Challenge 2 APC 2024

November 2024

1 Introduction to Neural Networks

Artificial neural networks (ANNs), also known as neural networks (NNs), are a branch of machine learning models inspired by biological neural networks. These networks consist of interconnected units, or *neurons*, which are connected in ways that mimic biological synapses, as shown in Figure 1a. Each artificial neuron receives input signals, processes them, and sends output signals to other connected neurons. In NNs, the connections typically carry signals represented by real numbers, with each neuron output computed by applying a non-linear activation function to the weighted sum of inputs and a bias term.

Each neuron, as depicted in Figure 1b, consists of input data with corresponding weights, a bias, and an output. Operationally, the neuron evaluation can be expressed as:

$$y(x) = f\left(\sum_{i=1}^{n} w_i x_i + b\right), \tag{1}$$

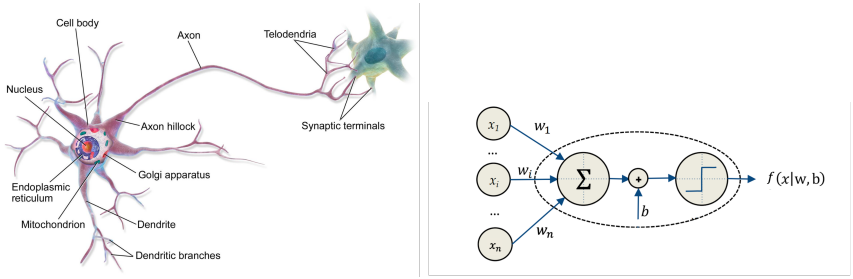
where $x \in \mathbf{R}^n$ is the input vector, $w \in \mathbf{R}^n$ is the weight vector, $b \in \mathbf{R}$ is the bias, and $f(\cdot)$ is the activation function. Common activation functions include sigmoid, tanh, and ReLU, with the latter often used in hidden layers for computational efficiency.

2 Convolutional Neural Networks (CNNs)

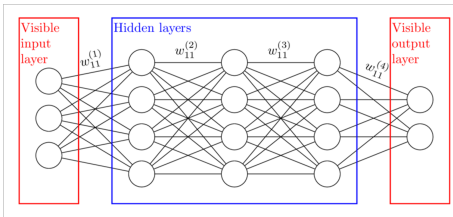
Convolutional Neural Networks (CNNs) are specialized in processing structured grid data, such as images, by applying filters to detect patterns and features. CNNs typically consist of convolutional layers, pooling layers, and fully connected layers, each with distinct functions.

2.1 Convolutional Layer

In each **convolutional layer**, a set of learnable filters slide over the input tensor, performing element-wise multiplications and summing the results to produce a feature map. This sliding process is governed by two parameters:



(a) Biological Neuron (b) Artificial Neuron



(c) NN structure

Figure 1: Comparison of biological neurons, artificial neurons, and neural network structure.

- **Stride:** The stride controls the step size of the filter as it moves across the input. A larger stride reduces the spatial dimensions of the output, as fewer regions are covered.
- **Padding:** Padding adds extra borders to the input tensor, allowing filters to overlap edge regions fully. This prevents a reduction in output size and helps retain spatial information at the edges.

For an input tensor `IM` with dimensions (H, W, D) and filters `F` with dimensions (F_H, F_W, D), the convolution operation yields an output tensor with dimensions $(H_{out}, W_{out}, n_{\text{filters}})$, where n_{filters} is the number of distinct filters applied in the layer. Each filter learns to detect a specific feature in the input tensor, such as edges or textures and produces a corresponding feature map in the output.

The dimensions of the output tensor are calculated as:

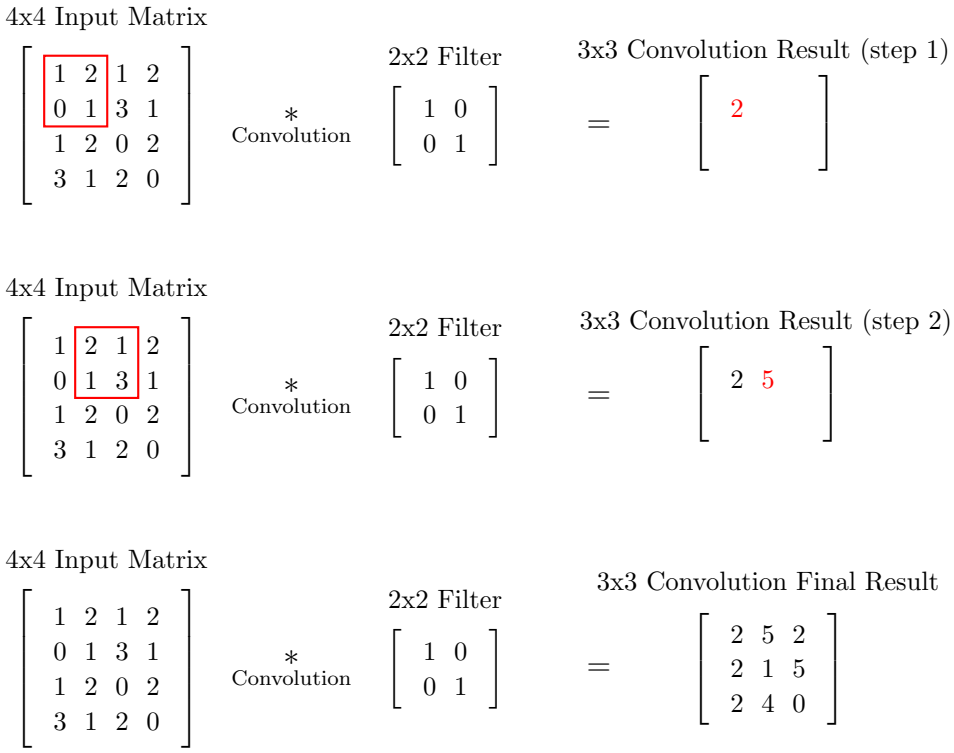
$$H_{out} = \frac{H - F_H + 2 \times \text{padding}}{\text{stride}} + 1, \tag{2}$$

$$W_{out} = \frac{W - F_W + 2 \times \text{padding}}{\text{stride}} + 1. \tag{3}$$

For the convolution operation at output position (i, j, k), where k indexes over filters, the formula is:

$$\text{Output}[i,j,k] = \sum_{h=0}^{F_H-1} \sum_{w=0}^{F_W-1} \sum_{d=0}^{D-1} \texttt{IM\_padded}[i \cdot \text{stride} + h, j \cdot \text{stride} + w, d] \cdot F_k[h,w,d] \tag{4}$$

F_k is the filter at index k. Also, in this assignment, we will consider a padding of zero, thus $\texttt{IM\_padded} = \texttt{IM}$.
Each feature map in the output corresponds to the response of a specific filter across the spatial dimensions of the input tensor. Figure 2 provides an example of a convolution between a 4x4 matrix and 2x2 filter, with stride of 1 and padding 0.



Convolution Calculations for Selected Output Elements:

(Step 1): $2 = (1 \times 1) + (2 \times 0) + (0 \times 0) + (1 \times 1)$

(Step 2): $5 = (2 \times 1) + (1 \times 0) + (1 \times 0) + (3 \times 1)$

Figure 2: Convolution of a 4x4 matrix with a 2x2 filter and stride of 1.

2.2 Pooling Layers

Pooling layers reduce spatial dimensions, making representations more compact. Common pooling operations include:

- **Max Pooling:** Selects the maximum value within a sliding window. For an input tensor `IM` with dimensions (H, W, D) , a window (filter) of size (F_H, F_W) and a stride, the output dimensions are:

$$H_{out} = \frac{H - F_H}{\text{stride}} + 1, \tag{5}$$

$$W_{out} = \frac{W - F_W}{\text{stride}} + 1. \tag{6}$$

  For each position (i, j, d) in the output tensor, the max pooling operation is defined as:

$$\text{Output}[i, j, d] = \max_{0 \le h < F_H,\, 0 \le w < F_W} \texttt{IM}[i \cdot \text{stride} + h, j \cdot \text{stride} + w, d] \tag{7}$$

  The max operation takes the maximum value within each $(F_H \times F_W)$ window along the depth dimension d.

- **Average Pooling:** Calculates the average value within a sliding window, following the same dimensional reduction as max pooling.

