GPU-accelerated training of deep neural networks in TMVA

Simon Pfreundschuh

May 31, 2016

In this report a design for an optimized implementation of deep feedfoward networks in TMVA is outlined. The primary aim of this project is to enable the training of neural networks on GPUs. This will be achieved by reimplementing the training algorithm in a generic way, which hides the structure of the underlying data types and thus allows efficient offloading of the computations to GPUs. Moreover, this will make the code independent of vendor specific APIs as well as simplify the porting of the code to other computing architectures.

The aim of this document is thus to introduce a low-level interface that separates the general training algorithm from the numerical operations that require device-specific tuning for optimal performance. In this way the current, general coordination of the training can remain unchanged and will be performed by the host, which launches the kernels that perform the actual calculations on the accelerator device.

1 The Neural Network Model

For this implementation we restrict ourselves to feedforward neural networks, where the activations $\mathbf{u}^l \in \mathbb{R}^{n_l}$ of a layer l are computed from the activations of the previous layer using

$$\mathbf{u}^l = f^l\left(\mathbf{W}^l\mathbf{u}^{l-1} + \boldsymbol{\theta}^l\right) \tag{1}$$

where $\mathbf{W}^l \in \mathbb{R}^{n_{l-1}} \times \mathbb{R}^{n_l}$ and $\boldsymbol{\theta}^l \in \mathbb{R}$ are the weights and bias values of layer l and f^l the corresponding activation function, which we assume here to be a scalar function $f : \mathbb{R} \to \mathbb{R}$ that is extended to vector or tensor arguments by element-wise application.

The training set is assumed to consist of m n-dimensional vectors $\mathbf{x} \in \mathbb{R}^n$. We are assuming the network to consist of an n_h hidden layers. For a given input vector $\mathbf{x}_i$, the ouput layer of the neural network computes an ouput vector $\mathbf{u}^{l_h}$, which is transformed into a probability or a class prediction by applying a suitable transformation.

2 Method

2.1 Neural Network Training

Deep neural networks are trained by optimizing a loss function $J\left(\mathbf{y}, \mathbf{u}^{l_h}, \mathbf{W}\right)$ that quantifies the correctness of the network prediction corresponding to the activations of the output layer $\mathbf{u}^{l_h}$ with respect to the true values $\mathbf{y}$ and possibly also including regularization terms, which are functions of the weights $\mathbf{W}$ of the network. This objective function is then minimized by applying a gradient-based optimization technique, usually a modification of the gradient descent method. The key to scalable training of deep neural networks is the training on small, randomized subsets of the training data, so called batches. On each batch the gradient is computed and used to train the network. The general method can thus be written as follows:

Algorithm 1 Neural Network Training

1: Initialize weight and bias variables
2: **repeat**
3: Generate random batch of size b
4: Propagate neural values foward through the network
5: Compute gradients on the batch using backward propagation
6: Apply minimization step
7: **until** converged

Here we will primarily focus on the steps in lines 4, 5 and 6, which contain the computationally demanding operations that will be offloaded to the accelerator device.

2.2 Forward Propagation

For the forward propagation, we will consider the propagation of the hole batch through the network, which will expose an additional axis of parallelization. The training input can then be viewed as as two-dimensional tensor $x_{i,j}$ with the index i running over the training samples in the batch and the index j running over the features of each training sample.

The neuron values of each layer can then also be represented by two-dimensional tensors $u^l_{i,j}$, with the index i running over the training samples in the batch and the index j running over the neurons in the given layer. We will refer to this tensor as the activation matrix of the given layer. The forward propagation of the neuron activations through the network is illustrated in Figure 1. To make the equations more easily readable Einstein notation is used, meaning that repeated indices imply a sum over those indices with the exception of indices that appear on the left-hand side of an equation.

The algorithm is given in pseudocode below. The computation of the neuron activations of each layer thus requires

- computation of the tensor product $W^l_{j,k} u^l_{i,k}$,
- addition of the bias vectors θ^l_k along the second dimension of the tensor,
- application of the activation function f^l to this tensor.

