Chair for System Simulation

Winter Term 2023/2024

Prof. Dr. Harald Köstler, Richard Angersbach

# Advanced Programming Techniques
# Sheet 2 — Project: Handwriting Recognition

This sheet contains the instructions for the semester project. The project is not mandatory. However, we strongly recommend that you work on it as it serves as a preparation for the exam. Successful completion is rewarded with bonus points in the exam.

**Important:** Do not get discouraged by the heavy load of information provided in this sheet. In case you are unfamiliar with deep learning, we highly recommend watching these **video tutorials**[1] before reading the assignment sheet. We tried to describe the program flow of the neural net as detailed as possible while leaving out some mathematical details. Keep in mind that for larger projects it is completely natural that many aspects will not be comprehensible at the beginning, especially for Task 2. The project submission will be roughly at the end of the semester, so we strongly advise you to *take your time and read this sheet step-by-step, do your own research and have discussions in your group* to become more familiar with the tasks of this project.

## Handwriting Recognition via Neural Networks

The main task of this project is to implement a fully-connected neural network (FCNN) in C++ to classify handwritten digits from the MNIST[2] dataset. Handwriting recognition involves converting handwritten characters into machine-readable text. The FCNNs used for this task are required to map pixel values of handwriting images to digital characters, e.g. labeling the image in Figure 1 as the digit "3". This project is split into two parts: Data handling for the MNIST dataset in Task 1 and the implementation of the neural net in Task 2. We recommend you to finish Task 1 first before continuing with the more time-consuming tasks in Task 2.

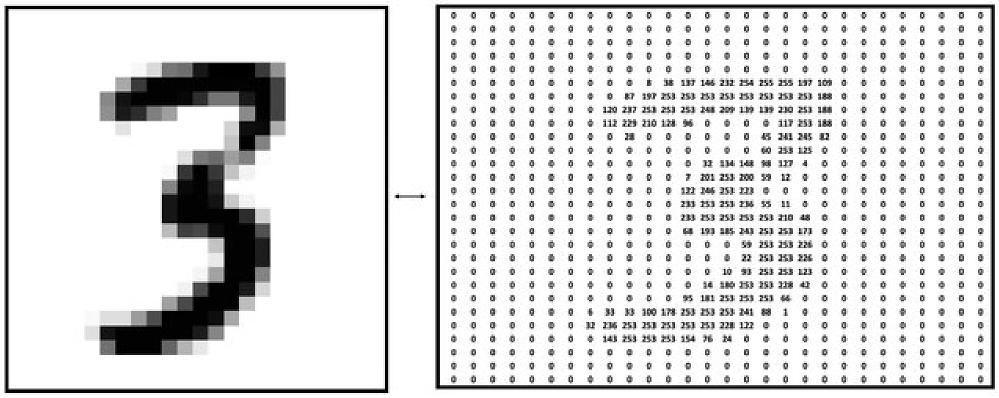


Figure 1: Exemplary MNIST image for classification (from [1])

[1]https://www.3blue1brown.com/topics/neural-networks
[2]http://yann.lecun.com/exdb/mnist/

## Workflow and submission

This is a **group project**. We ask you to form groups of three students. Since that is not always possible, you can also form groups of two or do the project alone. However, keep in mind that the workload for all groups is the same and, thus, doing the project in smaller groups will be highly challenging and time-consuming. Also, if everyone submits their own solution, the evaluation will be more involved and might take longer. Use this opportunity to practice working on a software project as a team.

The structure of your submission is mostly up to you. You have the opportunity to learn to structure your project yourself. There are only a few requirements that are listed below:

- You are forced to use our GitLab servers at the LSS to upload your submissions (if you want your submissions to be evaluated by us). We will provide you with an account and a repository where you simply commit and push your progress via Git (details below). In regular intervals, we will clone the project's `master` branch as is and perform the evaluation on it.
- You have to provide a set of shell scripts in the top-level directory of your repository:
  - `build.sh` is a shell script that, when executed, compiles and links your executable. This script would for instance run `cmake`, `make`, or directly invoke a compiler. It is up to you to put everything required to prepare your program in this script.
  - `read_dataset_images.sh` (cf. a more detailed description in Task 1 a)).
  - `read_dataset_labels.sh` (cf. a more detailed description in Task 1 b)).
  - `mnist.sh` (cf. a more detailed description in Task 2 a)).