Figure 3 illustrates the max pooling operation on 4x4 matrix with a window (filter) size of (2x2) and a stride of 2.

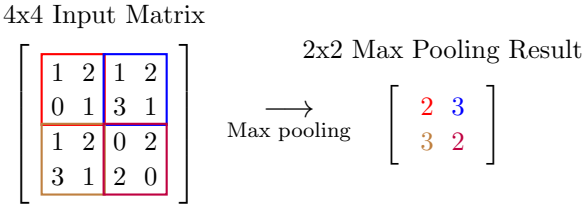


Figure 3: Max Pooling on a 4x4 matrix with a 2x2 filter and stride 2. Each colored box highlights the elements used to compute the corresponding maximum in the pooling result.

2.3 Fully Connected Layer

The **fully connected layer** (also called the dense layer) is a key component in neural networks, particularly in the final stages of a Convolutional Neural Network (CNN). Unlike the convolutional and pooling layers, which preserve the spatial structure of the input data, the fully connected layer reshapes the

data into a vector and connects every neuron to every neuron in the previous layer.

This layer acts as a combination of many neurons, where each neuron in the fully connected layer receives input from all the neurons in the previous layer, whether it is a convolutional or pooling layer. The weights associated with these connections are learned during training, allowing the network to make decisions based on the features extracted in earlier layers.

The fully connected layer can be thought of as a traditional neural network layer, where each output is calculated as in Equation 1. In a typical CNN, the final fully connected layer produces the output of the network, which could be a classification label in case of image classification tasks.

2.4 Applications of CNNs

CNNs are widely used in image classification, object detection, and semantic segmentation. Due to their ability to learn spatial hierarchies, they are effective in applications such as autonomous driving, medical image analysis, and natural language processing.

3 LeNet Architecture for Digit Recognition

In this assignment, we are considering a CNN called *LeNet*[1]. The *LeNet* architecture, as seen in Figure 4, developed by Yann LeCun, is a foundational CNN designed for digit recognition, commonly on the MNIST dataset[1]. LeNet processes 28×28 grayscale images through layers as follows:

1. **Convolutional Layer 1:** Applies 6 filters of size 5×5 and stride 1, generating a $24 \times 24 \times 6$ output.
2. **Max Pooling Layer 1:** Reduces each 2×2 window with stride 2, resulting in a $12 \times 12 \times 6$ output.
3. **Convolutional Layer 2:** Applies 16 filters of size 5×5 and stride 1 to yield a $8 \times 8 \times 16$ output.
4. **Max Pooling Layer 2:** Reduces the spatial dimension with 2×2 pooling and stride 2, producing a $4 \times 4 \times 16$ output.
5. **Fully Connected Layers:** These final layers transform the feature maps into class probabilities.

This architecture excels in digit recognition by capturing spatial hierarchies, transitioning from low-level features like edges to high-level features like digit shapes. The input is an image of a hand-written digit, and the network outputs the class associated with that digit. The forward pass involves feeding an input

[1]The MNIST dataset is a database of handwritten digits that is used to learn techniques and pattern recognition methods. It can be found here: `https://yann.lecun.com/exdb/mnist/`

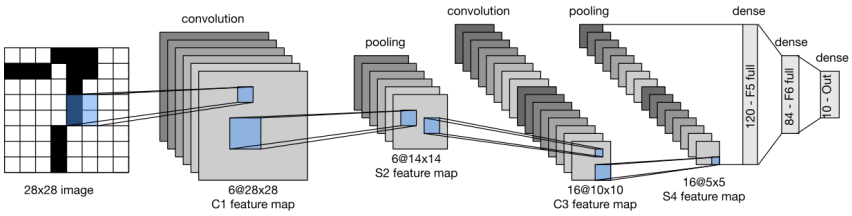


Figure 4: LeNet architecture for digit recognition

(such as an image of a handwritten digit) through each layer of the network, starting from the first layer and moving sequentially to the last, to compute the final output. Specifically, for a convolutional layer, that means performing the convolution and passing the result into an activation function (in our case, a ReLu activation function). For a max pooling layer that means performing only the max pooling operation since no activation function is required. For the fully connected layer, that means to compute the weighted sum of its input and the weights, and then pass the result into an activation function (in our case a sigmoid activation function). The final result is passed to a softmax function to classify the digit (i.e. output numbers 0-9). LeNet architecture is implemented by the function `test8()` in `test.hpp` of the code provided in `Assignment2_initial.zip`

4 Class diagram

Given the class diagram reported in Figure 5 you have to implement a library supporting the classification of the hand-written digit in C++.

