Artificial Neural Networks

Course-4

Gavrilut Dragos

rev 1.1

AGENDA FOR TODAY

- Neuron Unit
- Neuronal Network architecture
- Activation functions
- Feed Forward step

Basic calculus notions

Basic calculus notions

The **derivative** of a function is a method used to understand the behavior of functions and changes in quantities.

- The most common interpretation of the derivative is the **rate at which one quantity changes with respect to another**. It's a measure of how sensitive a function is to changes in its input values.
- In a graphical context, the derivative at a certain point corresponds to the **slope of the line** that touches the graph of the function at that point without cutting through it (the **tangent line**). This gives you an idea of how steep the function is at any given point.

Basic calculus notions

Let’s see some basic ways to compute the **derivate** of a function f, denoted by **f’**:

1. The derivative of a constant function is a 0.

$$f(x) = constant, f'(x) = 0$$

Example:

$$f(x) = 5, f'(x) = 0$$

Basic calculus notions

Let’s see some basic ways to compute the **derivate** of a function f, denoted by **f’**:

2. The derivative of a linear function:

$$f(x) = a \times x, a \in R, a \text{ } is \text{ } a \text{ } constant, f'(x) = a$$

Example:

$$f(x) = 5 \times x = 5x, f'(x) = 5$$

Basic calculus notions

Let’s see some basic ways to compute the **derivate** of a function f, denoted by **f’**:

3. The derivative of a polynomial function:

$$f(x) = a \times x^n, a, n \in R, a, n\ are\ constants,$$
$$f'(x) = a \times n \times x^{n-1}$$

Example:

$$f(x) = 5x^3 = 5 \times x^3, f'(x) = 5 \times 3 \times x^{3-1} = 15 \times x^2 = 15x^2$$

Basic calculus notions

Let’s see some basic ways to compute the **derivate** of a function f, denoted by **f’**:

4. The derivative of a exponential function:

$$f(x) = e^x, e = 2.73\ldots, f'(x) = e^x$$

Basic calculus notions

Let's see some basic ways to compute the **derivate** of a function f, denoted by **f'**:

5. The derivative of a trigonometric functions:

$$f(x) = \sin x\,,\ f'(x) = \cos x\,,$$
$$f(x) = \cos x\,,\ f'(x) = -\sin x$$

Basic calculus notions

Let's see some basic ways to compute the **derivate** of a function f, denoted by **f'**:

6. The derivative of summed/substracted functions:

$$f(x) = g(x) + h(x), g, h: R \to R, then\ f'(x) = g'(x) + h'(x)$$

$$or\ more\ genericaly\ f(x) = \sum_{i=1}^{n} g_i(x), f'(x) = \sum_{i=1}^{n} g'_i(x)$$

Basic calculus notions

Let's see some basic ways to compute the **derivate** of a function f, denoted by **f'**:

6. The derivative of summed functions:

Example:

Let's assume we have the following function: $f(x) = 2x^3 + 5$

We can say the $f(x) = g(x) + h(x)$ where $g(x) = 2x^3$ and $h(x) = 5$

Then $f'(x) = g'(x) + h'(x)$.

For $g(x)$ we have: $g'(x) = 2 \times 3 \times x^{3-1} = 6 \times x^{3-1} = 6x^2$. For $h(x)$ we have: $h'(x) = 0$

As a result: $\boldsymbol{f'(x) = g'(x) + h'(x) = 6x^2 + 0 = 6x^2}$

Basic calculus notions

Let’s see some basic ways to compute the **derivate** of a function f, denoted by **f’**:

7. The chain rule

$$f(x) = g\big(h(x)\big), \qquad with\ g, h: R \to R,$$
$$then\ f'(x) = g'\big(h(x)\big) \times h'(x)$$

OBS: this rules refers to chained functions (a function that gets an input, returns an output and then the output is the input for another function)

Basic calculus notions

Let's see some basic ways to compute the **derivate** of a function f, denoted by **f'**:

7. The chain rule

Example:

Let's assume we have the following functions: $h(x) = x^2, g(x) = 2x + 7$

and we want to compute: $f(x) = g(h(x))$.

For $h(x) = x^2, h'(x) = 2x$. For $\mathrm{g}(x) = 2x + 7, g'(x) = 2$

Then $\boldsymbol{f'(x) = g'(h(x)) \times h'(x) = 2 \times 2x = 4x}$

Basic calculus notions

Let's try an example to understand where this computations help.

Let's consider the following function: $f(x) = \sin x + x^2$

Given a specific value for "x" → lets assume x=2, can we find out in what direction (increase or decrease) we should modify x (so basically adding something or subtracting something) so that *f(x)* will be smaller.

In other words, we are interested in a value "k" with the following property:

$$f(x) = \sin x + x^2, f(x + k) < f(x)$$

Basic calculus notions

Let's try an example to understand where this computations help.

Let's consider the following function: $f(x) = \sin x + x^2$

Given a specific value for "x" → lets assume x=2, can we find out in what direction (increase or decrease) we should modify x (so basically adding something or subtracting something) so that *f(x)* will be smaller.

In other words, we are interested in a value "k" with the following property:

$$f(x) = \sin x + x^2, f(x + k) < f(x)$$

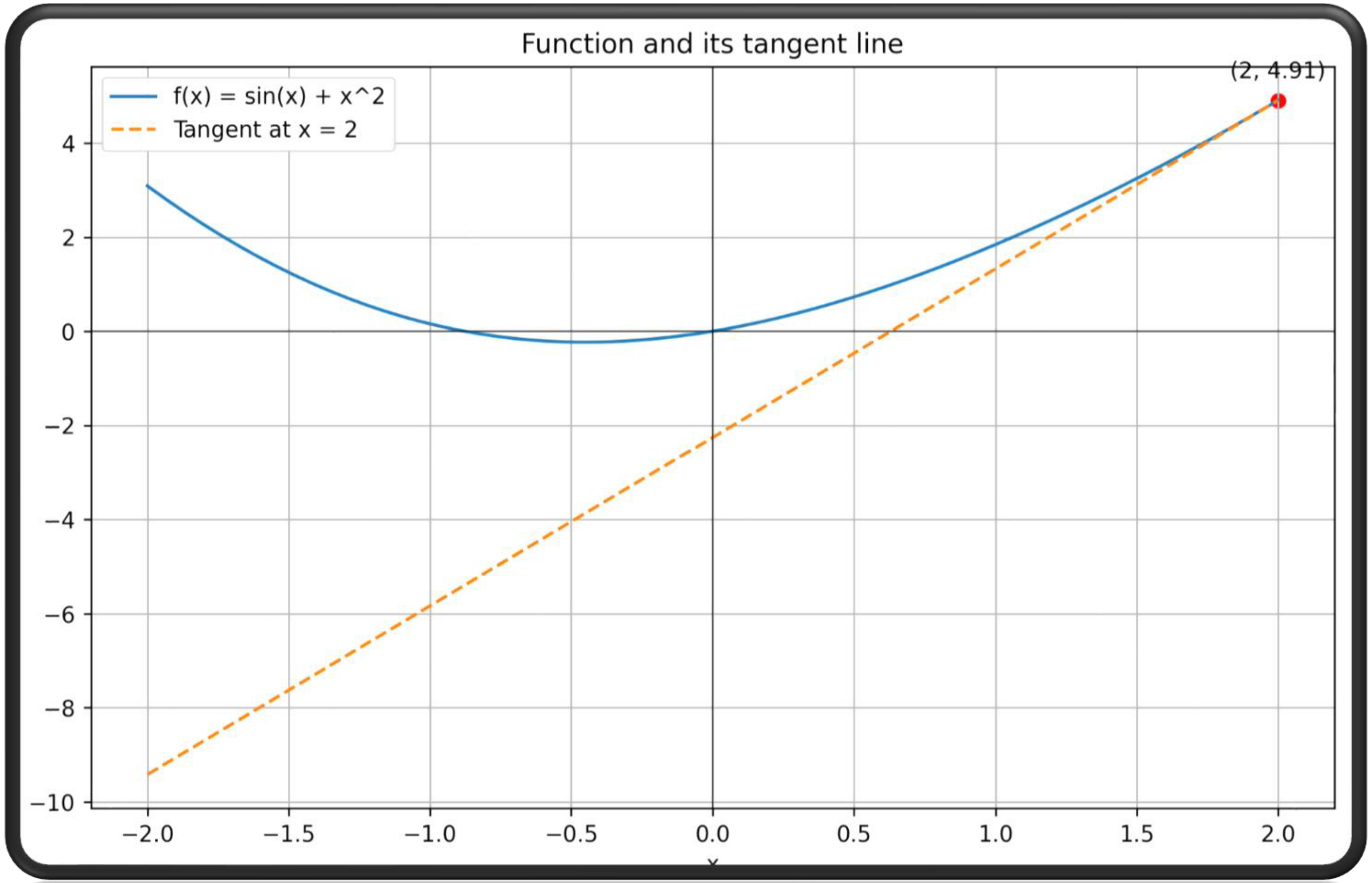
Basic calculus notions

Let's solve this problem step by step:

1. Find the derivative of f: $f(x) = \sin x + x^2$, $f'(x) = \cos x + 2x$
2. Compute the value of $f'(x)$ for x=2,
$$f'(2) = \cos 2 + 2 \times 2 = -0.41 + 4 \cong 3.58$$
3. If $f'(x)$ is positive (>0), the function is increasing so we should look for a value that is smaller than x. if $f'(x)$ is negative (<0) the function is decreasing so we should look for a value that is bigger than x.
4. In out case, $f'(2) \cong 3.58$ is positive, so in order decrease its value we should look for a value smaller than "2" to reduce the value of f.
5. In other words, the factor "k" from the previous slide should have the reverse sign of $f'(x)$, or in order minimize function f we should apply the following rule: $\boldsymbol{x = x + \alpha \times \left(-sign(f'(x))\right)}$