## Evaluation

The results of the evaluation will be accessible via GitLab[3]. At least once a day (but likely more frequently) all codes are evaluated, and the results are pushed to a directory in the repository.

### Bonus points

Bonus points on the exam are awarded for completing the following tasks:

- I/O for the MNIST dataset (cf. Task 1)
  - I/O for MNIST images (cf. Task 1 a)): 2 Points
  - I/O for MNIST labels (cf. Task 1 b)): 1 Point
- Evaluating your FCNN code with MNIST data (cf. Task 2 a))
  - Outperforming a random number generator (i.e. $> 10$ % accuracy): 1 Point
  - Predicting handwritten letters with high accuracy: 4 Points
  - Overfitting with full accuracy: 2 Points

### Project registration

Please form groups of three people. Registration will be open via StudOn.

**Deadline for the registration is Monday, December 18 at 23:59.**

[3]https://i10git.cs.fau.de/advpt-student/advpt-mnist-evaluation-results

There is a table where you can enter a group name and the three email addresses of the group members. **Please use your @fau.de email adresses.** Once you have registered, we will create a GitLab account for each of you on `https://i10git.cs.fau.de`. Additionally, we will create a repository for each group. You will be informed via mail. Please use the forum if you encounter any problems with the registration or your GitLab account.

Use the repository to write your implementation as described in this sheet. We will pull the `master` branch of your repositories on a regular basis and run our tests against it. The results will be published so you can observe whether your implementation works. Also, we will upload `stdout` and `stderr` of the calls to the aforementioned shell scripts so you have the necessary information to fix your build process and program.

**Please pay attention to the StudOn forum where we will post updates on the organization of this project.**

**Continuous integration** Your repository will contain a directory with MNIST datasets and another directory with reference solutions so that you can test your implementation. Additionally, it contains a file called `.gitlab-ci.yml` that triggers a continuous integration pipeline. Each time you push changes to your repository, the pipeline will be triggered and execute your build and run scripts. This way you can check whether your code builds properly on our system, **but please only use our CI when you actually want to test your code**. Otherwise, we recommend adding `[skip ci]` at the end of your commit messages. **Abusing the CI resources for anything unrelated to the project will lead to a disqualification of your group**.

**Reference system** The reference system has `cmake` version 3.14.4 and `gcc` 11.1.0 installed.

## The MNIST dataset

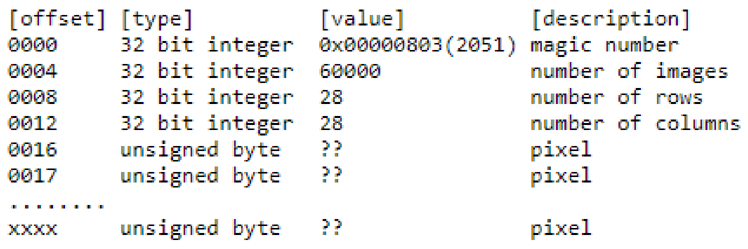
The MNIST dataset is a collection of greyscale images of handwritten digits ranging from 0 to 9. As shown in Figure 1, each image has a fixed size of 28x28 (= 784) pixels where each pixel denotes the intensity of the greyscale. The greyscale intensity ranges from 0 (white background) to 255 (black foreground). In total, the MNIST dataset comprises a training set of 60000 images and a testing set of 10000 images. The dataset is stored in four files and can be found in the `mnist-dataset/` directory in your Git repository:

- `train-images.idx3-ubyte`: training set images
- `train-labels.idx1-ubyte`: training set labels
- `t10k-images.idx3-ubyte`: test set images
- `t10k-labels.idx1-ubyte`: test set labels

The files for the images and labels contain binary data in the custom format shown in Figure 2. Integer values are stored in the most significant bit first (big-endian format) and **need to be swapped for little-endian machines**. The pixel and label data is encoded in unsigned bytes (i.e. `unsigned char` values ranging from 0 to 255).