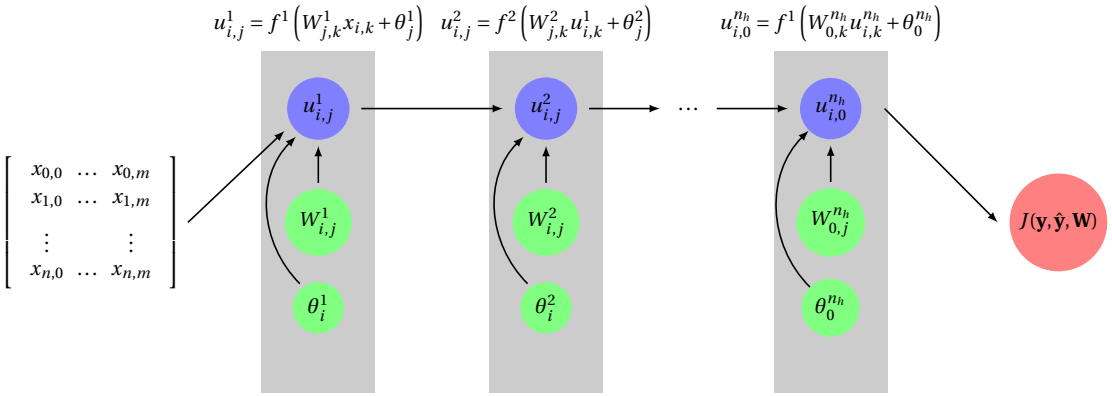


Figure 1: Forward propagation of the neuron values through the neural network. Repeated indices indicate summation over those indices.

Algorithm 2 Forward Propagation

1: $u^1_{i,j} \leftarrow f^1\left(W^1_{j,k}x_{i,k} + \theta^1_j\right)$
2: **for** $l = 1, \dots, n_h$ **do**
3: $\quad u^l_{i,j} \leftarrow f^l\left(W^l_{j,k}u^l_{i,k} + \theta^l_j\right)$
4: **end for**
5: obj $\leftarrow J\left(\mathbf{u}^l, \mathbf{y}, \mathbf{W}\right)$

2.3 Backward Propagation

The gradients of the objective function with respect to the weights of each layer are computed by repeated application of the chain rule of calculus. Starting from the gradients of the objective function with respect to the activations $u^{n_h}_{i,j}$ of the output layer $\frac{dJ}{du^{n_h}_{i,j}}$ the gradients can be computed from

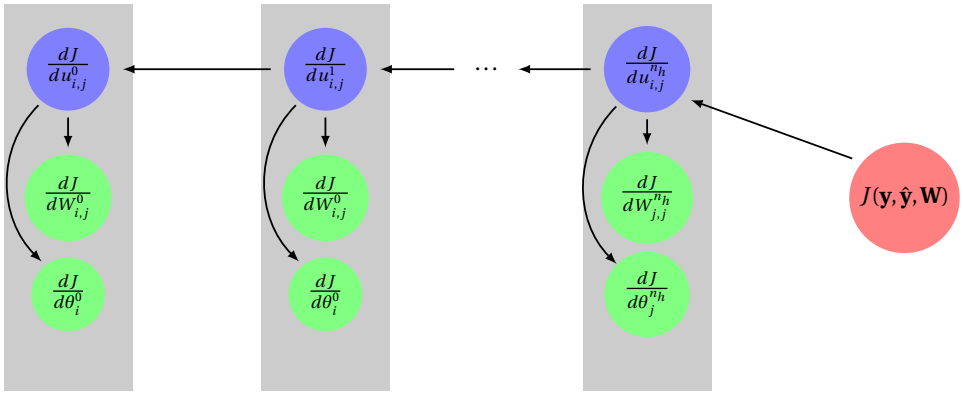
$$\frac{dJ}{dW^l_{i,j}} = u^{l-1}_{m,j}(f^l)'_{m,i}\frac{dJ}{du^l_{m,i}} + R(W^n_{i,j}) \tag{2}$$

$$\frac{dJ}{d\theta^l_i} = (f^l)'_{m,i}\frac{dJ}{du^l_{m,i}} \tag{3}$$

$$\frac{dJ}{du^{l-1}_{i,j}} = W^l_{n,j}(f^l)'_{i,n}\frac{dJ}{du^l_{i,n}} \tag{4}$$

