Challenge 2 APC 2023

November 2023

1 Introduction

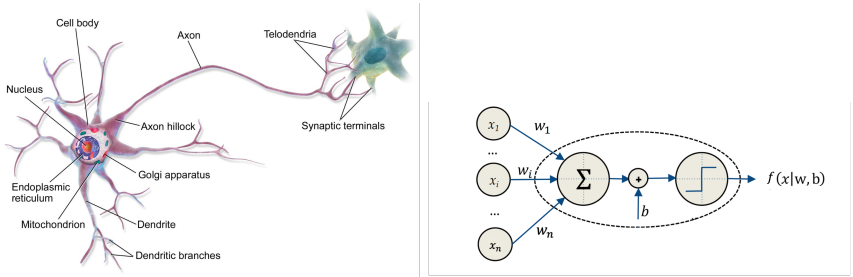
Artificial neural networks, also known as neural networks (NNs) or neural nets, reflect a branch of machine-learning models built using principles of neural organization in the biological neural networks constituting animal and human brains. In principle, NNs are based on a collection of connected units, often called nodes or artificial neurons. The connections created across such nodes mimic biological synapses (see Figure 1(a)) and are intended as ways to transmit signals between neurons. An artificial neuron therefore is designed to receive input signals, process them, and (possibly) send an output signal to all neurons connected to it. The "signal" at a connection level is typically represented by a real number, while the output of each neuron is computed by some non-linear function applied to the sum of its inputs and an additional number, called bias.

As shown in Figure 1b, each neuron consists of input data and corresponding weights, a bias, and an output. The weights determine the significance of the input variables, with greater weights giving more importance to the corresponding inputs, i.e., a more substantial contribution to the output value. Operationally, the inception phase—i.e. the use of the NN over a specific input—of a model operates as follows. First, each neuron calculates the weighted sum of its input(s) and adds the bias. A bias is a constant that provides an offset to the weighted sum of inputs influencing the neuron ease of activation and allowing it to learn and adapt to complex patterns in the data. Second, the total sum is subjected to an activation function, which determines the output of the neuron. The overall neuron evaluation can therefore be expressed as:

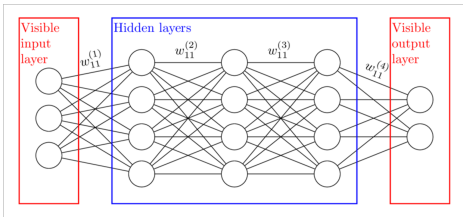
$$y(x) = f\left(\sum_{i=1}^{n} w_i x_i + b\right), \tag{1}$$

where $x \in \mathbf{R}^n$ is the input vector, $w \in \mathbf{R}^n$ is the vector of weights, $b \in \mathbf{R}$ is the scalar bias and $f(\cdot)$ is the non-linear activation function.

Activation functions help in introducing non-linearity in NNs. For example, sigmoid and tanh have been widely considered in the literature for their properties and suitability to binary and multi-class classification applications. For multi-class output, the softmax is typically used instead, and yields a probability distribution. Furthermore, the ReLU (which implements the function



(a) Biological Neuron (b) Artificial Neuron



(c) NN structure.

Figure 1: Biological neuron, artificial neuron, and NN structure (biases are omitted to simplify the representation)

$y(x) = \max(0, x)$) is also quite common and favored for hidden layers given its simplicity, computational efficiency, and ability to mitigate many numerical issues. The choice of activation functions depends on the problem characteristics and desired network behavior.

In most cases, neurons are organized into layers (see Figure 1c), and each layer may carry out distinct transformations on its inputs. A layer $\mathcal{L}$ is evaluated starting from the evaluation of its neurons, i.e., $Y(x)$ is a vector whose element in position ν is the result of Equation (1) applied to neuron $\nu \in \mathcal{L}$.

Finally, signals propagate from the initial layer (the input layer) to the final layer (the output layer), i.e., a network is evaluated by considering the output of layer l as the input of the next layer $l + 1$.

NNs excel in classification tasks by accurately assigning input data to predefined categories. An example task is image recognition, as shown in Figure 2, which is widely used by many modern systems. Moreover, NNs can be applied to matters such as computer security: spam detection is a good example of this. In regression tasks, NNs predict continuous values, making them valuable for applications like stock price forecasting or housing price predictions. In general, many applications in diverse fields have emerged in recent years, including finance, healthcare, and natural language processing. In fact, NNs capacity to learn from data and generalize patterns can be easily leveraged and applied pretty much everywhere.

Given the class diagram reported in Figure 3 you have to implement a library to support the evaluation of the function encoded in a NN.

