Thibault Chatel

Challenge – ZAMA

First Draft

The first step was to choose a structure for the **matrix** we will manipulate. I decided to do so using a struct with 3 fields :

- two for the dimensions : `nb_col` and `nb_row`
- one for the data of the matrix itself, stored flattened in a single `Vec`, row-wise : `data`.

Like that, accessing the [i,j] element of the underlying matrix can be done accessing `data[i*nb_col+j]`.
This choice was motivated by the fact that this representation is efficient because with contiguous data like this, we benefit from **cache** mechanism during row traversal, which is oftenly performed within the operators.

Once the structure was decided, I added a method to create an instance of it, `new`. To create the following matrix
1 2 3
4 5 6
in our code, we supply the dimensions and a borrowed slice from which we copy the data :

```
let matrix = Matrix::new(3,2,&vec![1,2,3,4,5,6]);
```

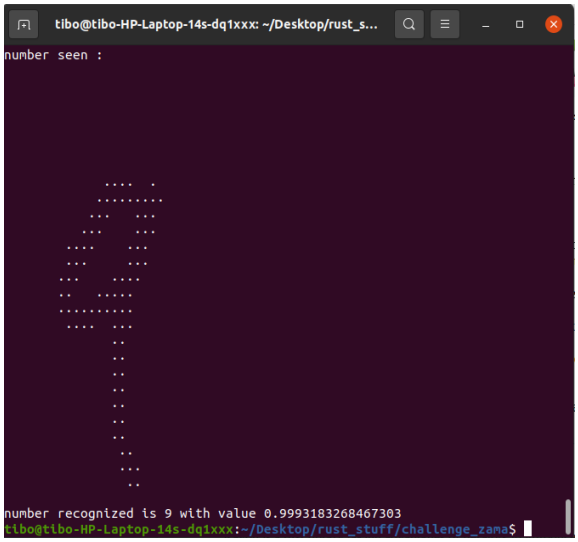
After that I was ready to implement the operators. I made the choice to translate the convolution and linear combination functions described in the provided document as a method on `&mut self`.
Writing the example, it felt more intuitive expressed that way.
For each written operator, I took the time to write a test associated to it in order to make sure that it was correct.

Some details for the `convolution` and `linear_combination` implementation :

- `convolution` : the first thing we do is we set the dimension fields of self. After that we modify its data to make it correspond to the result of the convolution operation : we go through each position of the matrix, and for each we compute the value at this location, equal to the sum of the product between the kernel coefficients and the sub matrix (relative to this specific location) coefficients.
- `linear_combination` : the first thing we do is we set the dimension field (only nb_row as we consider a method for input of one dimension only) of self. After that we modify its data to make it correspond to the result of the linear combination operation : we go through each position of the matrix and for each we compute the value at this location, equal to sum of the product between the corresponding row of weights (contiguous in memory given our memory representation of the matrix so really efficient!) and data of self + corresponding bias coefficient.

I chose to use a lot of iterators within my code because they make the code more readable and they come without a runtime performance penalty.
For the operators : `tanh`, `relu`, `softmax` I used the `iter_mut` method to directly apply function on data and avoid copies.

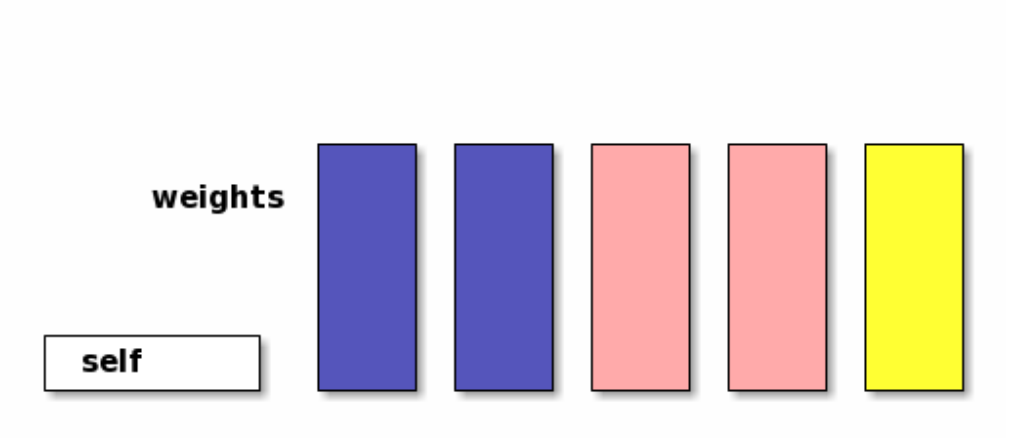
Once this was done, I was ready to use the provided parameters and see if the represented numbers were well recognized, following the indicated steps. After figuring out that I had to transpose the provided weights in order to use them as they should, I obtained the expected results !