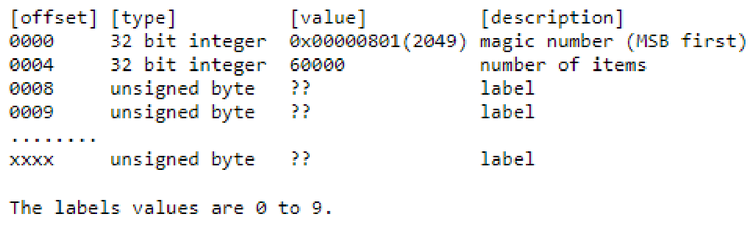
## Task 1 — I/O for the MNIST dataset

Your first task for the project is to implement input and output (I/O) routines for the MNIST dataset. The input of these routines is the aforementioned binary MNIST dataset files. To store the label and image data from the files, an array data structure is required. For this purpose, we provide you with a reference implementation of the previous tensor exercise sheet. It can be found in your Git repository in the `src/` folder. For the evaluation of your code, we rely on the output of its pretty-print function shown in Listing 1. We further split up this task to evaluate these routines for image and label data separately. Please pay close attention to the following descriptions of the expected memory layouts, data types and value ranges of the output data structures. For the sake of compactness, we will describe that using notations from the reference tensor class.



```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000803(2051) magic number
0004     32 bit integer  60000            number of images
0008     32 bit integer  28               number of rows
0012     32 bit integer  28               number of columns
0016     unsigned byte   ??               pixel
0017     unsigned byte   ??               pixel
........
xxxx     unsigned byte   ??               pixel
```

(a) Image data file format



```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000801(2049) magic number (MSB first)
0004     32 bit integer  60000            number of items
0008     unsigned byte   ??               label
0009     unsigned byte   ??               label
........
xxxx     unsigned byte   ??               label

The labels values are 0 to 9.
```

(b) Label data file format

Figure 2: File format for MNIST datasets (from http://yann.lecun.com/exdb/mnist/)

```
template< Arithmetic ComponentType >
void writeTensorToFile(const Tensor< ComponentType >& tensor, const std::string& filename);
```

Listing 1: Tensor pretty-print function

**Note:** you are not obliged to use the reference tensor class. In other words, you can modify the reference code, use data structures from scientific C/C++ libraries (e.g. Eigen tensors) or use your own data structures as long as they are output in the same way as in the reference pretty-print function.

**a) I/O for MNIST images** This task evaluates if your implementation can successfully read MNIST image data, convert the data into double-precision floating point values and output the resulting tensor using the pretty-print function in Listing 1. To evaluate your code, we run the `read_dataset_images.sh` shell script at the top level of your Git repository. Initially, the script only performs dummy echo statements. Extend it such that the C++ code responsible for handling this task is executed. The script expects three positional input arguments as shown in the following:

```
$ read_dataset_images.sh <image_dataset_input> <image_tensor_output> <image_index>
```

- `image_dataset_input` (String): Relative input path to an MNIST image dataset (e.g. `mnist-dataset/train-images.idx3-ubyte`).
- `image_tensor_output` (String): Relative output path for the file with the contents of the tensor pretty-print function.
- `image_index` (Integer): Value for selecting an image at a certain index of the MNIST dataset (e.g. the first MNIST training image, i.e. at index 0, depicts a "5").

For the output image tensors, we expect the tensor instances to have the:

- Type `Tensor<double>`.
- Two-dimensional shape `{number_of_rows, number_of_columns}` for the 28x28 pixels.

