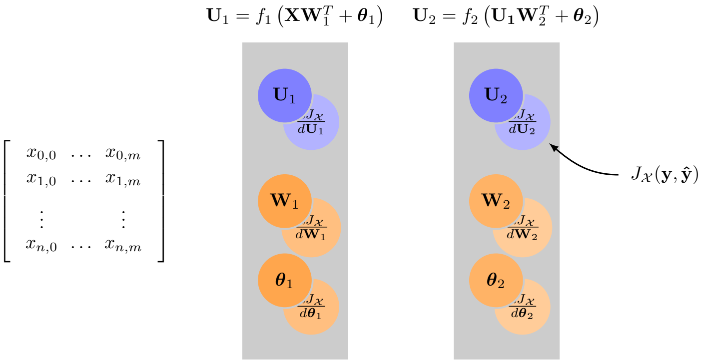


Forward and Backward Propagation

$$\mathbf{U}_1 = f_1\left(\mathbf{X}\mathbf{W}_1^T + \boldsymbol{\theta}_1\right) \quad \mathbf{U}_2 = f_2\left(\mathbf{U_1}\mathbf{W}_2^T + \boldsymbol{\theta}_2\right)$$



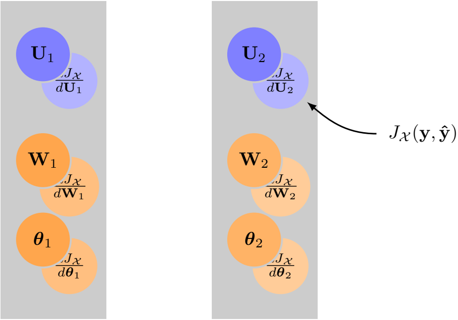
Forward and Backward Propagation



Forward and Backward Propagation

$\mathbf{U}_1 = f_1\left(\mathbf{X}\mathbf{W}_1^T + \boldsymbol{\theta}_1\right)$ $\mathbf{U}_2 = f_2\left(\mathbf{U_1}\mathbf{W}_2^T + \boldsymbol{\theta}_2\right)$

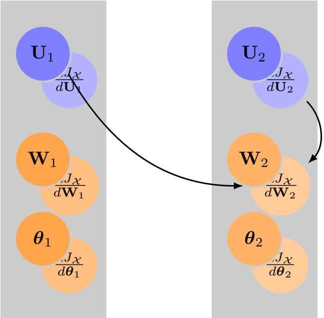
$$\begin{bmatrix} x_{0,0} & \dots & x_{0,m} \\ x_{1,0} & \dots & x_{1,m} \\ \vdots & & \vdots \\ x_{n,0} & \dots & x_{n,m} \end{bmatrix}$$



Forward and Backward Propagation

$$\mathbf{U}_1 = f_1\left(\mathbf{X}\mathbf{W}_1^T + \boldsymbol{\theta}_1\right) \qquad \mathbf{U}_2 = f_2\left(\mathbf{U_1}\mathbf{W}_2^T + \boldsymbol{\theta}_2\right)$$

$$\begin{bmatrix} x_{0,0} & \dots & x_{0,m} \\ x_{1,0} & \dots & x_{1,m} \\ \vdots & & \vdots \\ x_{n,0} & \dots & x_{n,m} \end{bmatrix}$$



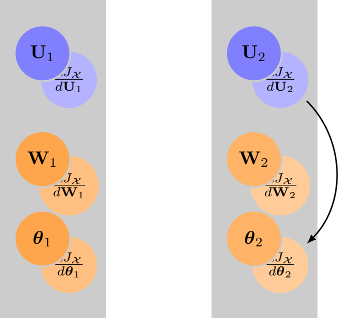
$J_{\mathcal{X}}(\mathbf{y}, \hat{\mathbf{y}})$

$$\frac{dJ_{\mathcal{X}}}{d\mathbf{W}_2} = \left(\mathbf{f}_2' \odot \frac{dJ_{\mathcal{X}}}{d\mathbf{U}_2}\right)^T \mathbf{U}_1$$