In particular, here you can find an overview of all methods, divided by classes.

- Class: `matrix`
  - `matrix(std::size_t nrs, std::size_t ncs)`
    Constructor that initializes a matrix with specified dimensions `nrs` (rows) and `ncs` (columns).
  - `void print()`
    Prints the matrix values to the standard output.
  - `void initialize_with_random_normal(double mean, double variance)`
    Initializes matrix elements with values drawn from a normal distribution with the specified mean and variance.

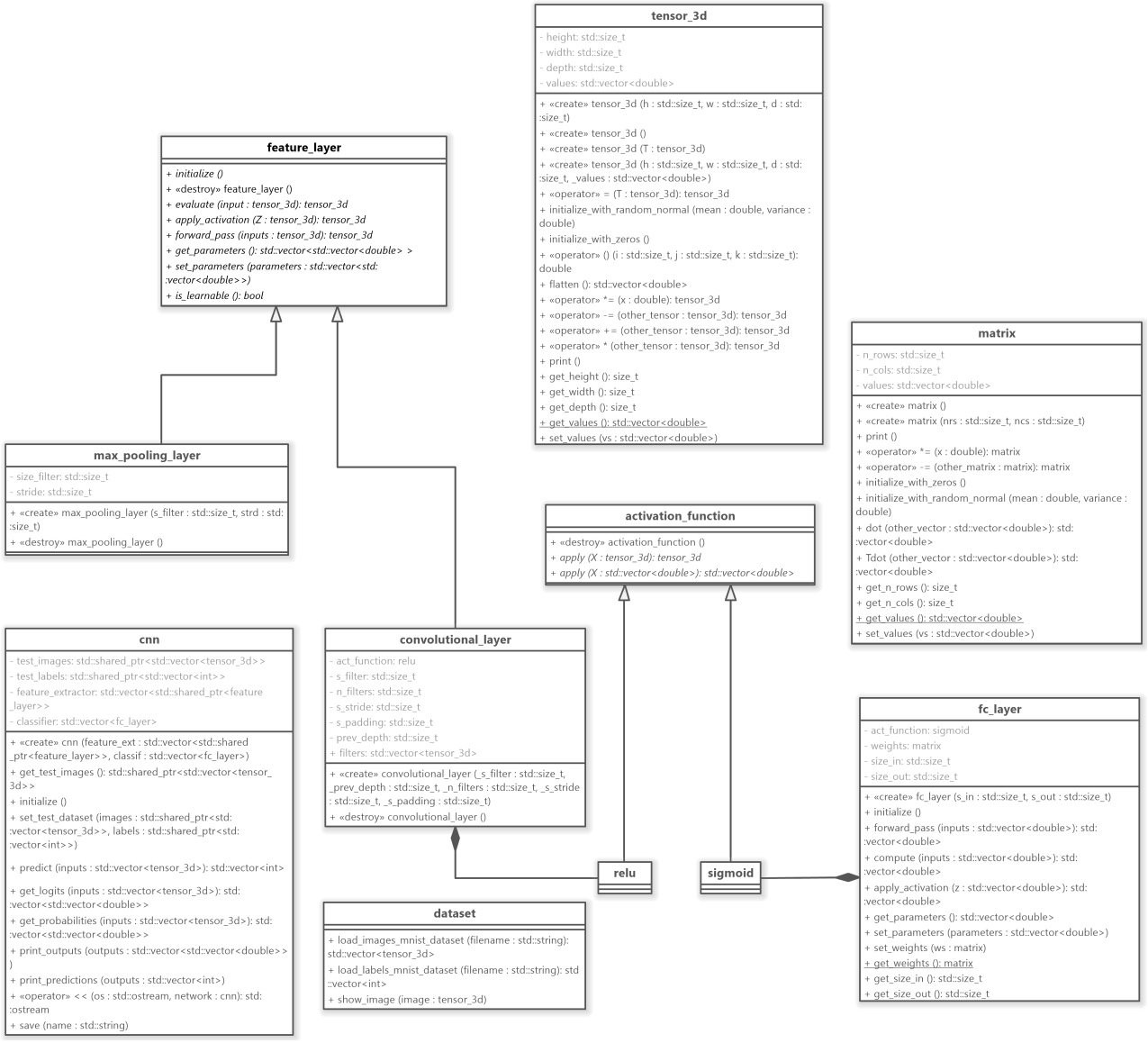


Figure 5: Class diagram

- `std::vector<double> dot(const std::vector<double>& other_vector)`
  Computes and returns the dot product of the matrix with a given vector.
- `std::vector<double> Tdot(const std::vector<double>& other_vector):`

  Computes the dot product of the transpose of the matrix with a given vector without explicitly storing the transpose.

- Class: `tensor_3d`
  - `tensor_3d(std::size_t h, std::size_t w, std::size_t d)`
    Constructor that initializes a tensor with given dimensions for height, width, and depth.
  - `tensor_3d(std::size_t h, std::size_t w, std::size_t d, std::vector<double>values)`
    Constructor that initializes a tensor with specified dimensions and a provided set of values.
  - `double operator()(std::size_t i, std::size_t j, std::size_t k) const`
    Provides read-only access to the element at position (i, j, k).
  - `double& operator()(std::size_t, std::size_t j, std::size_t k)`
    Provides access to the element at position (i, j, k) for modification.
  - `tensor_3d& operator*=(double x)`
    Scales the tensor by multiplying each element by a scalar value `x`.
  - `tensor_3d& operator+=(const tensor_3d& other_tensor)`
    Adds another tensor to the current tensor element-wise.
  - `tensor_3d operator*(const tensor_3d& other_tensor)`
    Performs element-wise multiplication with another tensor and returns the result.
  - `void print()`
    Prints the tensor's data to the standard output.

- Class: `feature_layer`
  - `void initialize()`
    Pure virtual method to initialize the layer, must be implemented by derived classes.
  - `tensor_3d evaluate(const tensor_3d& input)`