Here, $R(W^n_{i,j})$ is an additional contribution to the gradient from potential regularization terms in the objective function. The term $(f^l)'_{m,i}$ is used to denote the first derivative of the activation function evaluated at $W^l_{i,k}u^{l-1}_{m,k} + \theta_i$. Note that the computations in equation (**??**)

can be implemented using element-wise as well as standard and transposed matrix-matrix multiplication.



$$\frac{dJ}{du^n_{i,j}} = W^l_{n,j} \left(f^n\right)'_{i,n} \frac{dJ}{du^{n+1}_{i,n}} \qquad \frac{dJ}{du^n_{i,j}} = W^l_{n,j} \left(f^n\right)'_{i,n} \frac{dJ}{du^{n+1}_{i,n}}$$

$$\frac{dJ}{dW^n_{i,j}} = u^n_{m,j} \left(f^n\right)'_{m,i} \frac{dJ}{du^n_{m,i}} \qquad \frac{dJ}{dW^n_{i,j}} = u^n_{m,j} \left(f^n\right)'_{m,i} \frac{dJ}{du^n_{m,i}} \qquad \frac{dJ}{dW^{n_h}_{i,j}} = u^{n_h}_{m,j} \left(f^{n_h}\right)'_{m,i} \frac{dJ}{du^{n_h}_{m,i}}$$

$$\frac{dJ}{d\theta^n_i} = \left(f^n\right)'_{m,i} \frac{dJ}{du^n_{m,i}} \qquad \frac{dJ}{d\theta^n_i} = \left(f^n\right)'_{m,i} \frac{dJ}{du^n_{m,i}} \qquad \frac{dJ}{d\theta^{n_h}_i} = \left(f^{n_h}\right)'_{m,i} \frac{dJ}{du^{n_h}_{m,i}}$$

Figure 2: Backward propagation of the gradients through the neural network. Repeated indices indicate summation over those indices.

3 Low-level Interface

In this section the low-level interface is presented, which will separate the compute intensive, mathematical operations from the general coordination of the training.

3.1 Forward Propagation

We split the forward propagation of the activations through a given layer into two steps. In the first step, the linear activations are computed using

$$\mathbf{t}^l = \mathbf{u}^{l-1}\mathbf{W}^T + \boldsymbol{\theta}^l \tag{5}$$

These operations are implemented by the `multiply_transpose` and the `add_row_wise` functions in the low-level interface.

```
void multiply_transpose( &output,
                         const MatrixType &input,
                         const MatrixType &weights)

void add_row_wise(MatrixType &output,
                  const MatrixType &biases)
```

In the second step, the non-linear activation function of the layer are applied to the temporary result $\mathbf{t}^l$ yielding the activations $\boldsymbol{u}^l$ of the current layer.

$$\mathbf{u}^l = f\left(\mathbf{u}^{l-1}\mathbf{W}^T + \boldsymbol{\theta}^l\right) \tag{6}$$

These operations are implemented using the `evaluate` function described in Section 3.3.

3.2 Backward Propagation

The backward propagation is implemented by a single method in the low-level interface. For a given layer l, this function takes the gradients of the objective function with respect to the activations of the $l+1$ (forward direction) layer, the weights of the current layer and the activations of the $l-1$ layer (backward direction). It also takes as input the first derivatives of the activation functions, which should ideally be computed during the forward propagation phase. The `backward` method computes the gradient of the objective function with respect to the activation energies of the previous layer, the weights of the layer as well as the biases.