- The value range $[0.0, 1.0]$ which requires a linear mapping from the greyscale values in range $[0, 255]$, e.g. 255 should be mapped to 1.0 whereas 0 corresponds to 0.0. Note that this mapping is also important for the initialization of input data of your neural network.

**Hint:** A reference solution for a simple test in your CI pipeline can be found in your Git repository at location `expected-results/out-tensor-single-image.txt`. You can test your implementation by comparing the generated `image_out.txt` file with the reference solution.

```
$ read_dataset_images.sh mnist-datasets/train-images.idx3-ubyte image_out.txt 0
```

**b) I/O for MNIST labels** Similar to Task 1 a), this task evaluates if your implementation can successfully read MNIST image data, convert the data into double-precision floating-point values using a **one-hot encoding** and output the resulting tensor using the pretty-print function in Listing 1. Again, there is a shell script called `read_dataset_labels.sh` at the top level of your Git repository that should be extended to run your C++ implementation for this task. The script expects three positional input arguments as shown in the following:

```
$ read_dataset_labels.sh <label_dataset_input> <label_tensor_output> <label_index>
```

- `label_dataset_input` (String): Relative input path to an MNIST label dataset (e.g. `mnist-dataset/train-labels.idx1-ubyte`).
- `label_tensor_output` (String): Relative output path for the file with the contents of the tensor pretty-print function.
- `label_index` (Integer): Value for selecting a label at a certain index of the MNIST dataset (e.g. the first MNIST training label, i.e. at index `0`, is a `5`).

For the output label tensors, we expect the tensor instances to have the:

- Type `Tensor<double>`.
- One-dimensional shape of size `{10}`.
- The label values to be one-hot encoded, i.e. for the label "3" the tensor has a value of 1.0 at index 3 whereas all other tensor entries have the value 0.0. Note that this mapping is also important for the initialization of the output data of your neural network.

**Hint:** A reference solution for a simple test in your CI pipeline can be found in your Git repository at location `expected-results/out-tensor-single-label.txt`. You can test your implementation by comparing the generated `label_out.txt` file with the reference solution.

```
$ read_dataset_labels.sh mnist-datasets/train-labels.idx1-ubyte label_out.txt 0
```

## Task 2 — Network implementation

The main working package of this project is the implementation of a fully-connected neural network for the classification of handwritten digits.

## Network topology

For the implementation of your neural network, we expect the following network architecture shown in Figure 3. Summarized, the proposed network consists of the following layers:

- **Input layer** $I$: Consists of $n = 784$ input neurons and represents the flattened pixel values of the input images.