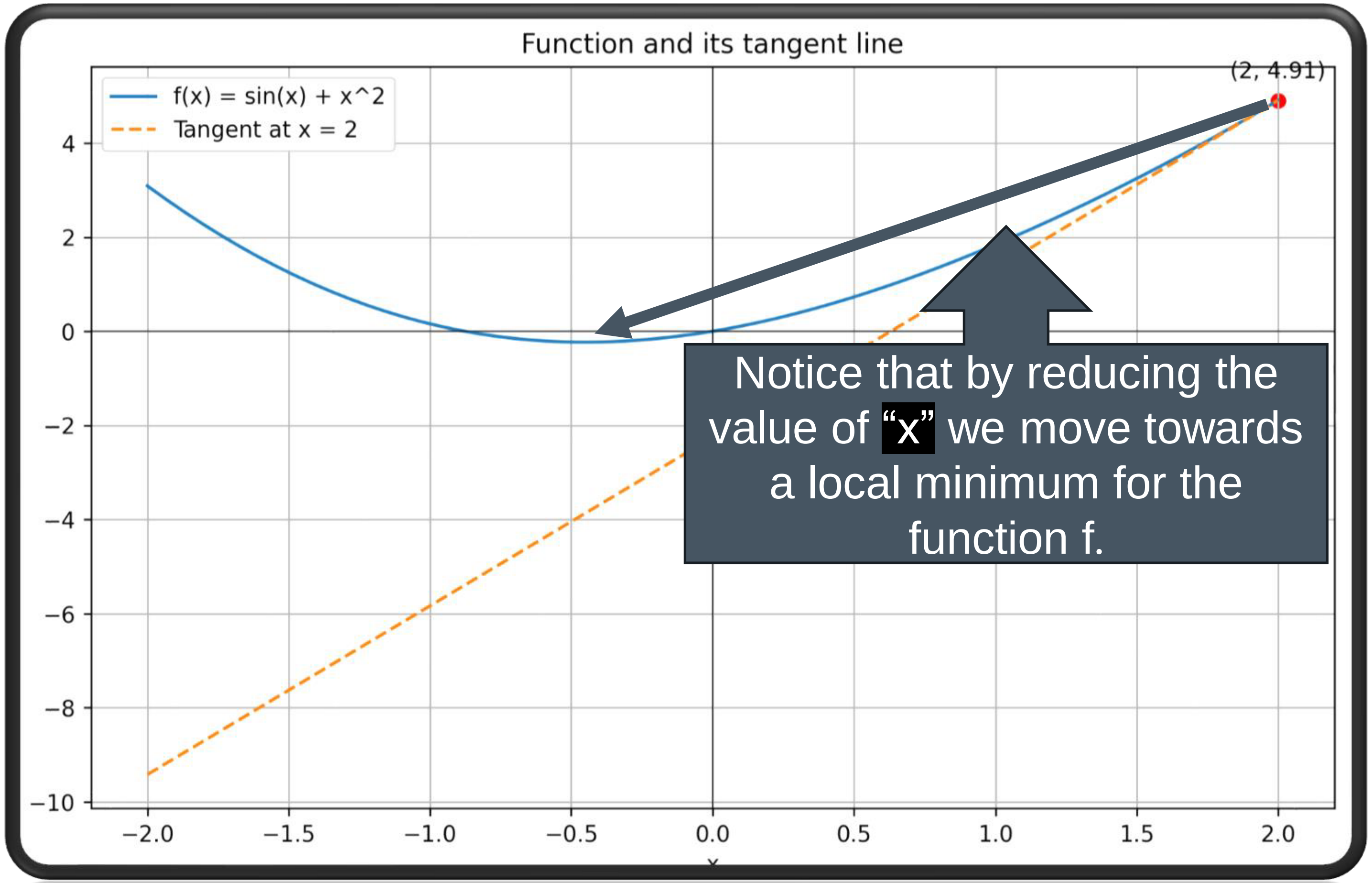
Basic calculus notions

Let’s see a graphical representation of the previous problem:



Basic calculus notions

Let’s see a graphical representation of the previous problem:

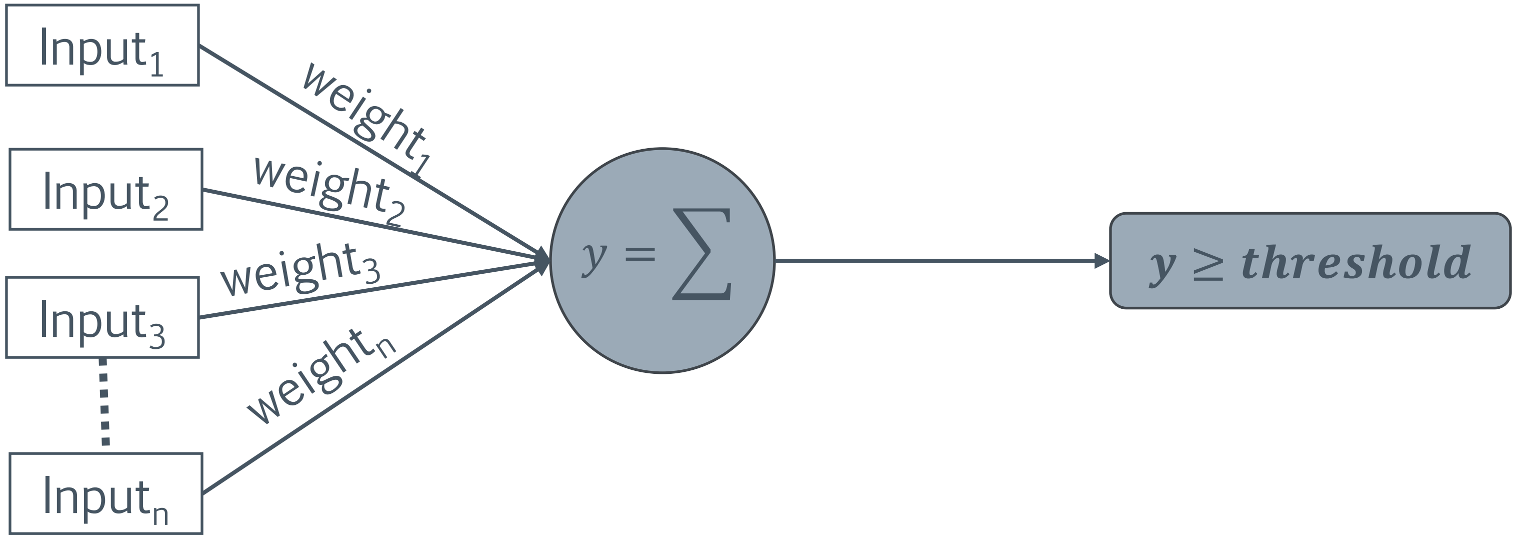


Notice that by reducing the value of “x” we move towards a local minimum for the function f.

Neuron unit

Neuron unit

We know that a perceptron can be described in the following way:



$$y = \sum_{i=1}^{n} Input_i \times weight_i\,, y \geq threshold$$

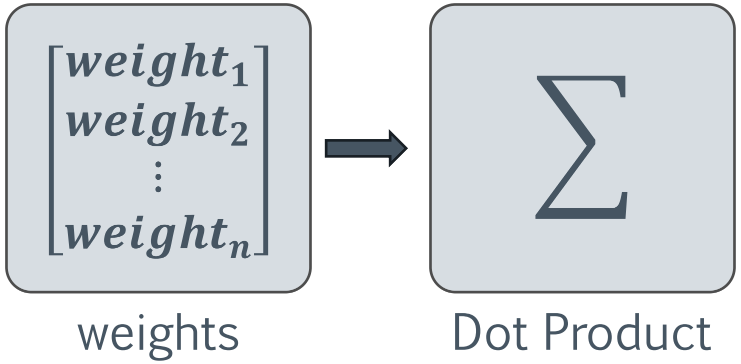
Neuron unit

But we can view a perceptron as a unit formed out of the following components:

- **Weights** (a vector of weights)
- **Sums up function** (a function that computes the dot product between the vector of weights and its input)
- **Activation function** (a function that takes the result of the previous function and convert that scalar value into another value).

Neuron unit

This means that we can organize a perceptron in the following layers:



This is the dot product between the weights and the inputs.

$$\sum_{i=1}^{n} weight_i \times input_i = weights \cdot inputs$$

Neuron unit

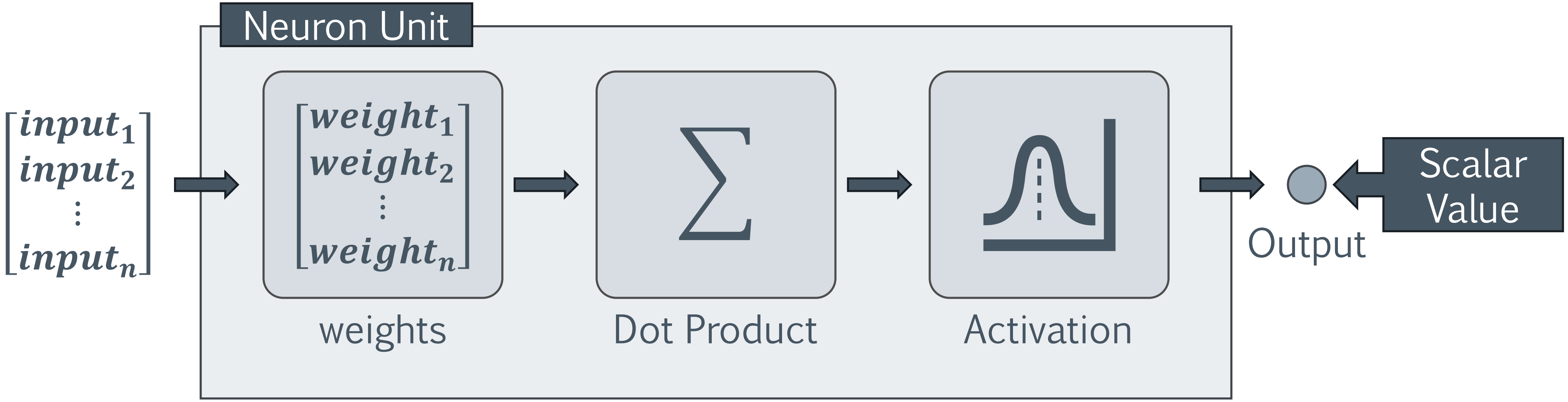
This means that we can organize a perceptron in the following layers:

This could be any kind of function that takes a scalar and returns another scalar. It is used for normalization / activation / etc. It is important that this function is **derivable**.

$$output = f(x)$$

Neuron unit

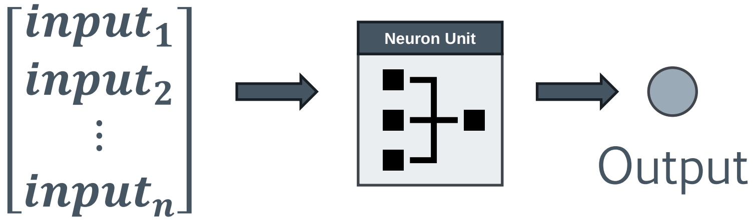
The full flow looks like this:



Further in this course we will use this unit as the basic component of a neural network

Neuron unit

We can use the following graphical representation for a neuron unit:



Or a mathematical representation:

$$NeuronUnit\colon R^n \to R, NeuronUnit(input) \to output,$$

$$NeuronInput\left(\begin{bmatrix} \boldsymbol{input_1} \\ \boldsymbol{input_2} \\ \vdots \\ \boldsymbol{input_n} \end{bmatrix}\right) = output$$

Neuronal Network Architecture

Neuronal Network Architecture

A neuronal network is a network formed out of multiple neurons organized in layers as follows:

1. An **input layer** (a set of neurons – at least one - that receive the input)

2. **Hidden layers** (one or more layers of neurons that process the output from the input layer)

3. **Output layer** (a final layer that takes values from the previous layer and convert it into the final value).