Forward and Backward Propagation

$$\mathbf{U}_1 = f_1\left(\mathbf{X}\mathbf{W}_1^T + \boldsymbol{\theta}_1\right) \qquad \mathbf{U}_2 = f_2\left(\mathbf{U_1}\mathbf{W}_2^T + \boldsymbol{\theta}_2\right)$$

$$\begin{bmatrix} x_{0,0} & \dots & x_{0,m} \\ x_{1,0} & \dots & x_{1,m} \\ \vdots & & \vdots \\ x_{n,0} & \dots & x_{n,m} \end{bmatrix}$$



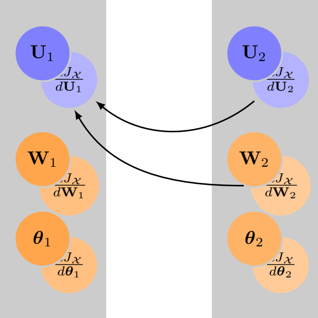
$J_{\mathcal{X}}(\mathbf{y}, \hat{\mathbf{y}})$

$$\frac{dJ_{\mathcal{X}}}{d\boldsymbol{\theta}_2} = \left(\mathbf{f}_2' \odot \frac{dJ_{\mathcal{X}}}{d\mathbf{U}_2}\right)^T \mathbf{1}$$

Forward and Backward Propagation

$$\mathbf{U}_1 = f_1\left(\mathbf{X}\mathbf{W}_1^T + \boldsymbol{\theta}_1\right) \qquad \mathbf{U}_2 = f_2\left(\mathbf{U_1}\mathbf{W}_2^T + \boldsymbol{\theta}_2\right)$$

$$\begin{bmatrix} x_{0,0} & \dots & x_{0,m} \\ x_{1,0} & \dots & x_{1,m} \\ \vdots & & \vdots \\ x_{n,0} & \dots & x_{n,m} \end{bmatrix}$$

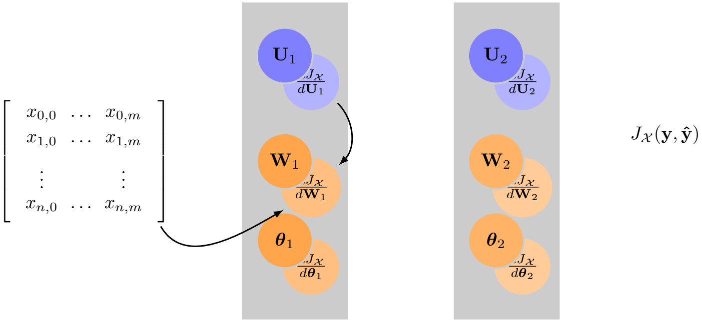


$$J_{\mathcal{X}}(\mathbf{y}, \hat{\mathbf{y}})$$

$$\frac{dJ_{\mathcal{X}}}{d\mathbf{U}_1} = \left(\mathbf{f}_2' \odot \frac{dJ_{\mathcal{X}}}{d\mathbf{U}_2}\right)\mathbf{W}_2$$

Forward and Backward Propagation

$$\mathbf{U}_1 = f_1\left(\mathbf{X}\mathbf{W}_1^T + \boldsymbol{\theta}_1\right) \qquad \mathbf{U}_2 = f_2\left(\mathbf{U_1}\mathbf{W}_2^T + \boldsymbol{\theta}_2\right)$$

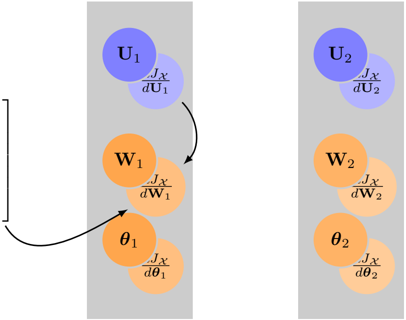


$$\frac{dJ_{\mathcal{X}}}{d\mathbf{W}_1} = \left(\mathbf{f}_1' \odot \frac{dJ_{\mathcal{X}}}{d\mathbf{U}_1}\right)^T \mathbf{X}$$

