High-Performance Computing
Prof. Dr. Ivan Kisel
Robin Lakos, Akhil Mithran, Oddharak Tyagi

High-Performance Computer Architectures
Practical Course
SoSe 2025

Issued: 14.05.2025 / Due: 21.05.2025

Institute for Computer Science

Exercise Sheet 3

Multi-Layer Perceptron

Neural networks, along with other machine learning algorithms, have become popular across various scientific disciplines. A fundamental example of neural networks is the multi-layer perceptron, which is often deemed a foundational neural network, since many advanced neural network models derive from it. This multi-layer perceptron has its roots in the simple perceptron, widely recognized as the cornerstone of neural network theory.

Nowadays, many old-fashioned algorithms are rethought with the knowledge about the power of machine learning algorithms. This does not exclude algorithms running on high-performance computers, and therefore, it is crucial to optimize these tools for high-performance computers as well.

In this exercise, you will complete the provided source code to have an operational multi-layer perceptron that is used to solve a classification task. For experimentation and testing, please download a well-known dataset for number recognition called *MNIST* here: https://hessenbox-a10.rz.uni-frankfurt.de/getlink/fi5AbbVoPXyKtw5qBWRqPw/ (password: hpc). This dataset features images of hand-drawn digits (0-9) that should be classified by the neural network.

The files are structured as follows. Each line consists of a label (the first number, which represents the digit drawn on the image) and 28×28 pixel values (grey-scale, 0-255) for the drawn digit itself. Each image is stored 1-dimensional (`std::vector<float>`), whereas all images together are stored 2-dimensional (`std::vector< std::vector<float> >`). Each image and label form together one training- xor test-sample.

While the supplied code is not optimized for speed, it is crafted to deepen your comprehension of the underlying mechanics. As we progress through subsequent exercises, we will refine and enhance this implementation.

Submission: Please upload your whole C++ CMake project including your solutions and meaningful comments as well as a PDF with your results and a short analysis to OLAT. Please also note the additional material uploaded to OLAT and do not hesitate to ask your tutor if you have any questions.

Exercise 2.1: Learning More about Neural Networks

First, understand and describe the different (hyper-)parameters and concepts of a model:

- depth of a network
- width of a layer
- training- vs. testing
- batch size

- epoch
- feed forward
- backpropagation
- loss
- learning rate

Exercise 2.2: Implementing the Feed Forward Functionality

Complete the `FeedForward`-method by implementing the affine transformation[1]. The affine transformation itself is based on matrix-vector multiplication `MatVecMul(...)` (row-major) and vector-vector addition `VecAdd(...)` (element-wise). Define these two methods in `Utils.cpp` for an operational affine transformation.

Exercise 2.3 Shuffling Training Samples and Labels

Depending on the weight optimization algorithm (e.g. with decaying learning rates), it might be necessary to shuffle the input samples and labels after each epoch. Define the `Shuffle(...)` method in `Utils.cpp`. This should change the order in which the samples are trained. Make sure that the labels keep connected to the respective input features.

Exercise 2.4 Hidden Neurons' δ

In the given source code, the calculation of hidden neuron's δ-values requires to perform a matrix transposition before using matrix-vector multiplication and the Hadamard product.

Firstly, define the `Utils::HadamardProduct`[2] for two vectors.

Secondly, implement the matrix transposition `Utils::Transpose(...)`. Then, think about to combine these two steps of matrix transposition and matrix-vector multiplication into a single function `Utils::MatTransposeVecMul(...)`. Do not just call the two functions inside the combined function, instead try to rethink the processing of data and which steps could be combined. Afterwards, answer the following questions. Why should it improve the run-time of the MLP training? Did it actually change the runtime? Compare and analyze the results.

Exercise 2.5 Learning Rate and TanH Activation Function

The results of the MLP with LeakyReLU activation, 784 input neurons, 800 hidden neurons, 10 output neurons and a given learning rate $\eta = 0.001$ (see `MLPLayer.h`) has a weak performance compared to the MLP without hidden layer. Test both and evaluate the results.

As a next step, keep the hidden layer with 800 neurons and change the learning rate (e.g. to 0.01 or 0.0001) and compare the results to the previous results. What has changed?

As a last step, implement the activation function TanH with its respective derivative. You can define the function in `MLPLayer.h`. Again, compare the results to previous results, using the default learning rate (0.001) and your best performing learning rate, if it is different.

Hint: None of these models should reach 100% accuracy on the test dataset, but some get close. After implementing and debugging the features of your program, we highly recommend to compile the code in `release` mode and to use maximum compiler optimization.

[1] https://en.wikipedia.org/wiki/Affine_transformation
[2] https://en.wikipedia.org/wiki/Hadamard_product_(matrices)