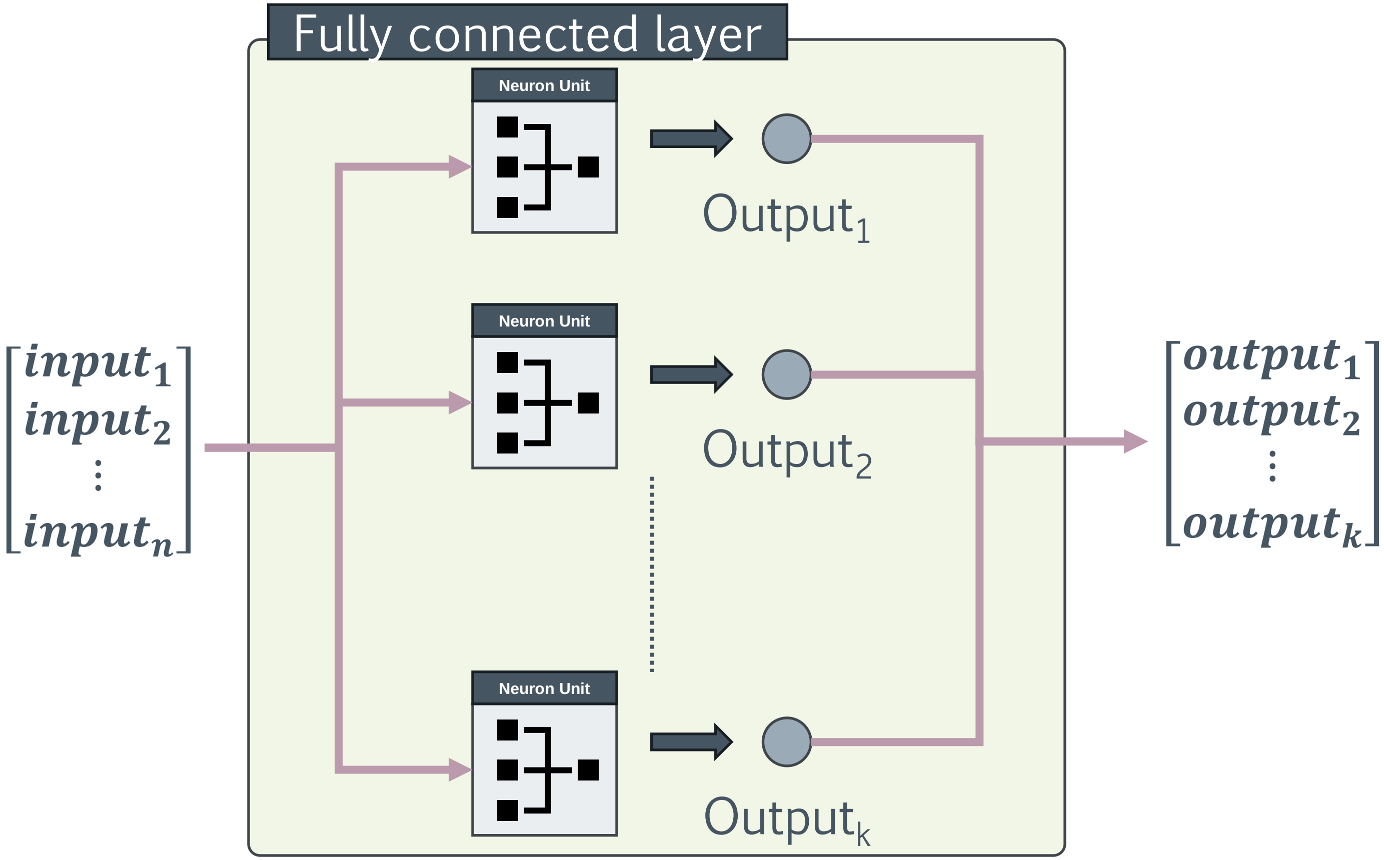
Neuronal Network Architecture

Observations:

- The input layer must have **at least one neuron**
- **Hidden layers are optional**
- The number of neurons in the hidden layers can **vary** from one hidden layer to another
- **Output layer usually consist in one neuron**, but for multi-class classification it can contain multiple neurons
- A neuronal network with one neuron in the input layer, no hidden layers and no output layers is essentially a perceptron

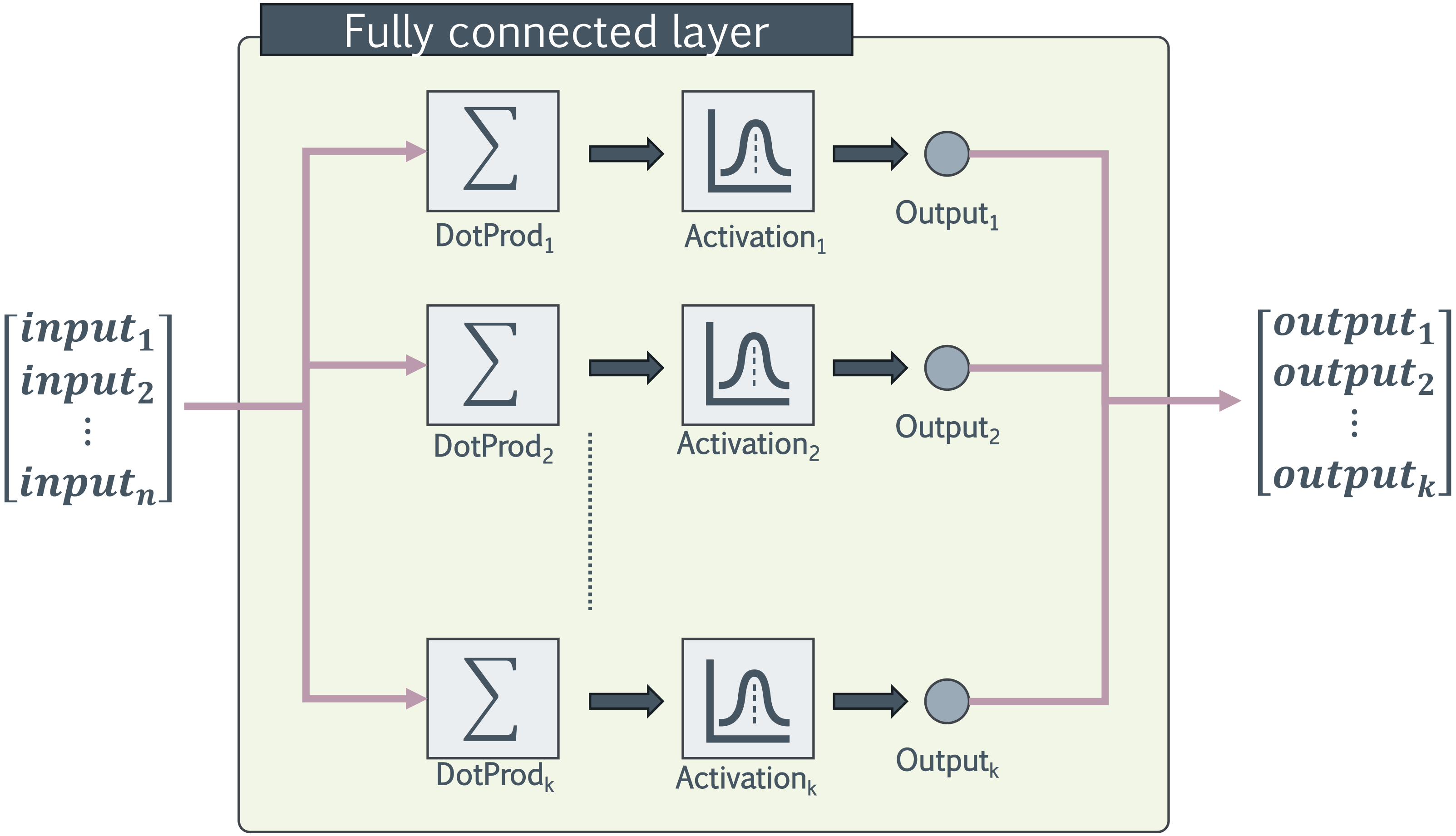
Neuronal Network Architecture

Let's see a graphical representation of a layer within a neuronal network (for the time being we will consider a fully connected neuronal network):



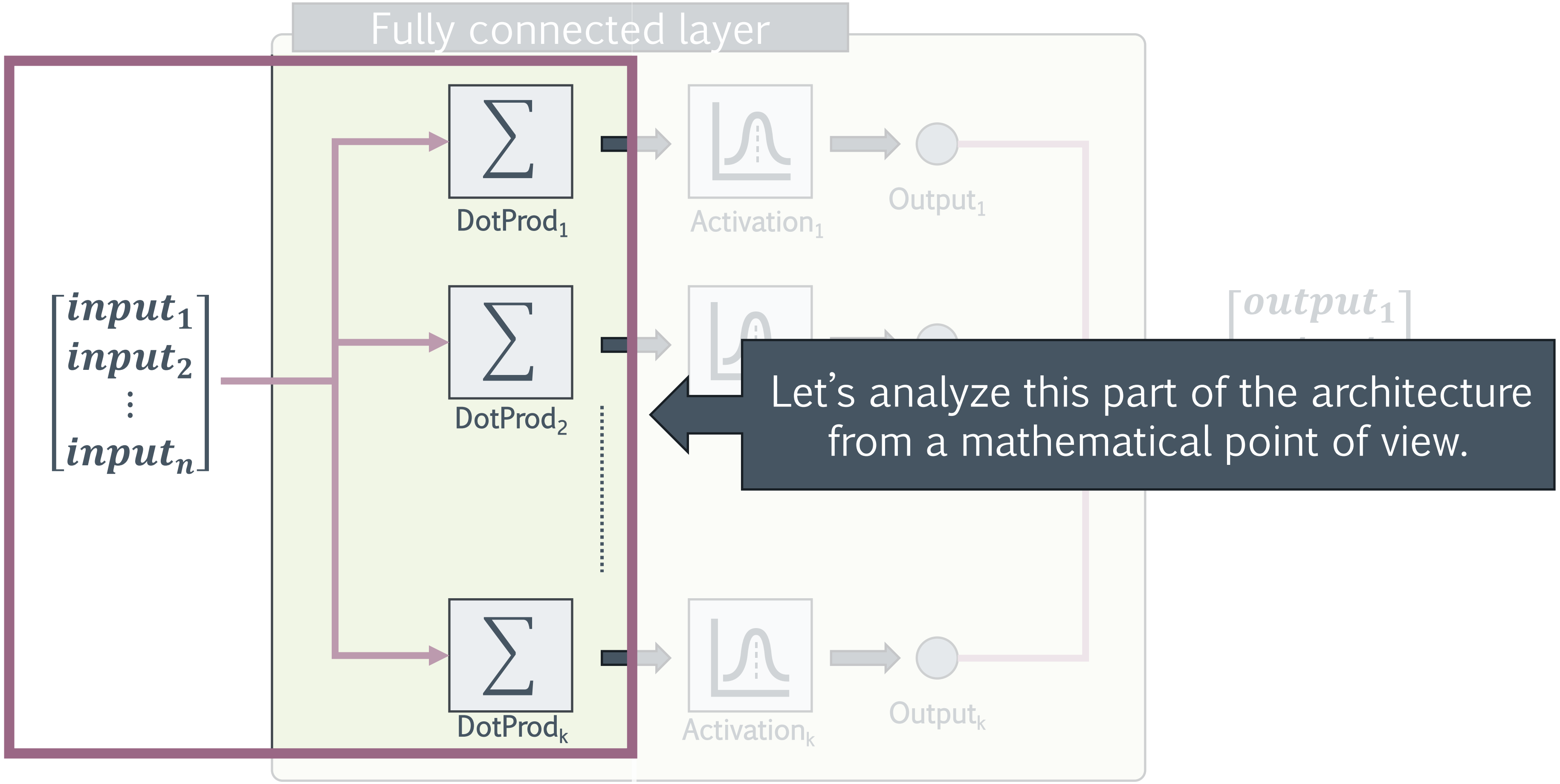
Neuronal Network Architecture

A more detailed view of this architecture looks like this:



Neuronal Network Architecture

A more detailed view of this architecture looks like this:



Neuronal Network Architecture

A more detailed view of this architecture looks like this:

$\sum$ $DotProd_1$ $= input \cdot NeuronUnit_1.Weights$

$\sum$ $DotProd_2$ $= input \cdot NeuronUnit_2.Weights$

$\vdots$

$\sum$ $DotProd_k$ $= input \cdot NeuronUnit_k.Weights$

While all of these “k” dot products produces individual scalar values, we can also consider that each one of this operations in fact produce a part of a vector.

$$= \begin{bmatrix} input \cdot NeuronUnit_1.Weights \\ input \cdot NeuronUnit_2.Weights \\ \vdots \\ input \cdot NeuronUnit_k.Weights \end{bmatrix}$$

This looks like a **matrix multiplication** operation !