Amelioration

We can easily introduce **multithreading** in our code. **Rayon** is a rust library taking advantages of iterators to introduce parallelism into existing code. The main idea behind it is to slice an iterator into sub iterators, which, being independent from each other given the applied operations, can be treated in parallel and so efficiently split the workload. The library relies on **work stealing** policy : each thread of the threadpool gets assigned a queue of tasks to be performed and a thread having an empty queue will feed itself by stealing from another thread's queue. Doing so we ensure a good work distribution among the threads and the job of feeding the waiting threads is done by these waiting ones. The whole library is based on a function called `join` which takes 2 closures as input. One is put inside the queue of the thread calling `join` and the other is directly performed by this same thread. When this closure is completed, whether the other one has been stolen by another thread or our thread picks it from the queue and processes it. From this primitive, Rayon offers `ParallelIterators` which divide their inputs into smaller chunks which are processed in parallel using `join`.

Considering our implementation of linear combination, what it does basically is going through each column of the *weights* matrix and perform the dot product between this column and *self* calling the method (1D given the assumption of the provided document). The columns are independent from each other. It is so an excellent candidate for multithreading, **assigning sets of columns to each of the thread** and then gather the results and this is exactly what Rayon offers !



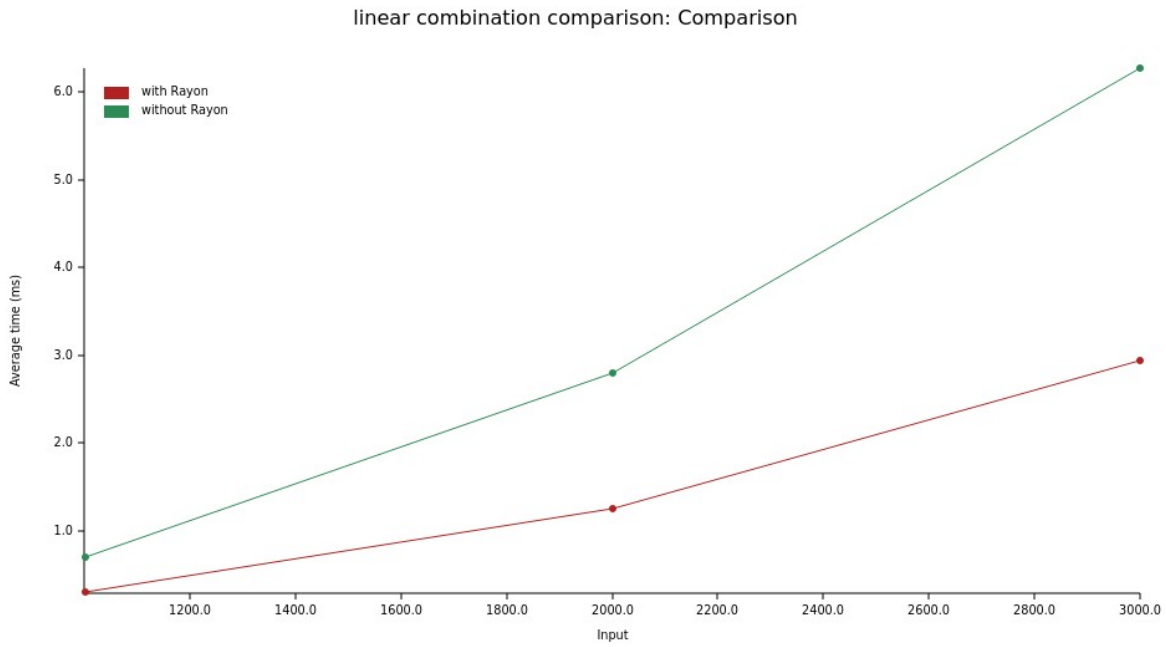
Division of the weights matrix, each color is linked to one thread

We just have to had the `into_par_iter` method to get our columns splitted between the threads of the threadpool.

We can be interested in the performance obtained doing so as threads have an overhead at creation for example, they can sometimes be more harmful than helpful. To compare the two versions we can use **Criterion**, a **benchmarking** library which also produces detailed charts to understand code's performance behavior.
I used my laptop to run it, which has the following configuration :

- *memory* : 7.5 GB
- *processor* : IntelCore i3-1005G1 (2 cores, 4 threads, 4MB cache)

An example of the charts produced, comparing the linear combination function with and without Rayon :



An example of the charts produced, comparing the linear combination function with and without Rayon for different input size

We see that Rayon speeds up more as the size of the entry grows.

The **tanh**, **relu** and **softmax** functions are also perfect candidates for Rayon, and written using iterators as we did makes it simple to turn it parallel.

I used **Travis CI** for **continuous integration**. I added a .travis.yml in my rust_stuff repository and it checks the build, clippy and test cargo's command for the rust_stuff/challenge_zama repository. The result of continuous integration is displayed in the README of this sub repository.