    Pure virtual method that performs the layer primary computation on the input tensor and returns the output tensor.
  - `tensor_3d apply_activation(const tensor_3d& Z)`
    Pure virtual method that applies an activation function to the input tensor `Z` and returns the activated tensor.

- `tensor_3d forward_pass(const tensor_3d& inputs)`
  Pure virtual method for a forward pass through the layer using the given inputs, returning the output tensor.

- Class: `fc_layer`
  - `fc_layer(std::size_t s_in, std::size_t s_out)`
    Constructor that initializes the fully-connected layer with specified input (`s_in`) and output (`s_out`) sizes.
  - `void initialize()`
    Initializes the layer weights, typically setting them to small random values or zeros.
  - `std::vector<double> forward_pass(const std::vector<double>& inputs):`

    Executes a forward pass through the layer, applying the weights and activation function to the input vector and returning the output vector.
  - `std::vector<double> compute(const std::vector<double>& inputs)`
    Computes the weighted sum of the inputs with the layer weights, returning the raw output (without activation).
  - `std::vector<double> apply_activation(const std::vector<double>& z)`
    Applies the activation function to the computed output vector `z`, returning the activated output.

- Class: `activation_function`
  - `tensor_3d apply(const tensor_3d& X)`
    Pure virtual function that applies the activation function to a 3D tensor input `X` and returns the result as a `tensor_3d`. This function is intended to be overridden by specific activation function implementations.
  - `std::vector<double> apply(const std::vector<double>& X)`
    Pure virtual function that applies the activation function to a 1D vector input `X` and returns the result as a `std::vector<double>`. This function is also intended to be overridden by derived classes.

- Class: `cnn`
  - `cnn(std::vector<std::shared_ptr<feature_layer>> feature_ext, std::vector<fc_layer> classif)`
    Constructor that initializes the CNN with specified feature extraction layers and fully connected classifier layers.
  - `std::shared_ptr<std::vector<tensor_3d>> get_test_images()`
    Returns a shared pointer to the test images used for evaluation.