Neuronal Network Architecture

A more detailed view of this architecture looks like this:

$$\underset{DotProd_1}{\boxed{\sum}} = input \cdot NeuronUnit_1.Weights$$

$$\underset{DotProd_2}{\boxed{\sum}} = input \cdot NeuronUnit_2.Weights$$

$$\vdots$$

$$\underset{DotProd_k}{\boxed{\sum}} = input \cdot NeuronUnit_k.Weights$$

$$= \begin{bmatrix} \boldsymbol{input_1} \\ \boldsymbol{input_2} \\ \vdots \\ \boldsymbol{input_n} \end{bmatrix} \times \begin{bmatrix} Weight_{(1,1)} & \cdots & Weight_{(1,n)} \\ \vdots & \ddots & \vdots \\ Weight_{(k,1)} & \cdots & Weight_{(k,n)} \end{bmatrix}$$

Neuronal Network Architecture

A more detailed view of this architecture looks like this:

$\boxed{\sum}$ $= input \cdot NeuronUnit_1.Weights$

DotProd$_1$

$\boxed{\sum}$ $= input \cdot NeuronUnit_2.Weights$

DotProd$_2$

$\vdots$

$\boxed{\sum}$ $= input \; \boxed{NeuronUnit_k.Weights}$

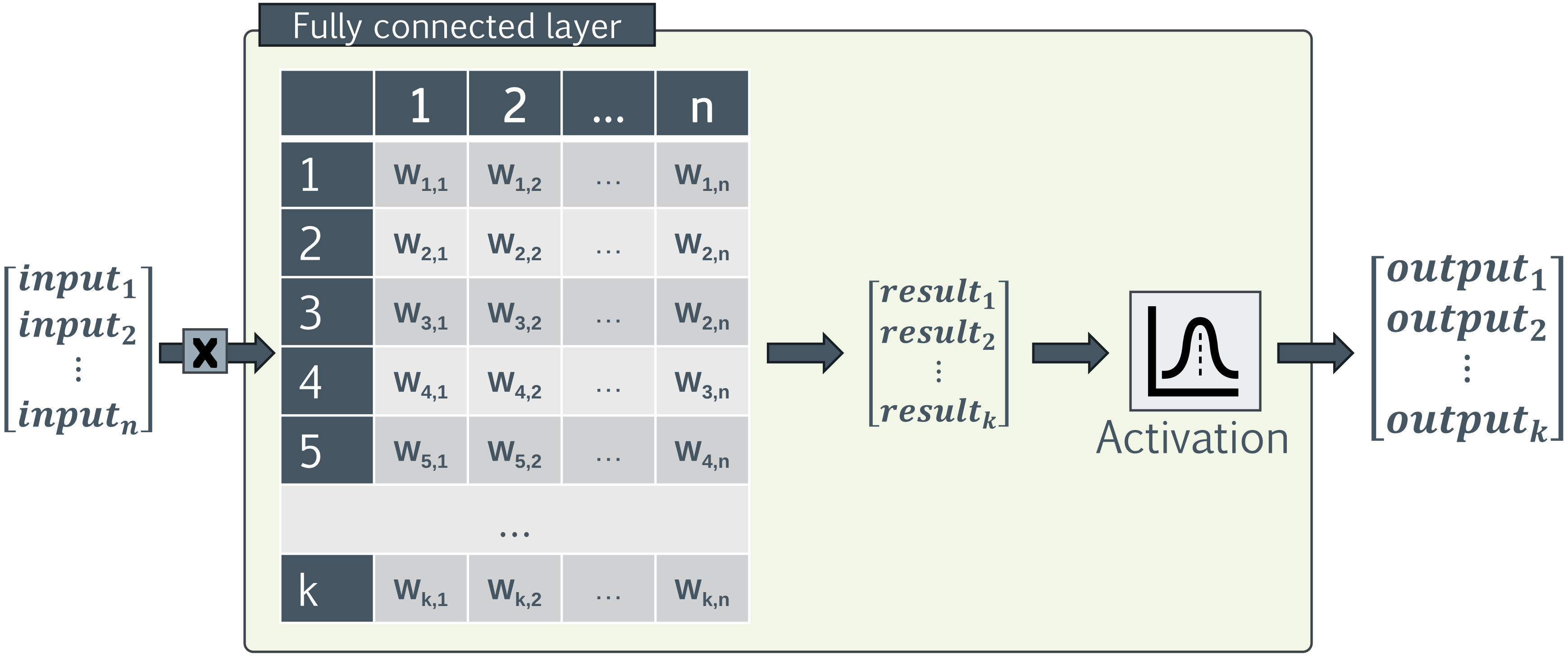
DotProd$_k$

$$= \begin{bmatrix} \boldsymbol{input_1} \\ \boldsymbol{input_2} \\ \vdots \\ \boldsymbol{input_n} \end{bmatrix} \times \begin{bmatrix} Weight_{(1,1)} & \cdots & Weight_{(1,n)} \\ \vdots & \ddots & \vdots \\ Weight_{(k,1)} & \cdots & Weight_{(k,n)} \end{bmatrix}$$

Where **$Weight_{(k,1)}$…$Weight_{(k,n)}$** are the weights associates with ***$NeuronUnit_k$***

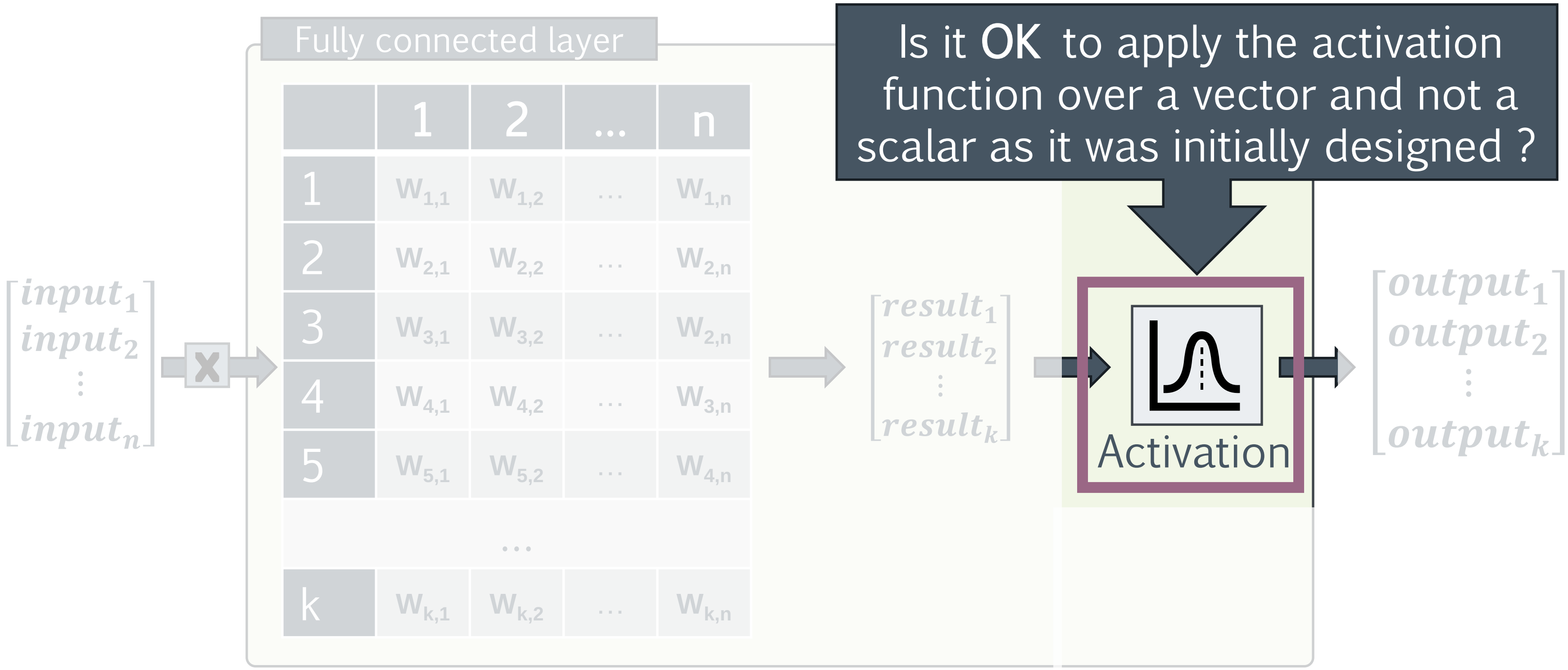
Neuronal Network Architecture

So, in fact, we can see a part of the original architecture presented as follows:



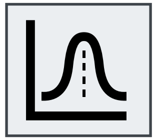
Neuronal Network Architecture

So, in fact, we can see a part of the original architecture presented as follows:



Neuronal Network Architecture

The activation function was defined as follows:



Activation

$$output = activation(x), with\ x, y = scalars$$

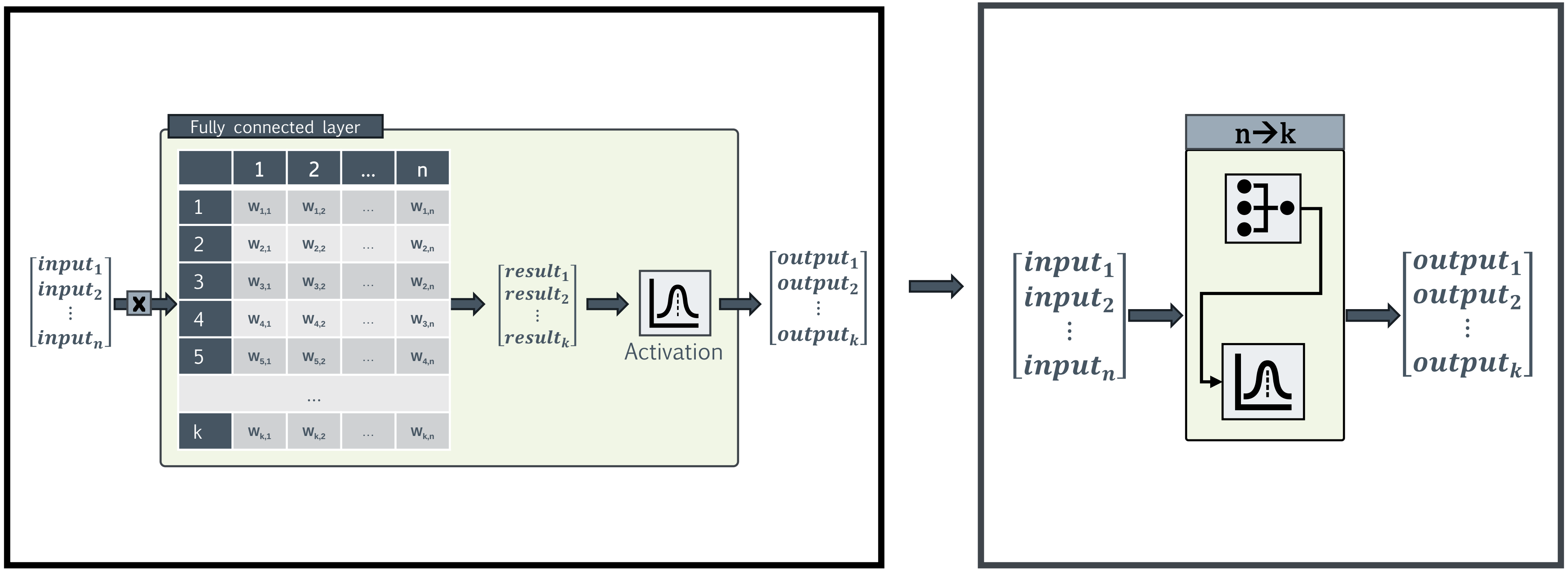
We can convert it to serve the same purpose as follows:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} ouput = \begin{bmatrix} activation(x_1) \\ activation(x_2) \\ \vdots \\ activation(x_n) \end{bmatrix}$$

The main advantage when using this type of form is that you can **also** apply the activation function to the entire vector and not just for individual scalars.