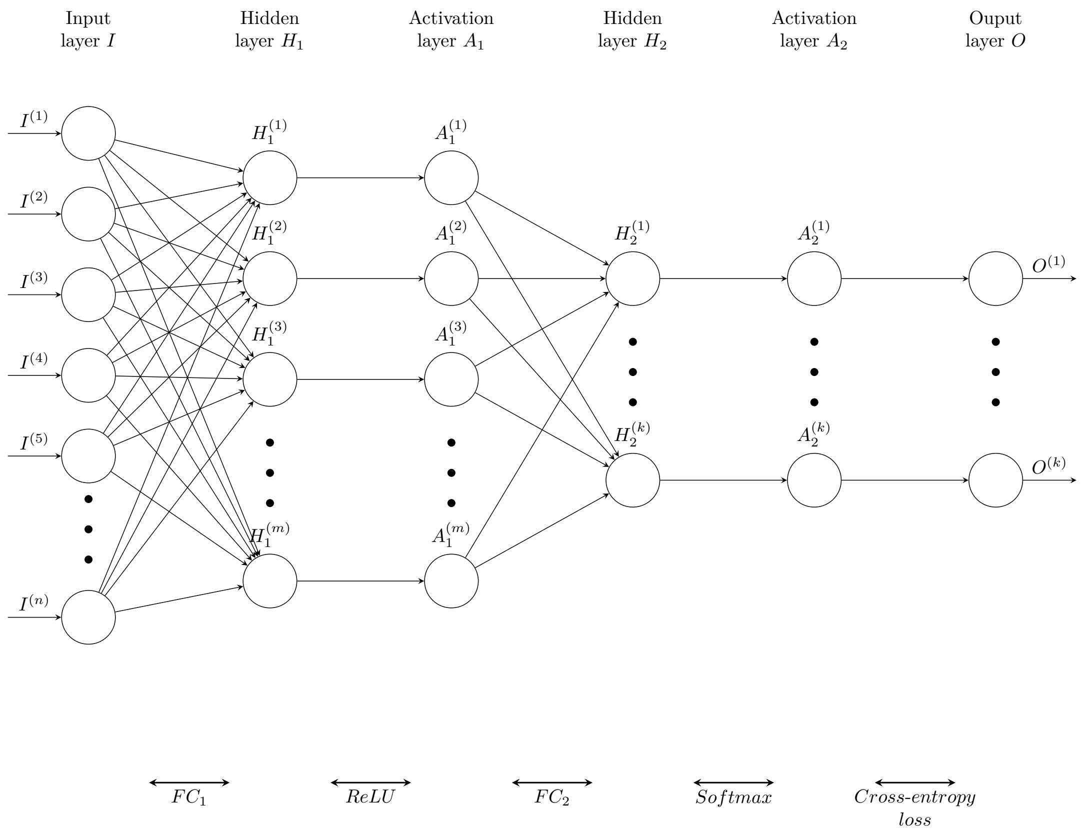


Figure 3: Project network architecture

- **Fully-connected layer 1 (FC1)** $H_1$: Has a parameterizable number of neurons $m$ and is responsible for learning spatial hierarchies and patterns in the input data. Here, each neuron is connected to every neuron from the previous input layer. The strength between two connected neurons from the previous and the current layer is stored in the *weights*, i.e. a matrix data structure. The weights belong to the trainable parameters of our network. Another trainable parameter is the *bias*, which allows us to adjust the output of the individual neurons independent on the input neurons.
- **ReLU activation function** $A_1$: Short for rectified linear unit (ReLU). Introduces non-linearity to the model by outputting the original input for positive values and outputting zero for negative values. The number of neurons in this layer remains the same, i.e. $m$.
- **Fully-connected layer 2 (FC2)** $H_2$: Responsible for a further refinement of the learned features from the previous layer. The number of neurons corresponds to the number of possible classes that can be predicted by the network, i.e. $k = 10$.
- **Softmax activation function** $A_2$: Facilitates the interpretation of the predictions of the network by transforming the output scores from the previous layer into a probability distribution.
- **Output layer** $O$: Holds the output of the softmax layer, i.e. the probability that the input $I$ belongs to a class $i \in [0, 9]$. The class with the highest probability is considered the predicted class. These results are used for the computation of the **cross-entropy loss** described later.

**PyTorch reference** We have added a reference code of the same architecture using the Python module PyTorch such that you can get an initial understanding of the program flow of FCNNs. We recommend you read through the reference code before starting with the following tasks. It is located in your Git repository under the `pytorch/` directory. The results (e.g. the development of the loss or the accuracy) of your FCNN implementation will most likely not coincide with the ones from the Python script since it is highly dependent on the implementation and setup for functionalities dealing with random numbers, e.g. the initialization of the weights of each layer. Since it is difficult to reproduce all employed functionalities of PyTorch exactly for this FCNN architecture, the results from this script **are not to be seen as ground truth** but should provide intuition on how our FCNN topology should behave, e.g. what order of magnitude for the prediction accuracy can be achieved.