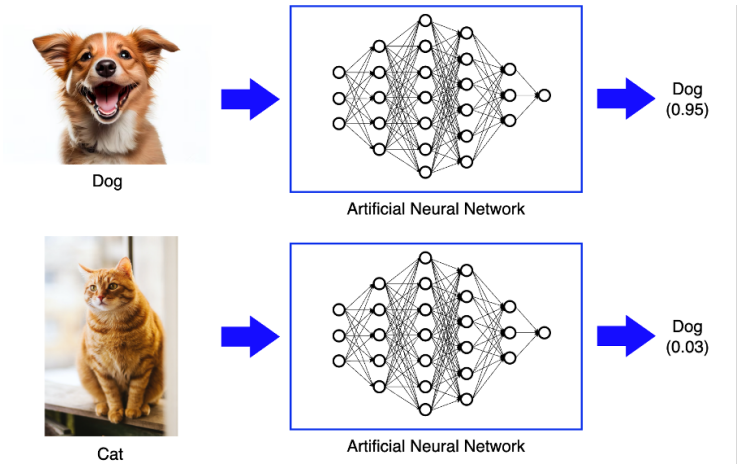


Figure 2: Image classification example classifying *dog* or *not dog*

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

- Class: `dense_matrix`
  - `dense_matrix()`
    Default constructor for creating an empty matrix.
  - `dense_matrix(rows: size_type, columns: size_type, value: const_reference)`
    Constructor for creating a matrix with specified dimensions and initial value.
  - `dense_matrix(istream&)`
    Explicit constructor for creating a matrix from the input stream.
  - `read(istream&): void`
    Reads matrix data from the input stream.
  - `swap(dense_matrix&): void`
    Swaps the contents of two matrices.
  - `operator()(i: size_type, j: size_type): reference`
    Provides access to the element at position (i, j) for modification.
  - `operator()(i: size_type, j: size_type): const_reference`
    Provides read-only access to the element at position (i, j).
  - `rows(): size_type`
    Returns the number of rows in the matrix.
  - `columns(): size_type`
    Returns the number of columns in the matrix.

