Clément Monari

Final Project

1) Description of the project:

This project is about the prediction of the winner of a 5 players against 5 players game. The name of this game is League of Legends. It's a 5v5 video game where every player has to choose a champion and use it to win the game with his allies. There are for the moment 162 different champions and a game lasts around 30 minutes. Each player has a rank in the game, which betrays how good a player is at the game, like the ranking for the tennis players for instance. There is also one server per region of the world, like Europe West, Europe East, North America, Asia etc….Here I focused on the server Europe West. My objective was to predict which team is going to win, knowing the actual ranking of the 10 players, and their Champions. For my predictions, I choose Tensorflow to create and evaluate my model. This was supposed to be a good library for my purpose, and moreover there is a pretty big community on this, which can be very usefull in case of need..

2) Creation of the database:

In order to create my database, I used the raw data that is on the server of the owning company of the game, Riot Games. To collect those data, I had to create an RIOT GAME Developer account in order to have an API Key. With this key, it was possible for me to send requests to the server. I created a few functions to be able to extract from the raw data the features of interest that I wanted. I needed to change the actual system of ranking with divisions and League points by a number between [0, 1]. I also changed the name of each champion by a boolean vector of size (162,) with a 1 at the alphabetic position of the name of the champion and 0 elsewhere. To create the vector of each team, I simply merged the 5 vectors of each champion of the team. One important thing is that there can only be one time the same champion in each game, so each team will only have a vector of champions composed of 0 and 1, never more than 1. The size of an example that we will put into our model will be (2, 167). For instance:

```
[[0.7528571428571429, 0.6421428571428571, 0.7407142857142858,
0.7016666666666667, 0.7230952380952381, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
[0.7019047619047619, 0.6754761904761905, 0.6983333333333334,
0.7083333333333334, 0.7111904761904762, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

The two vectors inside correspond to each of the 2 teams. Inside of a vector corresponding to a team, the 5 first float values correspond to the normalized ranking of the players. The other part of the vector corresponds to the champions chosen by this team (where there is a 1). For each of these game vectors, there is a number in output that says which team won. It is either a 0 if the first team won or a 1 if it was the second team.