## Network program flow

As evident from the PyTorch reference code, the program flow of our network implementation comprises a training and a testing phase. In the training phase, one or multiple images (i.e. a *batch* of images) from the *training dataset* are first propagated through the entire network in the forward direction (i.e. from left to right in Figure 3) until reaching the output layer. After the forward pass, the loss of the network is determined. The loss is a measure that quantifies the difference between the predicted class and the true class (i.e. label) probability distributions of the training data. This metric is computed with a loss function (cross-entropy in this project) and represents how well the network performs with the prediction task. Since the goal of the network training is to minimize the loss, the trainable parameters of the network (i.e. the weights and biases) are adjusted during the backward pass. The gradient of the loss with respect to (in short: w.r.t.) every trainable parameter is needed to optimize the loss. During the backward pass, a layer computes an error tensor and propagates it to a subsequent layer in the backward direction (i.e. from right to left) until reaching the input layer. These error tensors are used in some layers to compute loss gradients w.r.t. their own trainable parameters. These gradients can then be used by so-called *optimizers* to adjust the trainable parameters. We propose using a simple **stochastic gradient descent** (SGD) for this project. This procedure may be repeated with the same training dataset over the span of multiple *epochs*. Once the training phase is completed, the network is validated in the testing phase. Here, a batch of images from the *testing dataset* are fed forward to the trained network and the accuracy of the network (i.e. the fraction of correct predictions) is determined. Commonly, no backpropagation is done during testing.

**Batching** Feeding more than a single image through the network at once (aka *batching*) is a performance optimization technique used in deep learning to process multiple dataset items at once. Since multiple images are considered at once, a wider variety of scenarios is considered for tweaking the weights during backpropagation which may lead to fewer oscillations in the loss function descent. Note that **implementing batching is not mandatory for this project**, but helps to improve your overall performance.

### Forward propagation

Forward propagation is the process by which input data is fed through the layers of a neural network to produce a meaningful output or prediction that can be compared to the actual target values (i.e. label values). Note that we only briefly describe some mathematical components of the individual layers here. For more details, please refer to materials from the literature or from the lecture. The forward propagation consists of the following steps:

- Data loading and initialization: Load the MNIST dataset and add it to the flattened pixel array of input layer $I$. We recommend that you scale the greyscale pixel values to the range $[0, 1]$, e.g. as described in Task 1 a).
- Network parameter initialization: Initialize the weights and biases of your network. The initialization step plays an important role in the training convergence and is thoroughly discussed in the literature. Note that a zero-initialization may degrade the training convergence of your model significantly, so

it might be beneficial to take a look into other initialization techniques. For reproducibility, we also advise you to fix the random seed for your random number generators[4].

- FC1 forward pass: Thus for each neuron, a weighted sum is computed via a matrix-vector (without batching) or a matrix-matrix (with batching) multiplication between weights and input. Additionally, the bias is also applied to the weighted sum to compute the values of the output neurons.
- ReLU forward pass: Introduces non-linearity to each individual input neuron $x_i$ of $\boldsymbol{x}$ (i.e. output of FC1 forward pass) via the ReLU activation function:

$$f(x_i) = max(0, x_i).$$

- FC2 forward pass: Performs the same computations as in FC1, but with different weights, biases and input values.
- Softmax forward pass: Encode input neurons $\boldsymbol{x}$ (i.e. the output of the FC2 forward pass) to a probability distribution via the softmax activation function

$$\hat{y}_i = softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}},$$

  given a class $i$ out of $k$ total classes (here: $i \in [0, 9]$). For numerical stability, $x_i$ can be shifted to $\hat{x_i} = x_i - max(\boldsymbol{x})$.
- Cross-entropy loss computation: At the end of the forward pass, we compute the loss via the cross-entropy loss function defined as:

$$\sum_{i=1}^{k} y_i \, log(p_i),$$

  where $\hat{y_i}$ is the predicted probability and $y_i$ is the true class probability, i.e. 1.0 at the index of the label and 0.0 for others. Note that the true class probabilities can be modeled with the one-hot encoding tested in Task 1 b).