Forward and Backward Propagation

$$\mathbf{U}_1 = f_1\left(\mathbf{X}\mathbf{W}_1^T + \boldsymbol{\theta}_1\right) \qquad \mathbf{U}_2 = f_2\left(\mathbf{U_1}\mathbf{W}_2^T + \boldsymbol{\theta}_2\right)$$

$$\begin{bmatrix} x_{0,0} & \dots & x_{0,m} \\ x_{1,0} & \dots & x_{1,m} \\ \vdots & & \vdots \\ x_{n,0} & \dots & x_{n,m} \end{bmatrix}$$



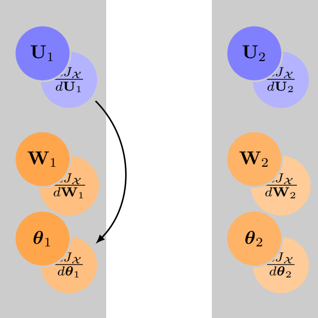
$$J_{\mathcal{X}}(\mathbf{y}, \hat{\mathbf{y}})$$

$$\frac{dJ_{\mathcal{X}}}{d\mathbf{W}_1} = \left(\mathbf{f}_1' \odot \frac{dJ_{\mathcal{X}}}{d\mathbf{U}_1}\right)^T \mathbf{X}$$

Forward and Backward Propagation

$$\mathbf{U}_1 = f_1\left(\mathbf{X}\mathbf{W}_1^T + \boldsymbol{\theta}_1\right) \qquad \mathbf{U}_2 = f_2\left(\mathbf{U_1}\mathbf{W}_2^T + \boldsymbol{\theta}_2\right)$$

$$\begin{bmatrix} x_{0,0} & \dots & x_{0,m} \\ x_{1,0} & \dots & x_{1,m} \\ \vdots & & \vdots \\ x_{n,0} & \dots & x_{n,m} \end{bmatrix}$$



$$J_{\mathcal{X}}(\mathbf{y}, \hat{\mathbf{y}})$$

$$\frac{dJ_{\mathcal{X}}}{d\boldsymbol{\theta}_1} = \left(\mathbf{f}_1' \odot \frac{dJ_{\mathcal{X}}}{d\mathbf{U}_1}\right)^T \mathbf{1}$$

Implementation

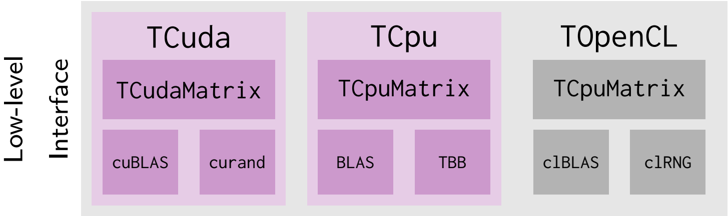
Design

- The backpropagation algorithm can be decomposed into primitive operations on matrices:
    - Matrix multiplication and addition
    - Application of activation functions
    - Computing of loss and regularization functionals
- General formulation of the backpropagation algorithm using those primitive matrix operations
- Optimized matrix operations provided by specialized low-level implementations