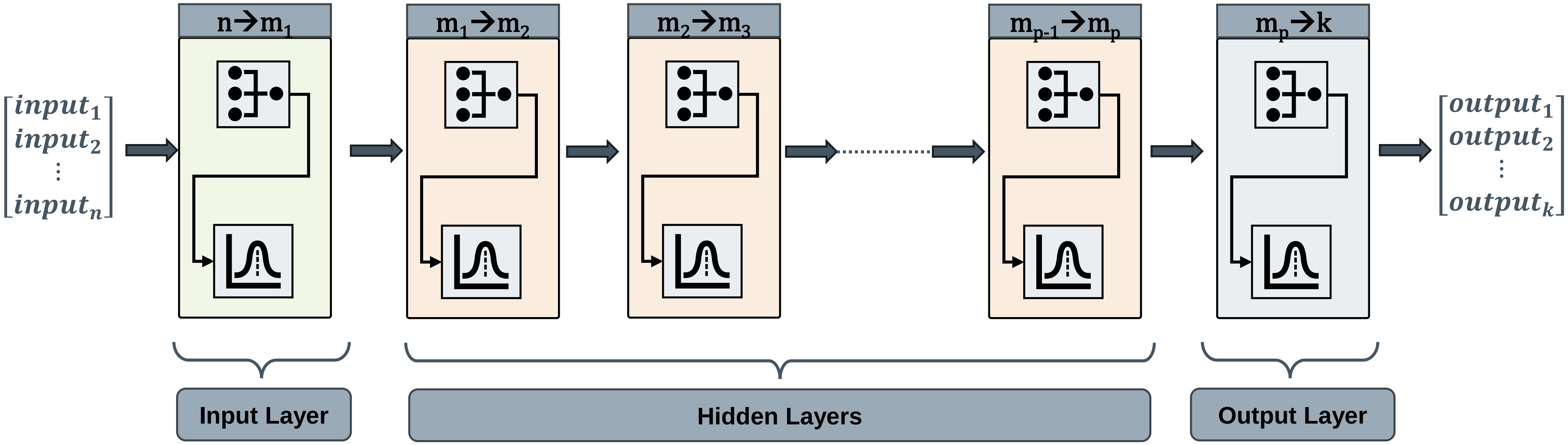
Neuronal Network Architecture

Or a more simplified version of a fully connected layer:



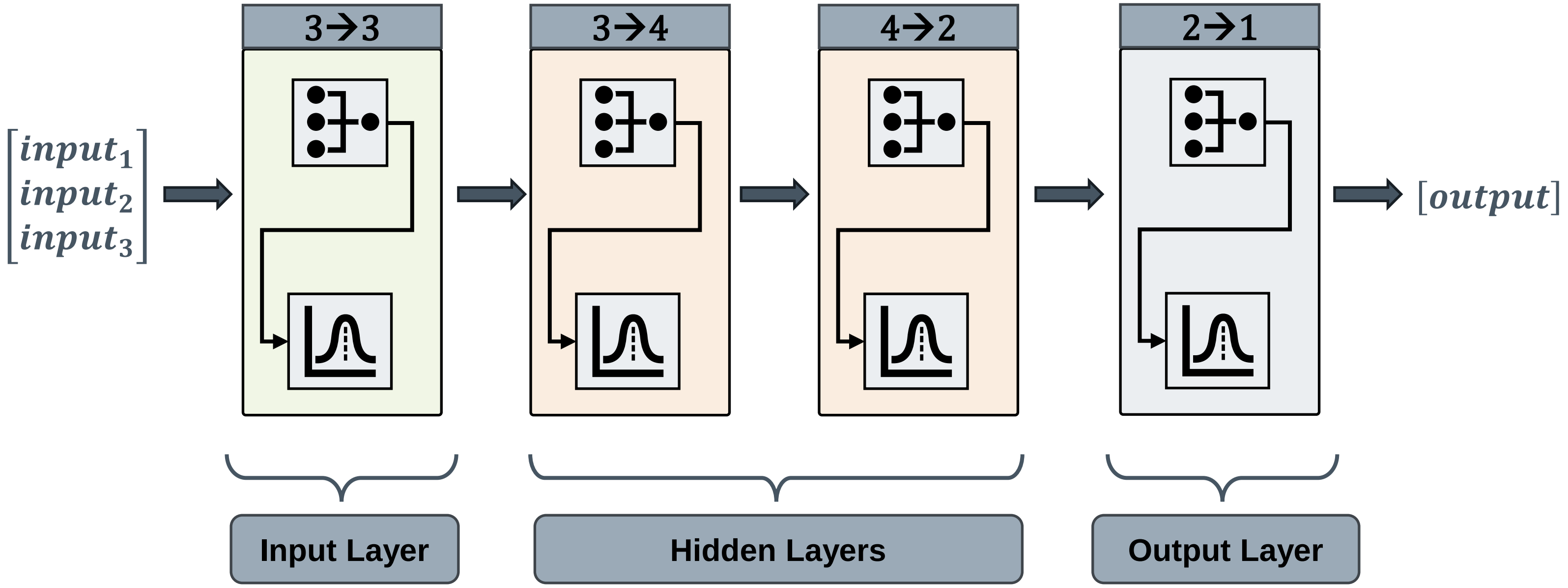
Neuronal Network Architecture

As a result, we can describe a fully connected neuronal network as follows:



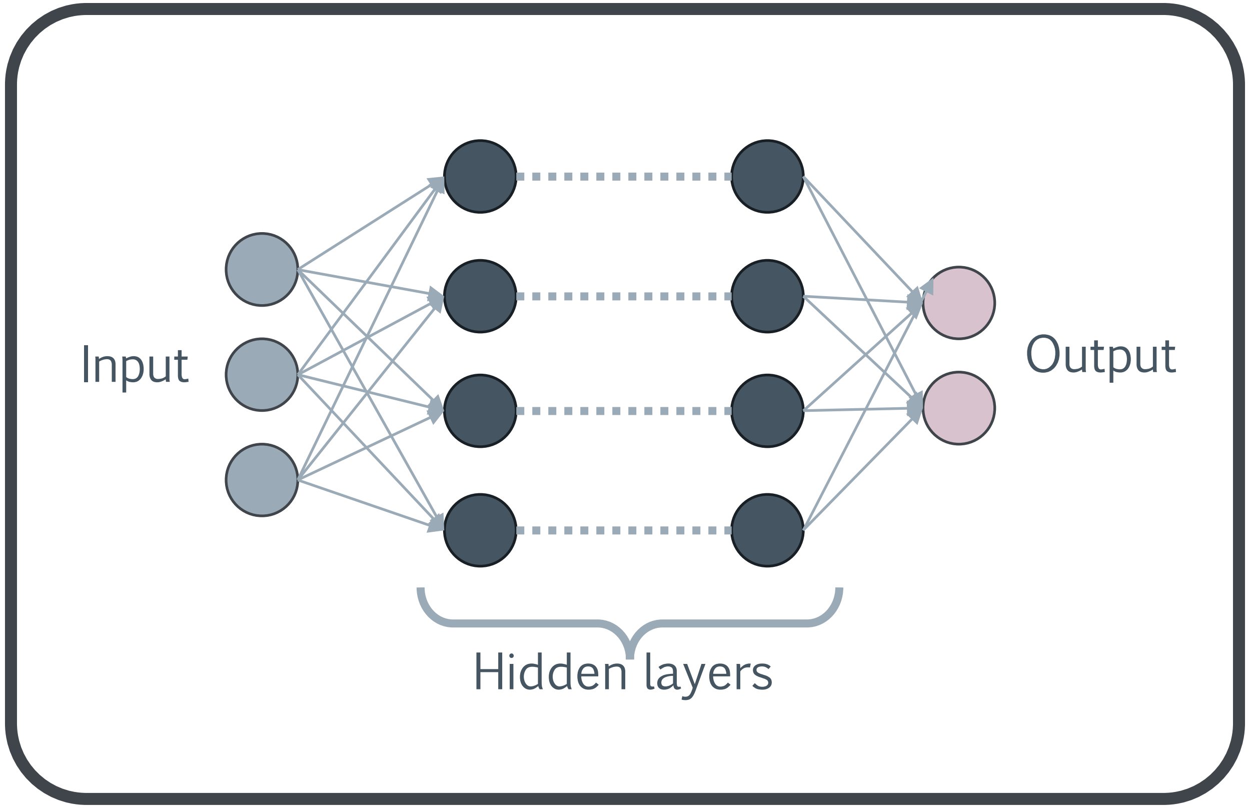
Neuronal Network Architecture

Let’s see a more practical example with a fully connected neuronal network with 3 inputs, 2 hidden layers of 4 and 2 neurons and one output.



