Neural Network

1 Introduction

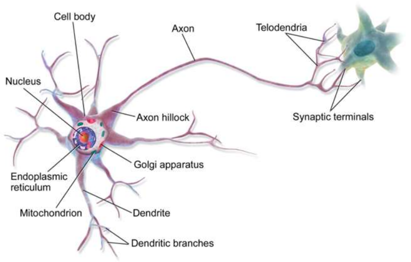
Artificial neural networks, also known as neural networks (NNs) or neural nets, reflect a branch of machine-learning models built using principles of neural organization in the biological neural networks constituting animal and human brains. In principle, NNs are based on a collection of connected units, 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:

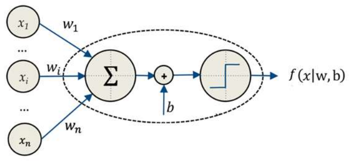
$$y(x) = f\left(\sum_{i=1}^{n} w_i x_i + b\right), \tag{1}$$

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.

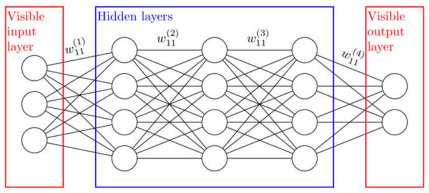
Activation functions help in introducing non-linearity in NNs. For example, sigmoid and tanh have been widely considered in the literature for their properties and suitability to binary and multi-class classification applications. For multi-class output, the softmax is typically used instead, and yields 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 L is evaluated starting from the evaluation of its neurons, i.e., $Y(x)$ is a vector whose element in position v is the result of Equation (1) applied to neuron $v \in$ 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.

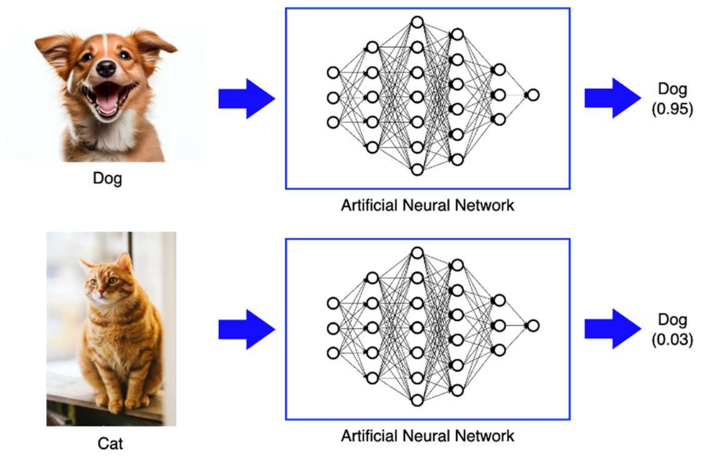


Figure 2: Image classification example classifying *dog* or *not dog*

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: sizetype): reference
    Provides access to the element at position (i, j) for modification.
  - operator()(i: size type, j: sizetype): 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.