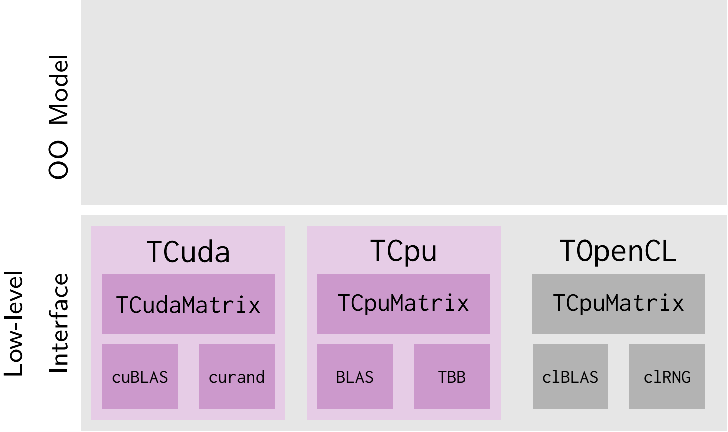
Design

Low-level
Interface

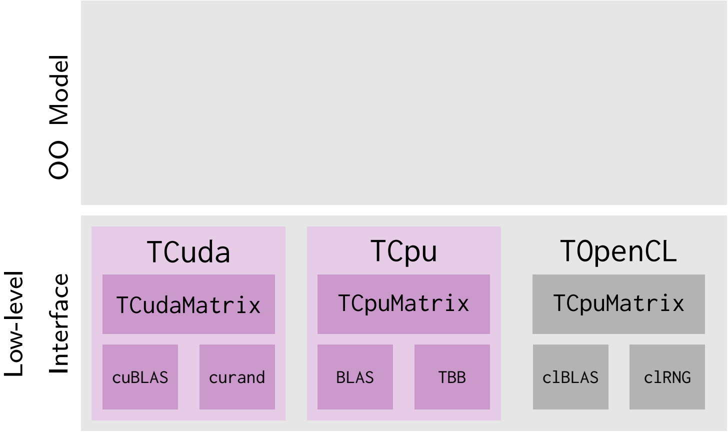
Design


Low-level
Interface
TCuda
TCudaMatrix
cuBLAS
curand
TCpu
TCpuMatrix
BLAS
TBB
TOpenCL
TCpuMatrix
clBLAS
clRNG

Design

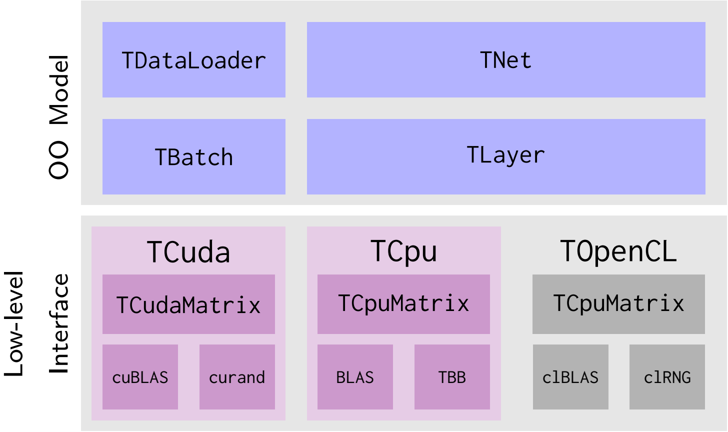

OO Model
Low-level
Interface
TCuda
TCudaMatrix
cuBLAS
curand
TCpu
TCpuMatrix
BLAS
TBB
TOpenCL
TCpuMatrix
clBLAS
clRNG

Design

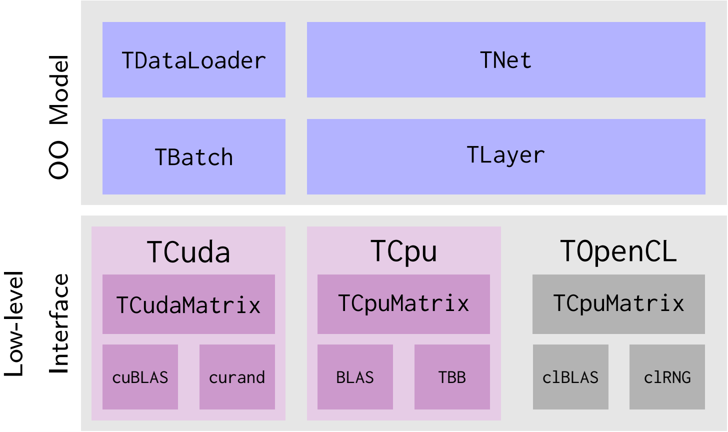


OO Model
Low-level Interface
TCuda
TCudaMatrix
cuBLAS
curand
TCpu
TCpuMatrix
BLAS
TBB
TOpenCL
TCpuMatrix
clBLAS
clRNG