## Backward propagation

The backward pass involves computing the gradients of the loss w.r.t the trainable parameters. These gradients are then used to optimize the trainable parameters to further reduce the loss. The data flow of a network layer is shown in Figure 4. The green arrows represent the data flow of the forward pass and the ocher arrows denote the backward pass. The output $\hat{y}$ is a function of input $x$ and trainable parameters $w$. The forward pass receives $x$ from the previous layer, and then $\hat{y}$ is computed using the parameters $w$. In the backward pass, each layer receives the gradient of the loss *w.r.t. the layer's output* $\frac{\partial L}{\partial \hat{y}}$ (commonly known as error tensor) from the subsequent layer in the forward direction. This error tensor can then be used to compute the gradient *w.r.t. the layer's trainable parameters* using the chain rule $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w}$. The computed gradient can then be used to optimize these parameters. Since each layer propagates an error tensor through the network in the backward direction, each layer must also compute an error tensor for its predecessor w.r.t. the predecessor's output. The output of the predecessor is the input of the current layer, i.e. the error tensor for the predecessor can be computed with the chain rule again as $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial x}$. In principle, the main idea of the backward pass is nothing else than recursively applying the chain rule. Thus, you should make sure that you **store the input tensors in your layers**.

Again, we keep the mathematical terms short here.

[4] `https://cplusplus.com/reference/random/mersenne_twister_engine/mersenne_twister_engine/`

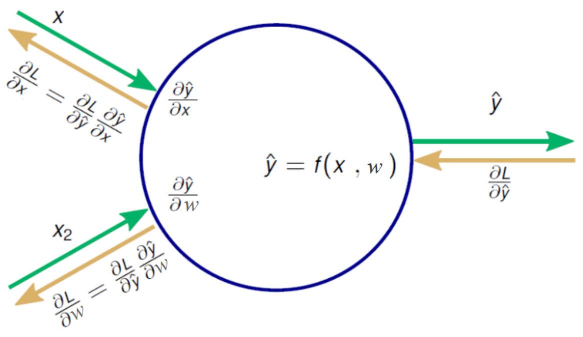

$x$
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial x}$
$\frac{\partial \hat{y}}{\partial x}$
$\hat{y} = f(x, w)$
$\frac{\partial \hat{y}}{\partial w}$
$x_2$
$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w}$
$\hat{y}$
$\frac{\partial L}{\partial \hat{y}}$

Figure 4: Backpropagation chain rule (modified from [2])

- Cross-entropy loss computation: Computes the initial error tensor since it is at the end of the forward traversal of the network. The gradient of the loss function w.r.t. inputs (i.e. output of the softmax forward pass) is computed as
$$e_n = -y/\hat{y},$$
where, again, $\hat{y}_i$ is the predicted probability and $y_i$ is the true class probability.

- Softmax backward pass: The softmax activation function does not have trainable parameters. Therefore, we simply compute the next error tensor $e_{n-1}$ for the predecessor layer as:
$$e_{n-1} = \hat{y} - (e_n - \sum_{j=1}^{k} e_{n,j} - \hat{y}_j)$$
where $\boldsymbol{\hat{y}}$ is the output generated from the forward layer and $e_n$ is our initial error tensor.

- FC2 backward pass: This layer contains trainable parameters and therefore requires the computation of the loss gradient $\frac{\partial L}{\partial w}$. This gradient is then used by the SGD optimizer at the end of a training iteration to adjust the weights and biases. Finally, the next error tensor is computed and backpropagated to the predecessor layer. Please refer to the lecture materials or the literature for the mathematical terms for computing both gradients.

- ReLU backward pass: Since the ReLU is also an activation function, we only apply the chain rule to compute the next error tensor as follows:
$$e_{n-1} = e_n \odot \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}.$$
Note that $\boldsymbol{x}$ is the input that was passed in the forward pass of the ReLU. The operator $\odot$ represents the element-wise multiplication.

- FC1 backward pass: (cf. FC2 backward pass)

- SGD optimization: Now, the computed loss gradients w.r.t. the trainable parameters are used in the SGD optimizer to adjust the weights with
$$w_{next} = w - \eta \frac{\partial L}{\partial w}.$$
Here, $\eta$ is the FCNN hyper-parameter for the learning rate.