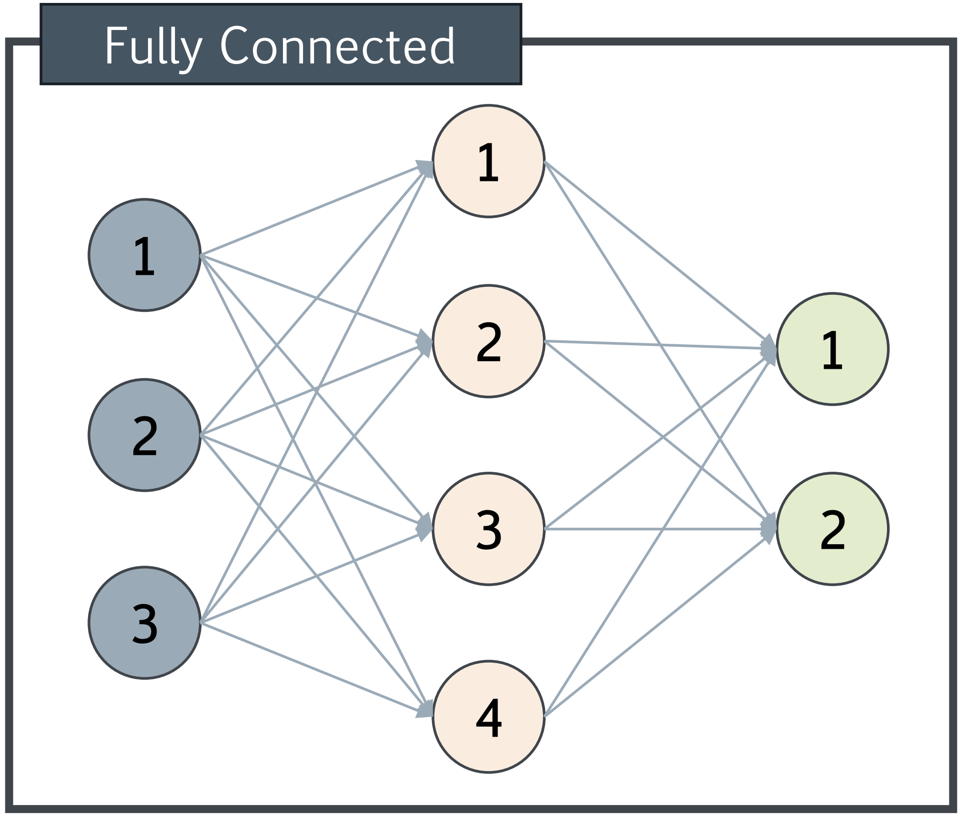
Neuronal Network Architecture

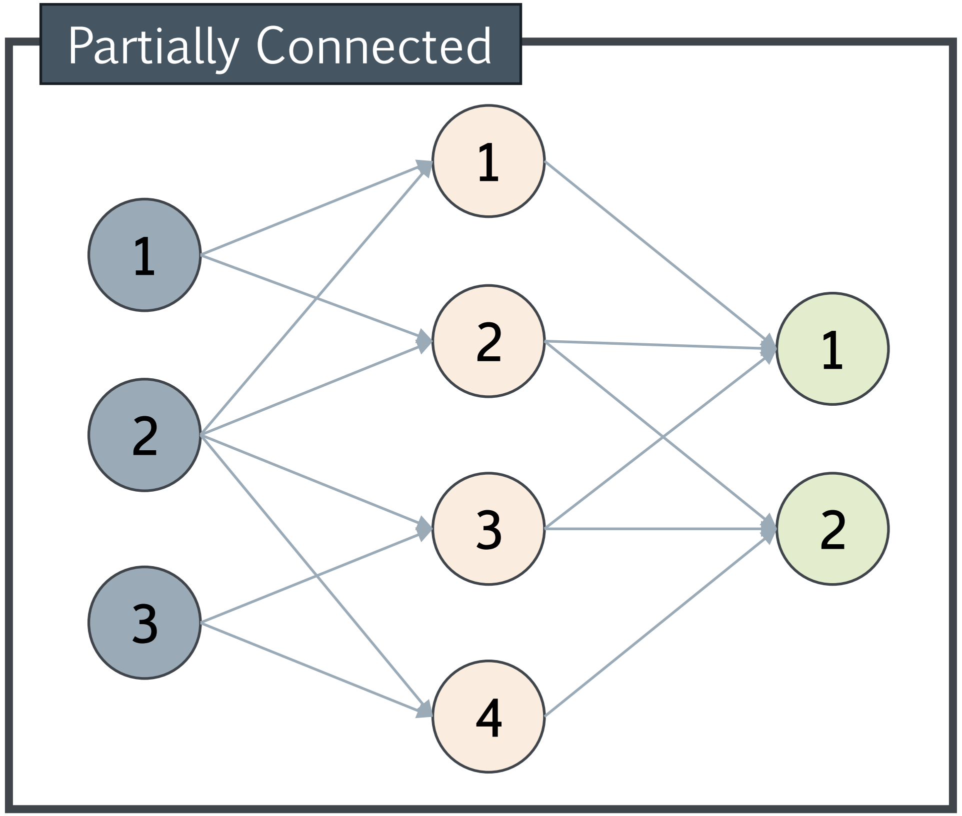
Other graphical representations look like this:



Neuronal Network Architecture

It is also important to notice that a neuronal network can be fully connected (or dense) or not.





Activation functions

Activation functions

Activation functions were name like this because their initial representation meant converting a value to 0 or to something else (different than 0) that will reflect an activation.

In practice, some of the activation functions don’t do this, they just transform the input into another value (for example there are activation functions that perform normalization tasks).

Activation functions

1. Identity function:

$$activation(x) = x$$

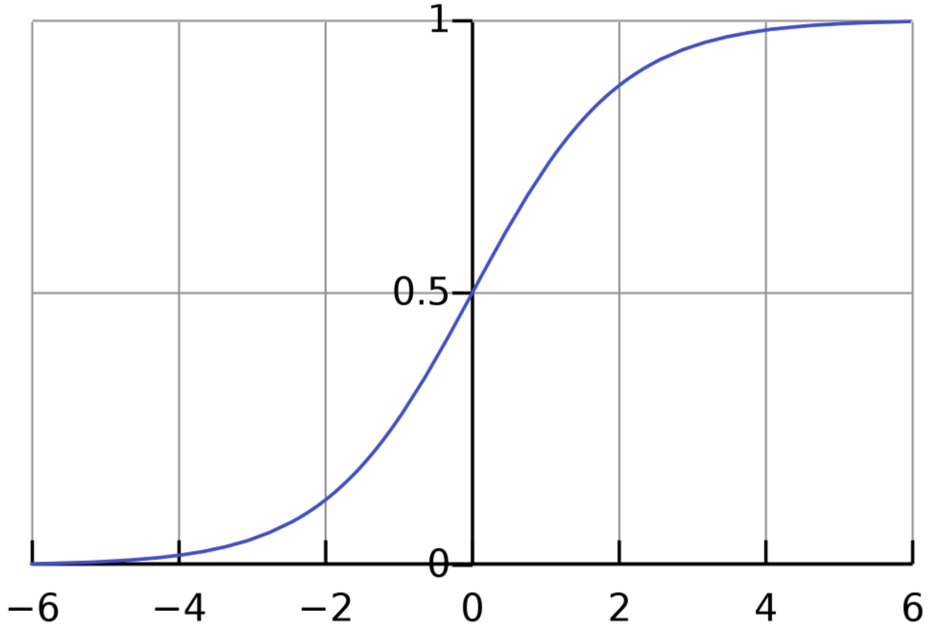
This is not really a function that is being used, more like a method to explain that no activation is needed.

Also called **linear activation**.

Activation functions

2. Sigmoid function:

$$activation(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$



The slope of a sigmoid:

$$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2}$$

We will discuss in the last part of this course why this formula is needed.

Activation functions

2. Sigmoid function:

- This function normalizes values in 0..1 interval. The larges a value is, the close this functions tends to 1. The lower a number is (into the negative space) the closest the result of this function will be towards 0.

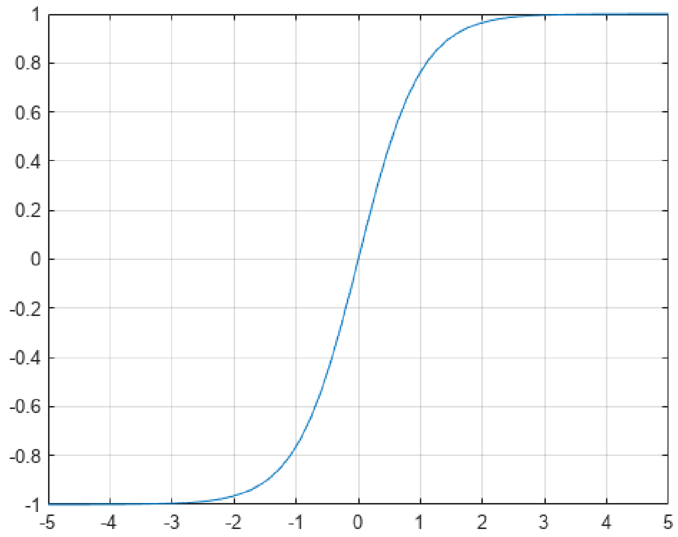
- Sigmoid functions are commonly used in the output layer of binary classification neural networks, where they model the probability that an input belongs to one of the two classes.

- For large values (positive or negative) the slope of the sigmoid function tends to 0 and results in a slow convergence during training (phenomena called **vanishing gradient**).

Activation functions

3. Hyperbolic Tangent (tanh):

$$activation(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Activation functions

3. Hyperbolic Tangent (tanh):

- This function normalizes values in -1..1 interval. The larges a value is, the close this functions tends to 1. The lower a number is (into the negative space) the closest the result of this function will be towards -1.

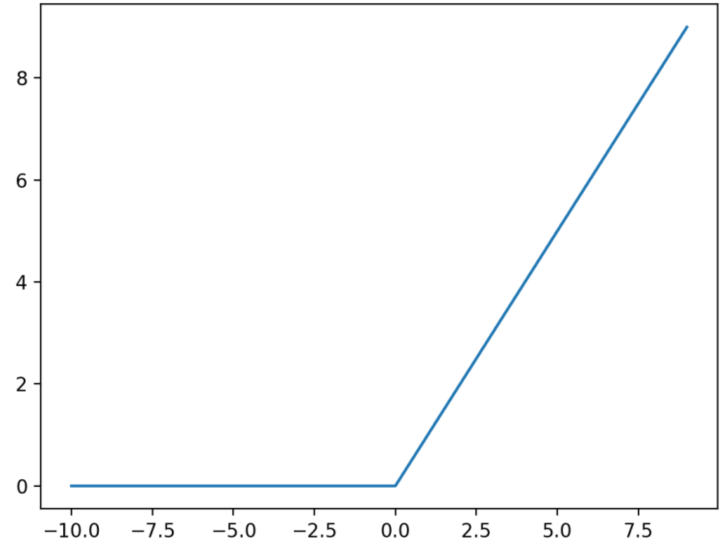
- Has the same **vanishing gradient** problem as the sigmoid function has.

- The ***tanh*** function is often used in neural networks for various tasks, including classification, regression, and recurrent neural networks (RNNs). It can be particularly useful in cases where the zero-centered property is desirable.

Activation functions

4. Rectified Linear Unit (ReLU)

$$activation(x) = ReLU(x) = \max(0, x$$



With several variations:

- Leaky ReLU
- Parametric ReLU
- Exponential Linear Unit (ELU)
- Scaled Exponential Linear Unit (SELU)

Activation functions

4. Rectified Linear Unit (ReLU) :

- 0 for negative values and linear for positive values

- It is not affected by **vanishing gradient** problem

- Might improve a neuronal network. If the output of a neuron is 0, then the computation from that moment is easier.