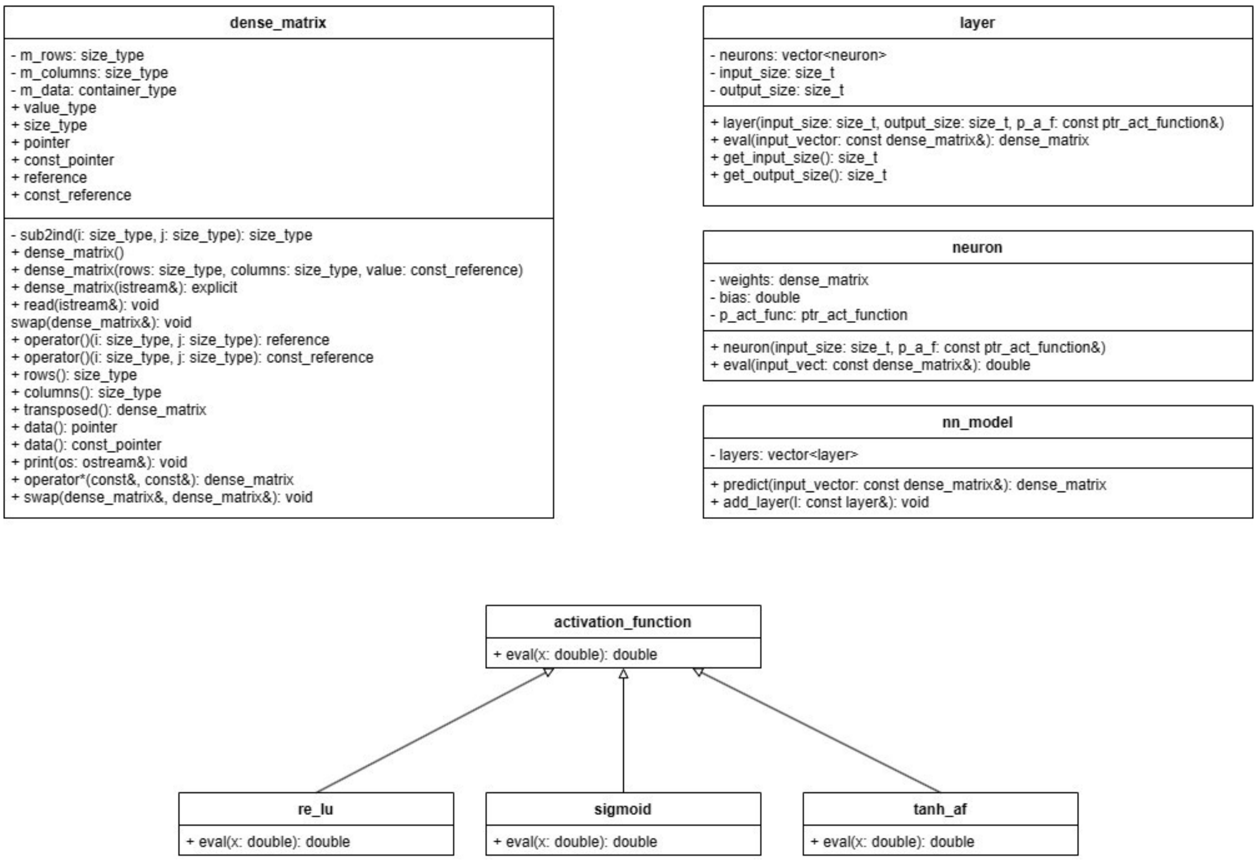

dense_matrix
- m_rows: size_type
- m_columns: size_type
- m_data: container_type
+ value_type
+ size_type
+ pointer
+ const_pointer
+ reference
+ const_reference
- sub2ind(i: size_type, j: size_type): size_type
+ dense_matrix()
+ dense_matrix(rows: size_type, columns: size_type, value: const_reference)
+ dense_matrix(istream&): explicit
+ read(istream&): void
swap(dense_matrix&): void
+ operator()(i: size_type, j: size_type): reference
+ operator()(i: size_type, j: size_type): const_reference
+ rows(): size_type
+ columns(): size_type
+ transposed(): dense_matrix
+ data(): pointer
+ data(): const_pointer
+ print(os: ostream&): void
+ operator*(const&, const&): dense_matrix
+ swap(dense_matrix&, dense_matrix&): void
layer
- neurons: vector<neuron>
- input_size: size_t
- output_size: size_t
+ layer(input_size: size_t, output_size: size_t, p_a_f: const ptr_act_function&)
+ eval(input_vector: const dense_matrix&): dense_matrix
+ get_input_size(): size_t
+ get_output_size(): size_t
neuron
- weights: dense_matrix
- bias: double
- p_act_func: ptr_act_function
+ neuron(input_size: size_t, p_a_f: const ptr_act_function&)
+ eval(input_vect: const dense_matrix&): double
nn_model
- layers: vector<layer>
+ predict(input_vector: const dense_matrix&): dense_matrix
+ add_layer(l: const layer&): void
activation_function
+ eval(x: double): double
re_lu
+ eval(x: double): double
sigmoid
+ eval(x: double): double
tanh_af
+ eval(x: double): double

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.
  - getinput 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.