- `void initialize()`
  Initializes all layers within the network by calling their respective initialization functions.
- `void set_test_dataset(std::shared_ptr<std::vector<tensor_3d>> images, std::shared_ptr<std::vector<int>> labels)`
  Sets the test dataset with images and corresponding labels for evaluation purposes.
- `std::vector<int> predict(const std::vector<tensor_3d>& inputs)`
  Generates a prediction by processing a batch of input images through the network, returning the predicted labels as an `std::vector<int>`.
- `void print_predictions(const std::vector<int>&) outputs`
  Prints the predicted labels in a formatted style to the console.
- `std::istream& operator>>( std::istream& is, cnn& network)`

  Overloaded stream operator that loads the network parameters from a specified input file, restoring a previously trained network state.
- `void load(const std::string& name)`
  Loads the network parameters from a file with the given name, allowing for reuse of pre-trained parameters.

5 Code implementation

The provided code zipped in `Assignment2_initial.zip`, is the base on which you should build your implementation.

Here is what you should implement without changing the provided code and/or initial declarations:

- Class: `convolutional_layer`
  - `tensor_3d evaluate(const tensor_3d& input)`
    Method that performs the convolution of the input with the convolutional filter as described in Equation 4.
  - `tensor_3d forward_pass(const tensor_3d& inputs)`
    Method that performs a forward pass of the layer to the input. The output is obtained after applying the activation to the result of the evaluate function.
- Class: `max_pooling_layer`
  - `tensor_3d evaluate(const tensor_3d& input)`
    Method that performs the max pooling on the input with the pooling filter as described in Equation 7.
  - `tensor_3d forward_pass(const tensor_3d& inputs)`
    Method that performs a forward pass of the layer to the input. The

output is obtained after applying the activation to the result of the evaluate function.

- Class: `fc_layer`
  - `std::vector<double> compute(const std::vector<double>& inputs)` Computes the weighted sum of the inputs with the layer weights, returning the raw output (without activation). This implements the equation 1. For simplicity, we consider the bias `b` as 0.
  - `std::vector<double> forward_pass(const std::vector<double>& inputs)` Executes a forward pass through the layer, applying the weights and activation function to the input vector and returning the output vector.

The following is the `main` function in the `main.cpp` file.

```
#include "test.hpp"

int main() {

    //test1();
    //test2();
    //test3();
    //test4();
    //test5();
    //test6();
    test7();

    return 0;

}
```

Important: you can comment and un-comment the lines with the `testK()` to test your code. `testK()` are implemented in `test.hpp`. Consider one test at a time. Note that some tests are incremental. For instance `test2()` depends on `test1()`. That means if you have issues with `test1()`, `test2()` will not work. `test4()` depends on `test3()`, in the same way `test6()` depends on `test5()`. Lastly `test7()` depends on all the other `testK()`. It is important that you do not change the content in the dataset and weights folders; otherwise, you will not be able to test your implementation. Also, do not add or remove any print on the standard output, in order to allow for a fair and exact evaluation of your results. `show_image` function in `dataset.cpp` helps display the image of the digit in the terminal, so we do not rely on an external library for image processing, which may result in installation problems for many of you.

6 Delivery Instructions

The assignment is not mandatory. If the solution is implemented correctly, it can lead to a +1 point in the final grade.

Please, follow these instructions for the delivery:

- Download the zipped folder `Assignment2_initial.zip` from WeBeep;
- Unzip the `Assignment2_initial.zip` folder;
- Change the name of the **unzipped** folder in "`YourCodicePersona`", e.g., "`10699999`". **It is important to do this before re-zipping the project with your solution.**
- Implement your code within the provided files;
- Test the code on the 7 test cases functions `testK()` (from `main.cpp`);
- Zip the folder containing the entire project, making sure that the resulting file name is "`YourCodicePersona.zip`", e.g., "`10699999.zip`";
- Upload on WeBeep $->$ Assignments $->$ Assignment2.

Attention: The assignment is personal; no group nor team work is allowed. In case of plagiarism, a -2 penalty is foreseen, you cannot participate in the next assignment, and you lose any points you earned with the previous assignment.

If you have questions, post them on the WeBeep Assignments forum. **Note that we will not provide feedback in the 24 hours before the deadline.**

Code submission opens on 25/11/2024 at 08:00, and closes on 30/11/2024 at 19:00 (Rome time).

If something changes in the source code we provided, an announcement will notify you. So, keep an eye on WeBeep these days.

References

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.