- ReLU and its variants are commonly used in convolutional neural networks (CNNs) for image processing tasks, as well as in various other deep learning architectures for tasks such as natural language processing and reinforcement learning

Activation functions

5. SoftMax:

For v = [v_1,v_2,..v_n] a vector we can normalize it in the following way:

$$softmax(v) = [f(1), f(2), \dots f(n)], with\ f(i) = \frac{e^{v_i}}{\sum_{j=1}^{n} e^{v_j}}, i \subset [1..n]$$

As a general observation, this functions normalizes a vector and translates it into a of set of percentages. This is important as it makes understanding how relevant is every value from a vector.

$$softmax(v) = [f(1), f(2), \dots f(n)], and \sum_{i=1}^{n} \left(\frac{e^{v_i}}{\sum_{j=1}^{n} e^{v_j}}\right) = 1$$

Activation functions

5. SoftMax:

Example:

Let v = [1,3,4,2]
We first compute the exponents:

- $e^1 \sim= 2.71$
- $e^3 \sim= 20.08$
- $e^4 \sim= 54.59$
- $e^2 \sim= 7.38$

$$let\ \boldsymbol{sum} = \sum_{j=1}^{4} e^{v_j} = 2.71 + 20.08 + 54.59 + 7.38 = 84.75$$

Then:

$$softmax(v) = \left[\frac{e^1}{sum}, \frac{e^3}{sum}, \frac{e^4}{sum}, \frac{e^2}{sum}\right] = \left[\frac{2.71}{84.75}, \frac{20.08}{84.75}, \frac{54.49}{84.75}, \frac{27.38}{84.75}\right] = [0.03, 0.24, 0.64, 0.09\ ]$$

Activation functions

5. SoftMax:

Example:

So , for v = [1,3,4,2], ***softmax***(v) = [0.03, 0.24, 0.64, 0.09]

We can also see ***softmax***(v) as [3%, 24%, 64%, 9%].

This allows us to see what is most important element in the vector (the 3rd).

OBS: since we are using exponents, we can use -∞ to force a value in the vector to translate into absolute 0 in the result.

$$e^{-\infty} = 0$$

Activation functions

5. SoftMax:

Practical observation: for v = [v_1,v_2,..v_n] a vector:

$$softmax(v) = [f(1), f(2), \ldots f(n)], with\ f(i) = \frac{e^{v_i}}{\sum_{j=1}^{n} e^{v_j}}, i \subset [1..n]$$

What happens if one or multiple elements in the vector are too large ? (for example: v = [1,2,3,1000000]. The problem here is that computing $e^{1000000}$ will not scale in 64/128 bits. As such we need to normalize the formula for *f(i)* in the following way:

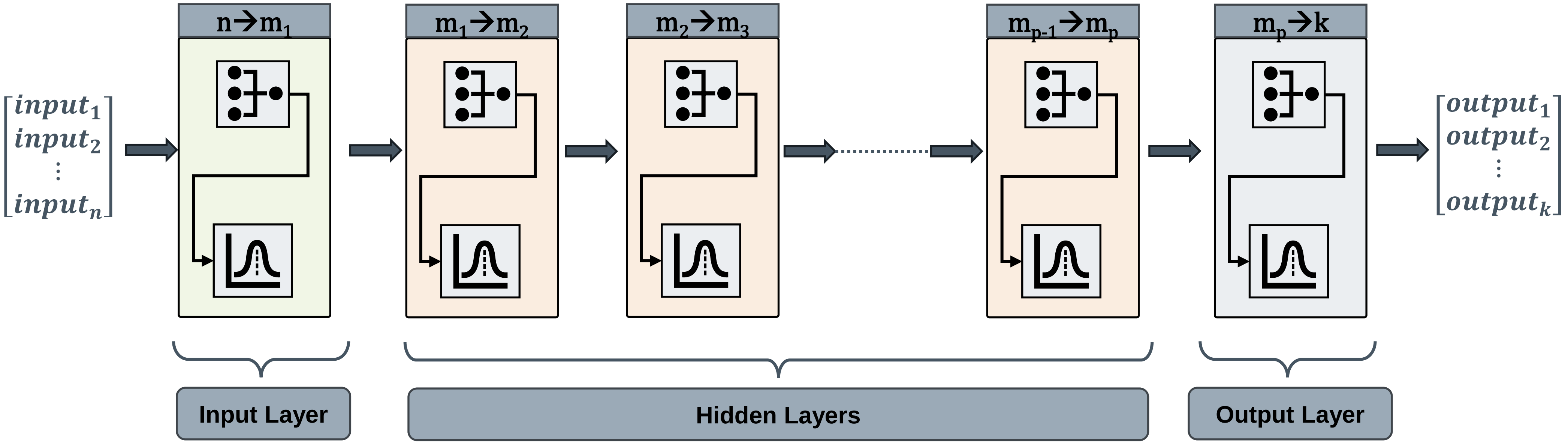
$$f(i) = \frac{\frac{e^{v_i}}{k}}{\sum_{j=1}^{n} \frac{e^{v_j}}{k}}, i \subset [1..n], with\ k\ a\ large\ numerical\ constant,$$

$$often\ k = \max_i e^{v_i}$$

Feed forward step

Feed forward step

› In the end, a neuronal network is formed out of layers. So, we can consider a generic function (**compute_output**) that takes an input and returns an output.



Feed forward step

The simple's way to envision the evaluation result from a neuronal network is as following:

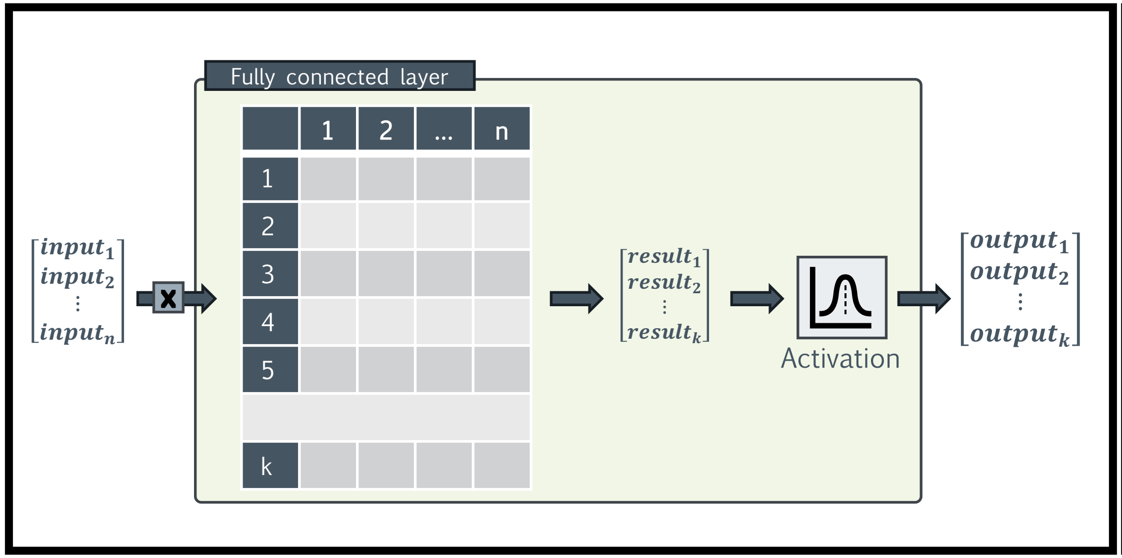
Evaluate function

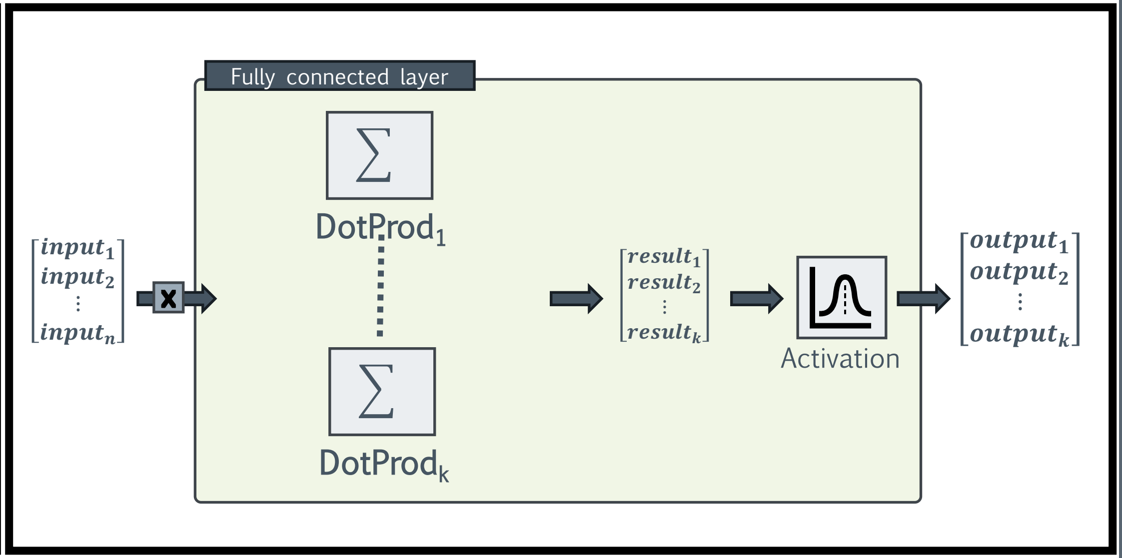
```
function evaluate(input, layers)
  foreach layer in layers
    output ← compute_output (input, layer)
    input ← output
  end foreach
  return output
end function
```

Where both ***input*** and ***output*** are vectors.

Feed forward step

So … let's define the ***compute_output*** function. In this case we may have two variants to write it:



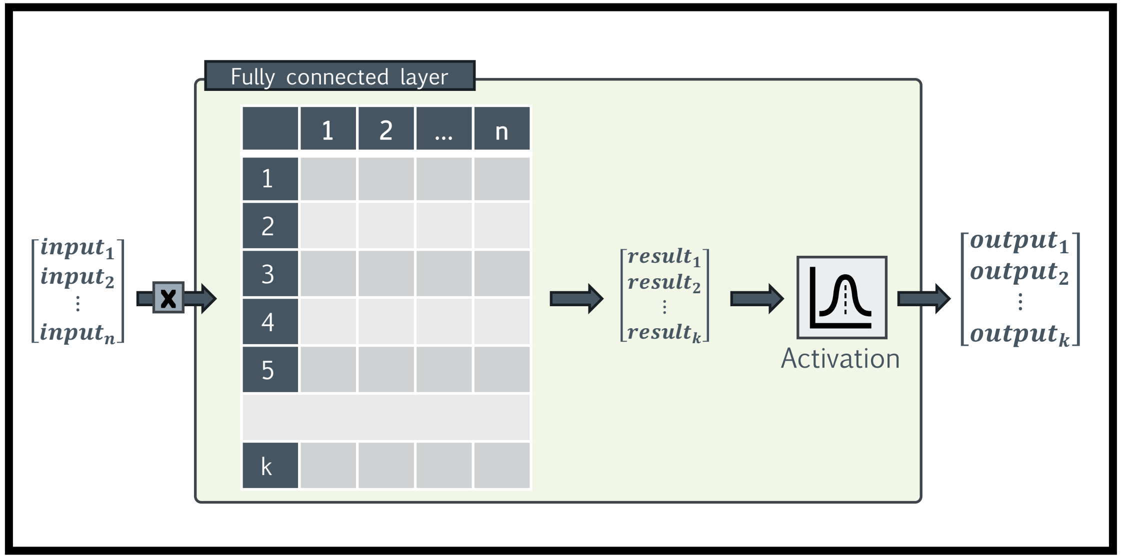


Using matrix operations (all weights from the neurons from one layer are stored in one matrix).