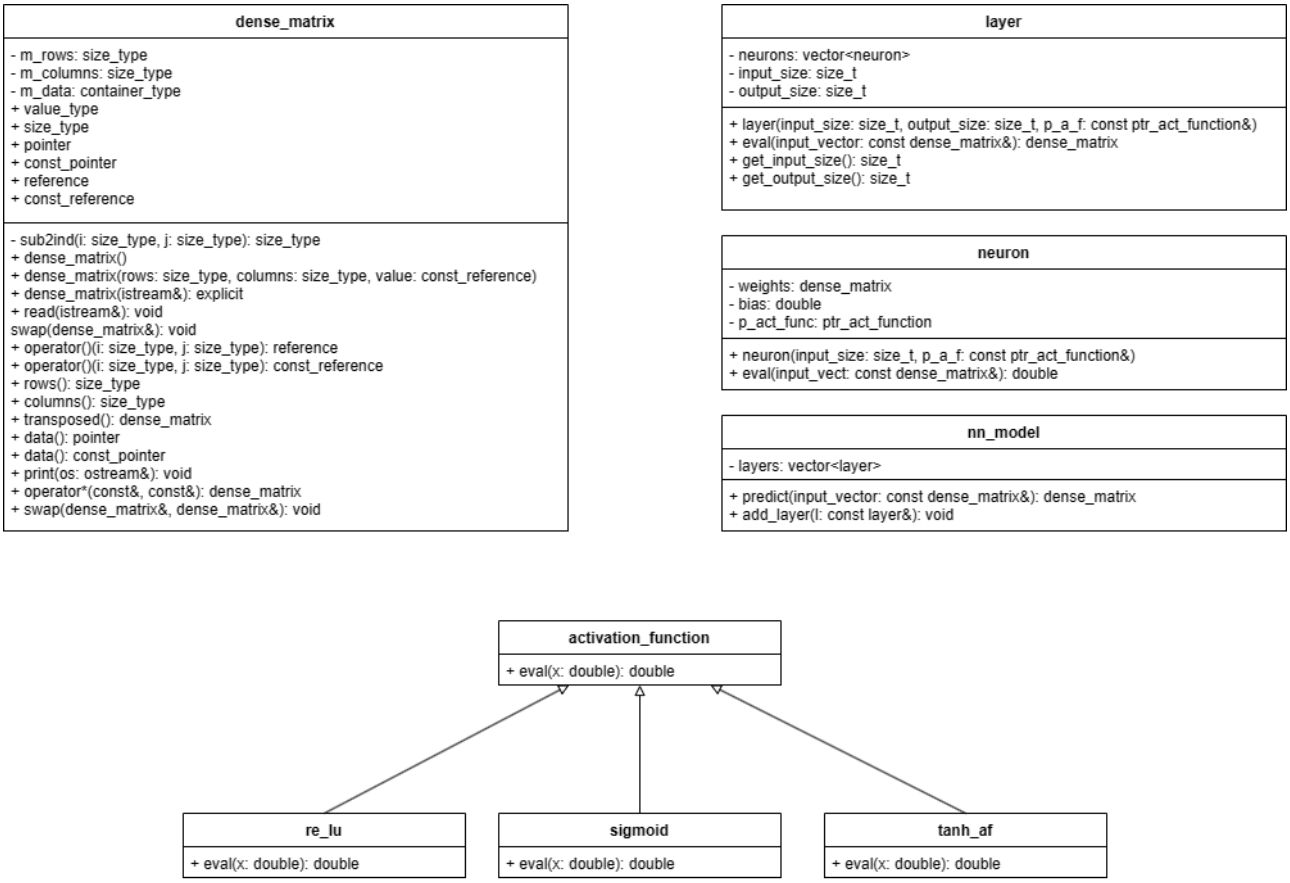


Figure 3: Class diagram

- `transposed(): dense_matrix`
  Returns a new matrix that is the transpose of the current matrix.
- `data(): pointer`
  Returns a pointer to the matrix data.
- `data(): const_pointer`
  Returns a constant pointer to the matrix data.
- `print(os: ostream&): void`
  Prints the matrix to the specified output stream.
- `operator*(const&, const&): dense_matrix`
  Multiplies two matrices and returns the result.
- `swap(dense_matrix&, dense_matrix&): void`
  Swaps the contents of two matrices.

- Class: `layer`
  - `layer(input_size: size_type, output_size: size_type, p_a_f: const ptr_act_function&)`
    Constructor for creating a neural network layer with specified input size, output size, and activation function.
  - `eval(input_vector: const dense_matrix&): dense_matrix`
    Evaluates the layer for a given input vector and returns the result.
  - `get_input_size(): size_type`
    Returns the input size of the layer.
  - `get_output_size(): size_type`
    Returns the output size of the layer.

- Class: `neuron`
  - `neuron(input_size: size_type, p_a_f: const ptr_act_function&)`
    Constructor for creating a neural network neuron with specified input size and activation function.
  - `eval(input_vect: const dense_matrix&): double`
    Evaluates the neuron for a given input vector and returns the result.

- Class: `nn_model`
  - `predict(input_vector: const dense_matrix&): dense_matrix`
    Makes predictions for a given input vector using the neural network model.
  - `add_layer(l: const layer&): void`
    Adds a layer to the neural network model.

- Class: `activation_function`

– `eval(x: double): double`
  Evaluates the activation function for a given input and returns the result.

For the sake of simplicity, assume that all the neurons weights are equal to 1 and the biases are zeros. As activation functions, you shall implement sigmoid, tanh, and ReLU. For the vector operations, you can rely on the `dense_matrix` class introduced during our classes.

2 Code implementation

The provided code, zipped in `Assignment2_initial.zip`, contains:

- the `dense_matrix` class implementation;
- the structure of the `neuron` class with its constructor implementation;
- a skeleton for the declarations of the `layer` class;
- a skeleton for the declarations of the `nn_model` class;.

You have to implement all classes and methods that are not implemented yet, without changing the provided code and/or initial declarations.

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

```
#include <iostream>
#include <memory>
#include "activation_function.h"
#include "sigmoid.h"
#include "tanh_af.h"
#include "re_lu.h"
#include "dense_matrix.hpp"
#include "utils.h"
#include "neuron.h"
#include "layer.h"
#include "nn_model.h"

using std::shared_ptr;
using std::make_shared;

int main() {

//#include "test1.h"
//#include "test2.h"
//#include "test3.h"
//#include "test4.h"
//#include "test5.h"
//#include "test6.h"
```

```
#include "test7.h"


}
```

Important: you can comment and un-comment the lines with the `#include ‘‘testK.h’’` to test your code (consider one test at a time. Note that tests are incremental. If you have issues with test$X$, test$X+1$ will not work). In order to compare the results with those in the `main.cpp` file, it is important that you do not change the input vector value and/or the structure of the NN provided for your tests; otherwise, your implementation may work but not show the same results. 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.

3 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 `testK.h` (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 -1 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 30/11/2023 at 08:00, and closes on 4/12/2023 at 18:00 (Rome time).

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