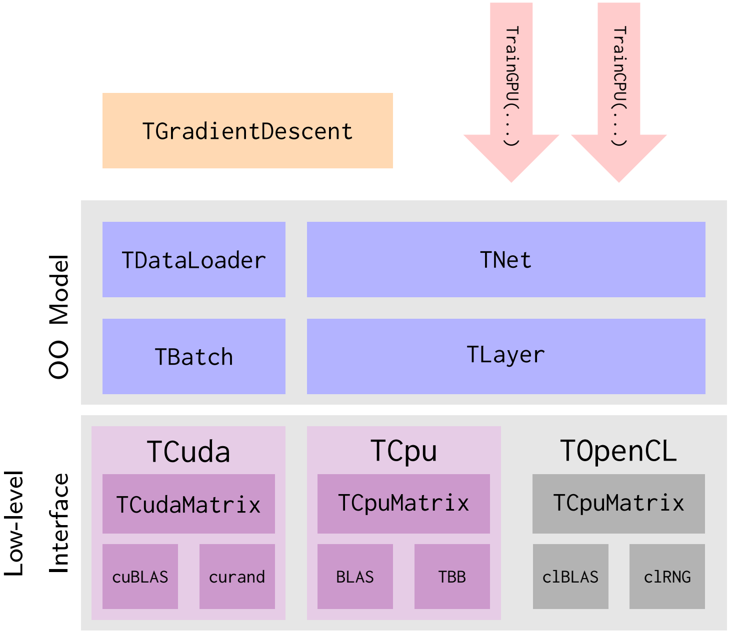
Design


OO Model
TDataLoader
TNet
TBatch
TLayer
Low-level
Interface
TCuda
TCudaMatrix
cuBLAS
curand
TCpu
TCpuMatrix
BLAS
TBB
TOpenCL
TCpuMatrix
clBLAS
clRNG

Design


OO Model
TDataLoader
TNet
TBatch
TLayer
Low-level
Interface
TCuda
TCudaMatrix
cuBLAS
curand
TCpu
TCpuMatrix
BLAS
TBB
TOpenCL
TCpuMatrix
clBLAS
clRNG

Design


TrainGPU(...)
TrainCPU(...)
TGradientDescent
OO Model
TDataLoader
TNet
TBatch
TLayer
Low-level
Interface
TCuda
TCudaMatrix
cuBLAS
curand
TCpu
TCpuMatrix
BLAS
TBB
TOpenCL
TCpuMatrix
clBLAS
clRNG

Design

The Low-Level Interface:

- Implemented by architecture classes: `TCuda, TCpu, TOpenCL`
- Architecture classes provide **matrix** and **scalar** types as well as **host** and **device** buffer types

The Object Oriented Model:

- Generic neural network implementation: Classes are templated by architecture class.
- The `TNet` class provides a general implementation of the backpropagation algorithm.
- The `TDataLoader` takes care of the streaming of data to the device.

Dependencies

CPU Implementation:

- BLAS: quasi-standard, various optimized open source implementations available, possibility to link against vendor provided implementations when available
- TBB: Considered using Root's ThreadPool, but lacks block range functionality

Dependencies

CPU Implementation:

- BLAS: quasi-standard, various optimized open source implementations available, possibility to link against vendor provided implementations when available
- TBB: Considered using Root's ThreadPool, but lacks block range functionality

CUDA Implementation:

- cuBLAS and cuRAND freely available as part of the CUDA Toolkit

Dependencies

OpenCL Implementation:

- clBLAS: Part of the open-source clMath
- clRNG: Also part of the clMath libraries
- Encountered portability problems with the clRNG library.

Verification and Testing

Verification

- The code includes a reference low-level implementation based on Root's `TMatrix` class.
- Backpropagation algorithm verified using **numerical differentiation**.
- Generic unit test for all routines in the low-level interface based on the reference implementation.
- Training routines verified by learning full-rank linear mappings.

Performance

Performance Model

Consider a layer l with n_l neurons, n_{l-1} input neurons and a batch size of n_b.