**a) Evaluating your model** Similar to the previous I/O tasks, we provide a shell script barebone `mnist.sh` at the top level of your Git repository that should be extended to execute the training and testing phase of your C++ network implementation. This script takes a single positional argument `<input_config>` (String) which denotes a relative input path to a configuration file and is run as follows:

```
$ mnist.sh <input_config>
```

### Input configuration file

To steer the program flow of your FCNN implementation, input arguments for I/O and network hyper-parameters have been introduced. Due to the sheer number of arguments passed to your neural network executable, providing them within configuration files can prove to be more elegant than using command-line arguments for each. In this project, your implementation should be able to parse an input configuration file as in Listing 2. Please note that the order of the input parameters is not fixed and can be different when evaluating your code.

```
rel_path_train_images = train-images.idx3-ubyte // relative path to train image dataset
rel_path_train_labels = train-labels.idx1-ubyte // relative path to train label dataset

rel_path_test_images = t10k-images.idx3-ubyte // relative path to test iamge dataset
rel_path_test_labels = t10k-labels.idx1-ubyte // relative path to test label dataset

rel_path_log_file = log_predictions.txt // relative path to output log file (see below)

// hyper-parameters
num_epochs = 5           // number of epochs executed
batch_size = 1           // size of a batch
hidden_size = 500        // number of neurons after the first fully-connected layer (i.e. m)
learning_rate = 1E-3     // learning rate used by optimizers (e.g. SGD)
```

Listing 2: Exemplary input configuration file

### Prediction logger

To validate the accuracy of your neural network implementation, we expect a log file to be output by your program. An exemplary log file for a batch size of 100 is shown in Listing 3. As can be seen, the log file tracks the individual prediction and label classes for an image within a batch. Another reference log file for a single image (i.e. batch size of 1) can be found in `expected-results/out-prediction-log-single-image.txt`.

```
Current batch: 0
 - image 0: Prediction=7. Label=7
 - image 1: Prediction=2. Label=2
 - image 2: Prediction=1. Label=1
 - image 3: Prediction=0. Label=0
 - image 4: Prediction=4. Label=4
 - image 5: Prediction=1. Label=1
 - image 6: Prediction=4. Label=4
 - image 7: Prediction=9. Label=9
 - image 8: Prediction=6. Label=5
 - image 9: Prediction=9. Label=9
  ...
 - image 99: Prediction=9. Label=9
Current batch: 1
 - image 100: Prediction=6. Label=6
 - image 101: Prediction=0. Label=0
 - image 102: Prediction=5. Label=5
  ...
 - image 199: Prediction=2. Label=2
...
```

Listing 3: Exemplary input configuration file

**General remarks**

- Use double-precision throughout the FCNN implementation.
- Fix your random seed to obtain reproducible results.
- The reference solution of the tensor class is **slow** and potentially too slow to overcome the time limits for the evaluation of your code. Apply suitable optimizations to your code to overcome this issue. Some ideas are:
  - Modifying/replacing the tensor class implementation to overcome the bottleneck.
  - Implementing general optimizations such as address precalculation.
  - Employing shared-memory parallelism using C++ threads, OpenMP, pthreads, etc.
  - Implementing cache-efficient matrix-vector/matrix-matrix multiplications

**Tips**

- Read this sheet carefully!
- Find good literature for the math behind FCNNs.
- Write (sanity) tests for every single network layer to make sure everything works as intended.
- Make sure that your implementation works for an overfitting problem (e.g. using only a single dataset image) before starting with the whole MNIST dataset.
- Start off with a batch size of 1 and implement batching as soon as everything works first.
- A negative loss might indicate that your Softmax implementation is incorrect.
- Double-check your tests when your loss diverges.
- In case your loss scales with the batch size, a normalization is missing.

## References

[1] Soha Boroojerdi and George Rudolph. Handwritten multi-digit recognition with machine learning. pages 1–6, 05 2022.

[2] Andreas Maier. Activation functions and convolutional neural networks. Deep Learning Lecture Notes, 2020. Link to the univis course.