Note that the formulas for backpropagation can be implemented using element-wise matrix multiplication as well as standard and transposed matrix-matrix multiplication.

```
template<typename RealType>
    void backward(MatrixType & activation_gradients_backward,
                  MatrixType & weight_gradients,
                  MatrixType & bias_gradients,
                  MatrixType & df,
                  const MatrixType & activation_gradients,
                  const MatrixType & weights,
                  const MatrixType & activations_backward,
                  Regularization r)
```

3.3 Activation Functions

The activation functions are represented by the `ActivationFunction` enum class.

```
/*! Enum that represents layer activation functions. */
enum class ActivationFunction
{
    IDENTITY = 'I',
    RELU     = 'R'
};
```

The evaluation of the activation functions is performed using the `evaluate` function template which forwards the call to the actual evaluation function corresponding to the given type of the activation function. An architecture-specific implementation simply overloads those evaluation function for the architecture-specific matrix data type. As an example the signature for the evaluation of the ReLU function is given below:

```
// Apply the ReLU functions to the elements in A.
inline void relu(MatrixType &A);
```

In addition to the evaluation of the activation functions, also a function that computes the first order derivatives of the activation functions must be provided. The signature for the computation of the first order derivatives of the ReLU function is also given below.

```
// For each element in A evaluate the first derivative of the ReLU function
// and write the result into B.
inline void relu_derivative(MatrixType &B, const MatrixType &A);
```

3.4 Loss Functions

Similar to the activation functions, the loss functions are also represented by an enum class.

```
enum class LossFunction
{
    MEANSQUAREDERROR = 'R'
};
```

The evaluation of the loss functions is implemented by the overloaded `evaluate` function, which computes the loss from a given activation matrix of the output layer of the network and the matrix $\mathbf{Y}$ containing the true predictions. Similar to the implementation of activation functions, the generic `evaluate` function forwards the call to a given device-pecific evaluation function, that is overloaded with the device-specific matrix type.

```
template<typename MatrixType>
    inline double evaluate(LossFunction f,
                    const MatrixType & Y,
                    const MatrixType & output)
```

In addition to that, we need to be able to compute the gradient of the loss function with respect to the activations of the output layer. This is implemented by the `evaluate_gradient` function, which also just forwards the call for a given loss function to the corresponding device-specific function.

```
template<typename MatrixType>
    inline void evaluate_gradient(MatrixType & dY,
                                  LossFunction f,
                                  const MatrixType &Y,
                                  const MatrixType &output)
```

Currently only the mean squared error function is implemented as a loss functions the function signature for the call to the device specific method is given below:

```
template<typename RealType>
inline RealType mean_squared_error(const TMatrixT<RealType> &Y,
                                   const TMatrixT<RealType> &output)
```

3.5 Regularization

For the treatment of regularization, we proceed in a similar way as above. The type of the regularization is represented by an enum class

```
/*! Enum representing the regularization type applied for a given layer */
enum class Regularization
{
    NONE = '0',
    L1   = '1',
    L2   = '2'
};
```

The generic `regularization` function resolves the type of the regularization and forwards the call to the corresponding device specific method.

```
template<typename MatrixType>
    auto regularization(const MatrixType & A,
                        Regularization R)
    -> decltype(l1_regularization(A));
```

In addition to that, we need to add the gradient of the regularization to the gradients of the prediction loss function. This is implemented by the `add_regularization_gradient` method.

```
template<typename MatrixType>
    void add_regularization_gradient(MatrixType &A,
                                     const MatrixType &W,
                                     Regularization R)
```

The type signatures of the device specific functions that compute the L2 regularization for a given weight matrix and add the gradients of the L2 regularization term with respect to a given matrix to another matrix are given below.

```
inline RealType l2_regularization(const MatrixType & W)
inline void     add_l2_regularization_gradient(MatrixType & A,
                                               MatrixType & W)
```

And similarly for L1 regularization:

```
inline RealType l1_regularization(const MatrixType & W)
inline void     add_l1_regularization_gradient(MatrixType & A,
                                               const MatrixType & W)
```

3.6 Initialization

The initialization of the layers is treated in a similar faschion as the activation and output functions as well as the regularization. An enum class specifies which type of initialization should be performed and the and a generic function the forward the call to the initialize function to the device-specific implementation of the desired initialization method.

```
/* Enum represnting the initialization method used for this layer. */
enum class InitializationMethod
{
    GAUSS    = 'G',
    UNIFORM  = 'U',
    IDENTITY = 'I'
};
```

The function signatures for the device-specific initialization methods are given below:

```
inline void initialize_gauss(MatrixType & A);
inline void initialize_uniform(MatrixType & A);
inline void initialize_identity(MatrixType & A);
```