Forward Propagation:

- Multiplication of weight matrix $\mathbf{W}_l$ with activation gradients:

$$n_l n_b (2 n_{l-1} - 1) \ \text{FLOP}$$

- Addition of bias terms $\boldsymbol{\theta}_l$:

$$n_l n_b \ \text{FLOP}$$

- Application of activation function f_l and its derivatives:

$$2 n_l n_b c_f \ \text{FLOP}, \quad c_f \approx 1$$

Performance Model

Consider a layer l with n_l neurons, n_{l-1} input neurons and a batch size of n_b.

Backward Propagation

- Hadamard product:

$$n_l n_b \text{ FLOP}$$

- Computation of previous layer activations:

$$n_{l-1} n_b (2n_l - 1) \text{ FLOP}$$

- Computation of weight and bias gradients:

$$n_{l-1} n_l (2n_b - 1) + n_l (n_b - 1) \text{ FLOP}$$

Performance Model

Consider a layer l with n_l neurons, n_{l-1} input neurons and a batch size of n_b.

Total:

$$\sum_l 6n_l n_b n_{l-1} + 4n_l n_b - n_l(n_{l-1} + 1) - n_b n_{l-1}$$

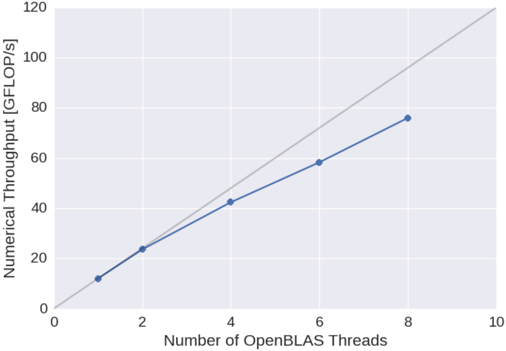
- Terms involving $n_l n_b n_{l-1}$ dominate complexity for the *hidden* layers.

Benchmarks

- Training Data:
  - Randomly generated data from a linear mapping $\mathbb{R}^{20} \to \mathbb{R}$
  - 10^5 input samples
- Network structure:
  - 5 hidden layers with 256 neurons
  - *tanh* activation functions
  - Squared error loss
- Computation of the numerical throughput based on the time elapsed for performing 10 training epochs.

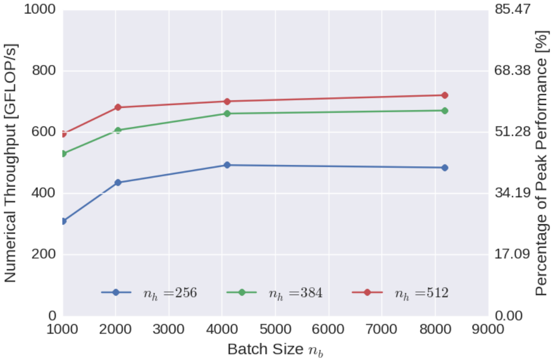
CPU Performance

Implementation: Multithreaded OpenBLAS and TBB
Hardware: Intel Xeon E5-2650, 8×4 cores, 2 GHz, estimated peak performance per core: 16 GFLOP/s



GPU Performance

Network: 20 input nodes, 5 hidden layers with n_h nodes each, squared error loss
Hardware: NVIDIA Tesla K20, 1.17 TFLOP/s peak performance (double)



GPU Performance

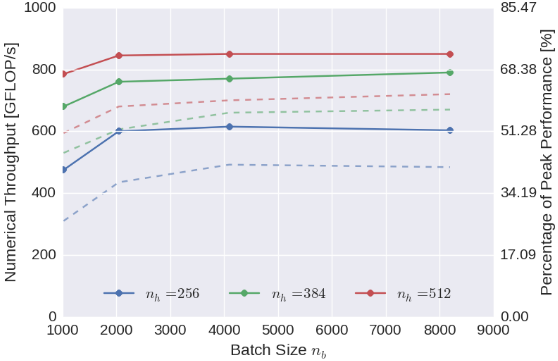
Optimization:

- Use compute streams to expose more parallelism to the device.
- Compute gradients for multiple batches in parallel.