Using dot product over each neuron from the layer.

Feed forward step

So … let’s define the ***compute_output*** function. In this case we may have two variants to write it:



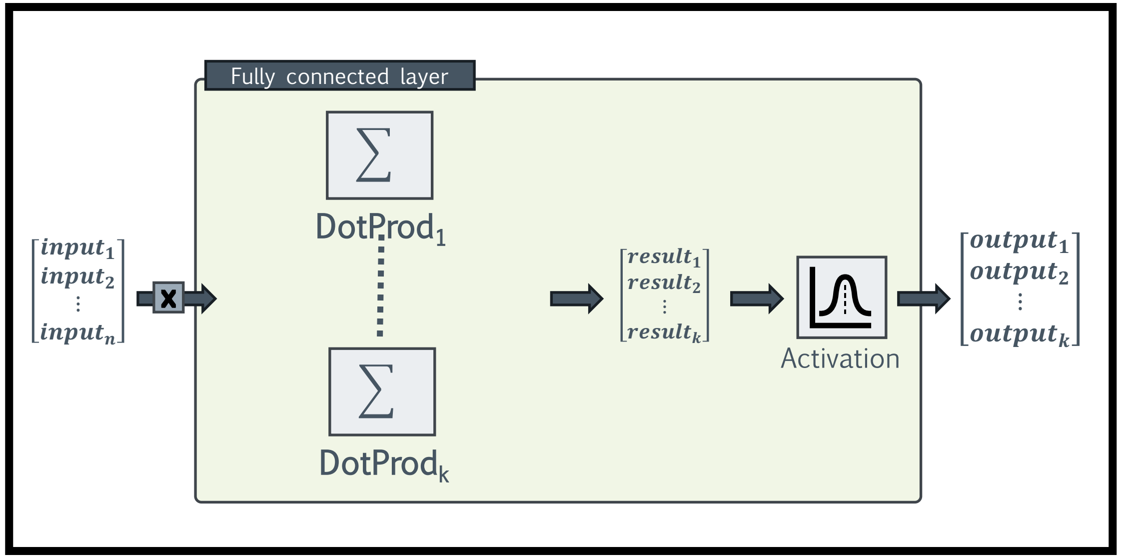
 # Evaluate function

```
function compute_output(input, layer)
  result ← layer.matrix x input
  return activation(result)
end function
```

In this case the code is quite simple as it only performs matrix operations.

Feed forward step

So … let's define the ***compute_output*** function. In this case we may have two variants to write it:



 # Evaluate function

```
function compute_output(input, layer)
  r ← [] // empty vector
  foreach neuron in layers.neurons
     r ← r ⊕ dotproduct (neuron.weights, input)
  end foreach
  return activation(r)
end function
```

The code is similar (but we use the dotproduct). The operation ⊕ means adding something to the vector (the equivalent of += from python).

Feed forward step

This means that the entire process (evaluation and training) looks like the following:

Line	Process
1	`function evaluate_and_train(trainingSet, layers)`
2	`  repeat`
2	`    foreach entry in trainingSet`
3	`      output ← evaluate (entry.input, layers)`
4	`      if output != entry.label then`
5	`          // adjust the weight of all layers`
6	`      end if`
7	`    end foreach`
8	`  until exit_condition`
9	`end function`

Feed forward step

This means that the entire process (evaluation and training) looks like the following:

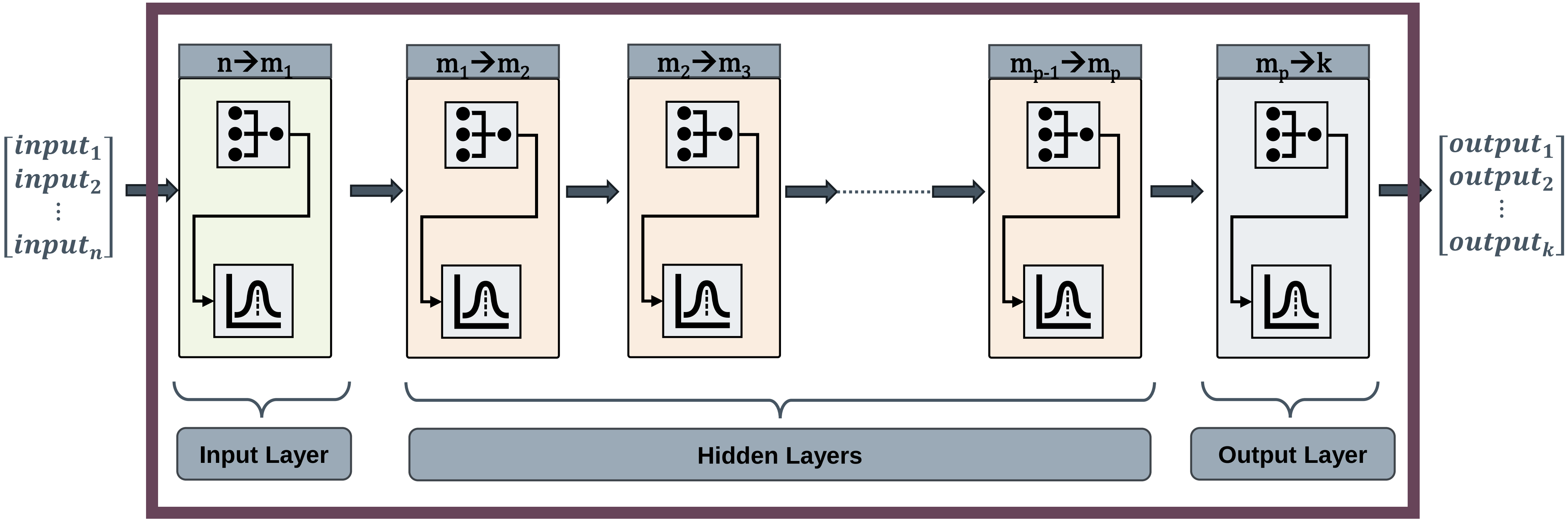
Line	Process
1	`function evaluate_and_train(trainingSet, layers)`
2	`repeat`
2	`foreach entry in trainingSet`
3	`output ← evaluate (entry.input, layers)`
4	`if output != entry.label then`
5	`// adjust the weight of all layers`
6	`end if`
7	`end foreach`
8	
9	e

The next step is to determine how we can achieve this !

Cost function

Cost function

Let's consider this generic representation of a neuronal network.



To simplify this representation, let's consider that this neuronal network is in fact a function – and lets call that function NN

Cost function

This means that we can represent a neuronal network in a mathematical way as follows:

$$NN(input) = output, NN: R^n \rightarrow R^k, with\ input = \begin{bmatrix} input_1 \\ input_2 \\ \vdots \\ input_n \end{bmatrix} and\ output = \begin{bmatrix} output_1 \\ output_2 \\ \vdots \\ output_k \end{bmatrix}$$

Now, lets consider that our training set looks like this:

Entry	$Input_1$	$Input_2$	…	$Input_n$	$Output_1$	$Output_2$	…	$Output_k$
#1								
#2								
…								
#m								

Cost function

This means that we can represent a neuronal network in a mathematical way as follows:

This is the input value associated with entry **#2** from the training set.

We can refer to is as: $\boldsymbol{input_2} = \begin{bmatrix} \mathbf{0.1} \\ \mathbf{0.5} \\ \dots \\ \mathbf{-1.2} \end{bmatrix}$

This is the output value associated with entry **#2** from the training set. We can think about the output value as the label (in many cases, the output value might be one scalar instead of multiple ones).

We can refer to is as: $\boldsymbol{output_2} = \begin{bmatrix} \mathbf{1.5} \\ \mathbf{2} \\ \dots \\ \mathbf{0.01} \end{bmatrix}$

Entry	$Input_1$	$Input_2$	...	$Input_n$	$Output_1$	$Output_2$	...	$Output_k$
#1								
#2	0.1	0.5		-1.2	1.5	2		0.01
...								
#m								

Cost function

For the entry #2 from the training set, we will obtain the following equation:

$$NN\left(\begin{bmatrix} 0.1 \\ 0.5 \\ \vdots \\ -1.2 \end{bmatrix}\right) = \boxed{\begin{bmatrix} output_1 \\ output_2 \\ \vdots \\ output_k \end{bmatrix}}$$

This is also called *predicted value*

Cost function

For the entry #2 from the training set, we will obtain the following equation:

$$NN\left(\begin{bmatrix} 0.1 \\ 0.5 \\ \vdots \\ -1.2 \end{bmatrix}\right) = \begin{bmatrix} output_1 \\ output_2 \\ \vdots \\ output_k \end{bmatrix}$$

Ideally, we would like the result of ***NN*** function to be:

$$NN\left(\begin{bmatrix} 0.1 \\ 0.5 \\ \vdots \\ -1.2 \end{bmatrix}\right) = \begin{bmatrix} 1.5 \\ 2 \\ \vdots \\ 0.01 \end{bmatrix}$$

Entry	$Input_1$	$Input_2$	…	$Input_n$	$Output_1$	$Output_2$	…	$Output_k$
#1								
#2	**0.1**	**0.5**		**-1.2**	**1.5**	**2**		**0.01**
…								
#m								

Cost function

For the entry #2 from the training set, we will obtain the following equation:

$$NN\left(\begin{bmatrix}0.1\\0.5\\\vdots\\-1.2\end{bmatrix}\right)=\boxed{\begin{bmatrix}output_1\\output_2\\\vdots\\output_k\end{bmatrix}}$$

Ideally, we would like the result o

This actually translates that we need to find a way for the output vector (the prediction) to be closer than what we expect.

$$NN\left(\begin{bmatrix}0.1\\0.5\\\vdots\\-1.2\end{bmatrix}\right)=\boxed{\begin{bmatrix}1.5\\2\\\vdots\\0.01\end{bmatrix}}$$

Cost function

Let's write the ***NN*** function a little bit different:

$$NN\left(\begin{bmatrix} input_1 \\ input_2 \\ \vdots \\ input_n \end{bmatrix}\right) = \begin{bmatrix} prediction_1 \\ prediction_2 \\ \vdots \\ prediction_k \end{bmatrix}, or\ NN(\,input) = prediction$$

and we need to compare this to $\text{expected} = \begin{bmatrix} expected_1 \\ expected_2 \\ \vdots \\ expected_k \end{bmatrix}$

Cost function

This can be done using the Mean Squared Error function:

$$MSE = \frac{1}{k}\sum_{i=1}^{k}(prediction_i - expected_i)^2$$

Observations:

- The closest the prediction vector is to the expected one, the closest the MSE will be to 0
- If we can consider MSE a cost function, then **training** a neuronal network implies **minimizing** the costs (that in turn translates in making the prediction (output) be closest to the expected values

Cost function

Mean Squared Error has some advantages:

- Continuous and differentiable
- Resembles the variance formula from regression (we can look at this formula as a way to evaluate how far the input vector is from a hyperplane defined by the network).
- In practice, the MSE formula uses 1/2k instean of 1/k to allow derivation:

$$MSE = \frac{1}{2 \times k} \sum_{i=1}^{k} (prediction_i - expected_i)^2$$

Q & A