GPU Performance

Optimization:

- Use compute streams to expose more parallelism to the device.
- Compute gradients for multiple batches in parallel.
- Using 2 streams:

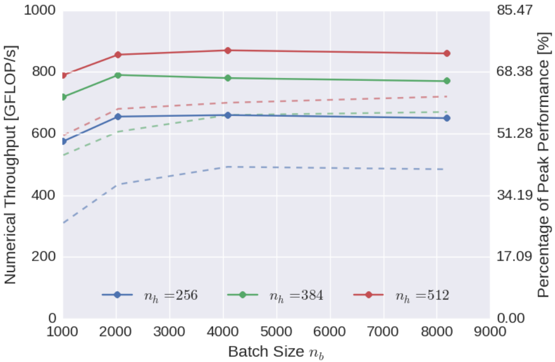


GPU Performance

Optimization:

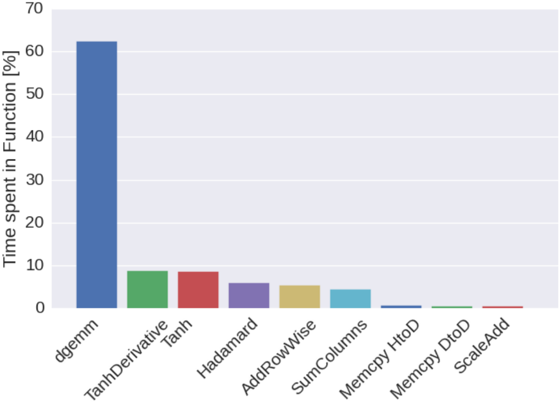
- Use compute streams to expose more parallelism to the device.
- Compute gradients for multiple batches in parallel.

- Using 4 streams:



GPU Performance

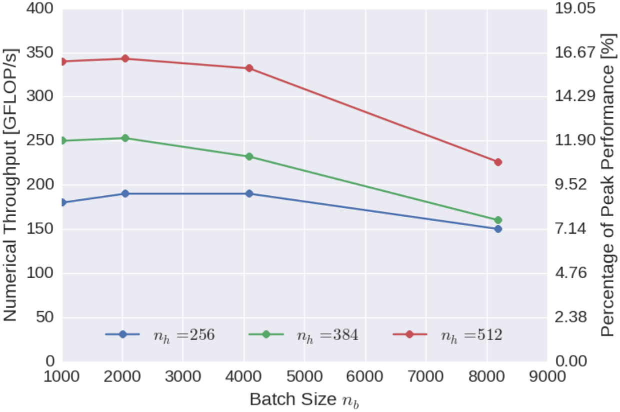
Network: 20 input nodes, 5 hidden layers with 256 nodes each, squared error loss
Hardware: NVIDIA Tesla K20, 1.17 TFLOP/s peak performance (double)



OpenCL Performance

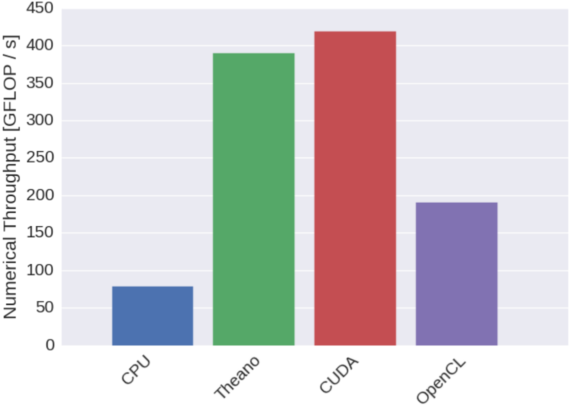
Network: 20 input nodes, 5 hidden layers with 256 nodes each, squared error loss

Hardware: AMD FirePro W8100, 2.1 TFLOP/s peak performance (double)



Summary

Network: 20 input nodes, 5 hidden layers with 256 nodes each, squared error loss



Application to the Higgs Dataset

The Higgs Dataset

- **Signal Process**:

$$gg \to H^0 \to W^{\pm}H^{\mp} \to W^{\pm}W^{\mp}h^0 \to W^{\pm}W^{\mp}b\bar{b}$$

- **Background Process**:

$$gg \to t\bar{t} \to W^{\pm}W^{\mp}b\bar{b}$$

[1] See `http://arxiv.org/pdf/1402.4735v2.pdf`

The Higgs Dataset

- **Signal Process**:

$$gg \to H^0 \to W^\pm H^\mp \to W^\pm W^\mp h^0 \to W^\pm W^\mp b\bar{b}$$

- **Background Process**:

$$gg \to t\bar{t} \to W^\pm W^\mp b\bar{b}$$

- 21 **low-level features**: Momenta of one lepton and the four jets, jet b-tagging information, missing transverse momentum
- 7 **high-level features**: Derived invariant masses of intermediate decay products
- Dataset consisting of 11 million simulated collision events

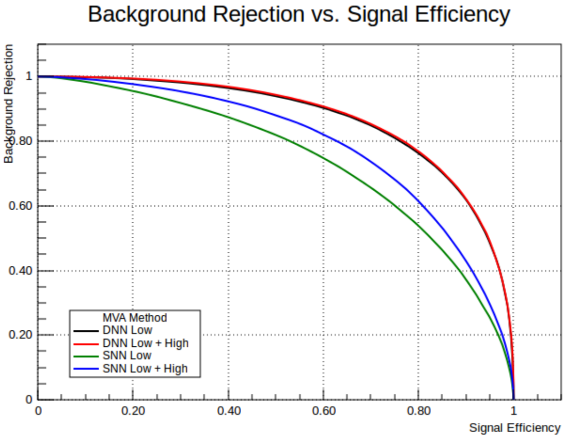
---

[1] See `http://arxiv.org/pdf/1402.4735v2.pdf`

Shallow vs. Deep Networks

- **Shallow Network**: 1 hidden layer with 256 neurons and *tanh* activation function and cross entropy loss
- **Deep Network**: 5 hidden layers with 256 neurons and *tanh* activation function and cross entropy loss
- Both networks trained once using only low-level features and once using both high-level and low-level features.

Shallow vs. Deep Networks

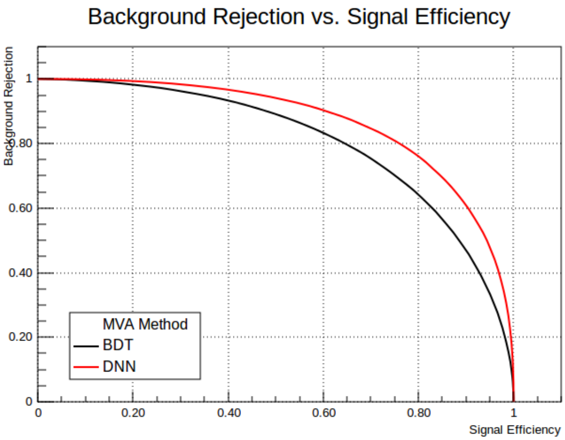


Deep Networks vs. BDT

- **Deep Network**: 5 hidden layers with 256 neurons and *tanh* activation function and cross entropy loss
- **Boosted Decision Trees**: 1000 Trees, maximum depth 3
- Both classifiers trained on low- and high-level features

Method	Training Time [h]	Area under ROC Curve
BDT	4.78 h	0.806
DNN	1.46 h	0.876

Deep Networks vs. BDT



Summary and Future Outlook

Results

- Testing and debugging of the prototype implementation of deep neural networks in TMVA.
- Production-ready implementation of parallel training of deep neural networks on CPUs and CUDA-capable GPUs.
- Reproduced Higgs benchmark results.

Future Outlook

- **Near Future:**
  - Integration of the CPU and CUDA
  - Finish OpenCL implementation
- Analyze performance on different architectures
- Extend neural network functionality: batch normalization, activation functions, AdaGrad, ...

Acknowledgments

Acknowledgments

- Project carried out at **CERN** within the **Google Summer of Code** program
- **Supervisors**: Sergei V. Gleyzer, Lorenzo Moneta

Acknowledgments

- Project carried out at **CERN** within the **Google Summer of Code** program
- **Supervisors**: Sergei V. Gleyzer, Lorenzo Moneta

Thank You!