During the creation of my database, I faced a problem. Indeed, there is a limit to the number of requests that you can send to the server within a time period. I can't launch more than 20 requests every 1 second and no more than 100 requests every 2 minutes. This limit seems pretty high but in fact it is quite low because I need to send more than 15 request to get all the info's for 1 game. Due to this limitation, I couldn't create a huge database in order to train a deep learning algorithm, so I decided to focus on a simpler model. I worked with a database of 3000 examples.

3) implementations of the models:

Concerning the models that I implemented, first, in order to have a point of comparaison for all my future results, I considered the simplest possible model (which I called the "trivial model"): the model that predicts always the same result, and this result is the mean of all the output of the training data. Then I tried a linear model, with one perceptron and with identity as activation function. Then I added a sigmoid activation function to this perceptron. After, I added a hidden layer with one perceptron, 2 perceptrons and then 3. I splitted my database randomly in 80% for training and 20% for test, and I used :

- Binary Accuracy as accuracy function
- Binary Cross Entropy and Mean Squared Error as loss functions. This problem can indeed be considered either as a classification problem or as a modelisation problem

For each complexity of the model, I initialized the weight at random and I did several initializations depending on the complexity of the model. The more the model has parameters, the more initializations I did. The goal of doing this was to avoid the model from always going in a local minimum, being well known that there are much more chances to fall in a local minimum if the model is complex, compared to a more simple model. I chose to do 500 iterations because it was a good number in order to stabilize the loss and the accuracy.

For each of these cases, I measured the loss and the accuracy at the end of the training for the training data and the test data, and I got the result plotted in the following figures.

4) differences between the models:

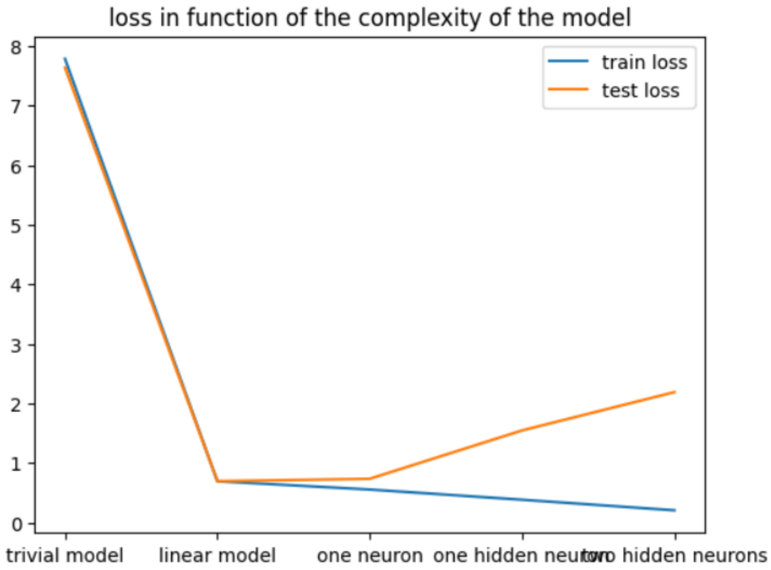

Figure 1: training loss = binary cross entropy

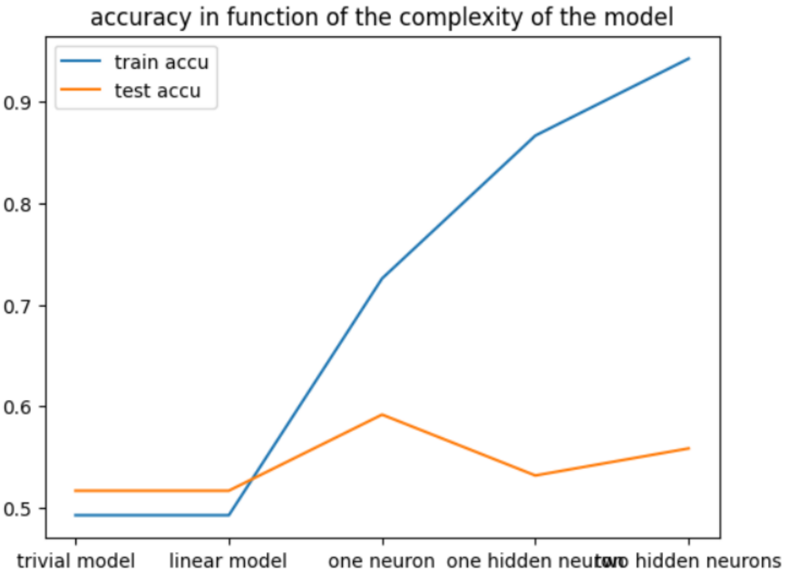

Figure 2 : training loss = binary cross entropy

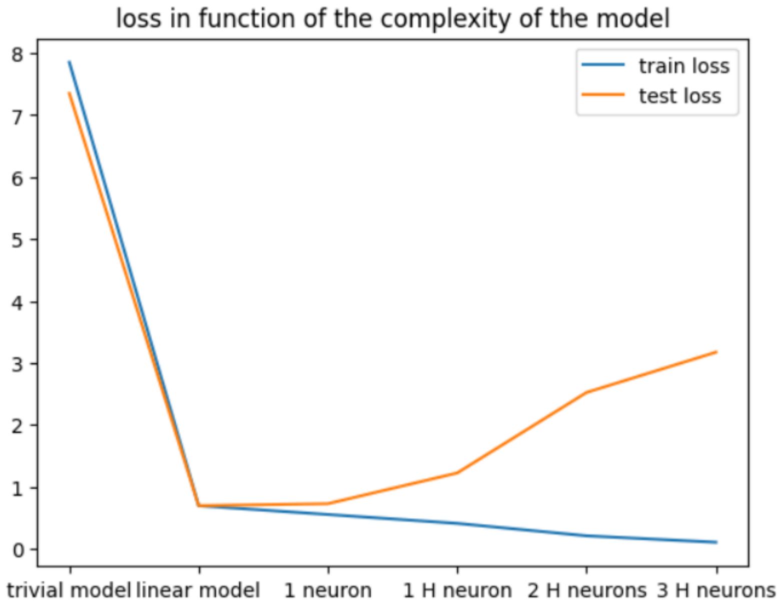

Figure 3 : training loss = Mean Squared Error

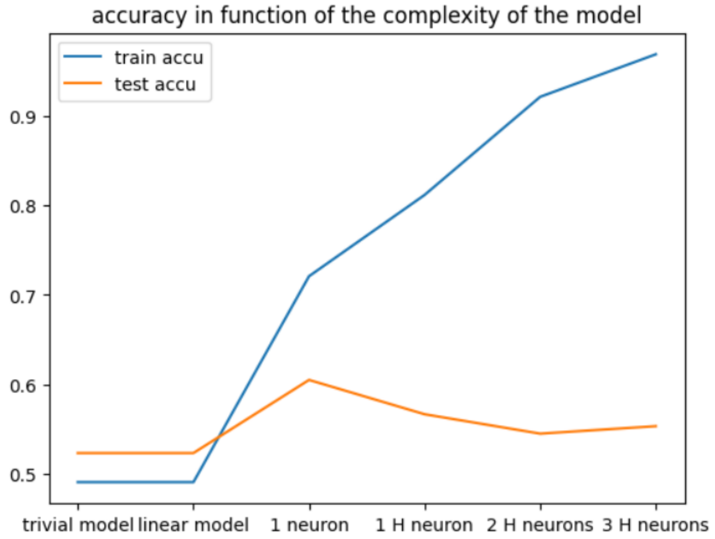

Figure 4 : training loss = Mean Squared Error

5) **Results and possible upgrades:**

Regarding the result that I got from this project: First, between all the models that I tried, the best one in terms of generalization is the model with one perceptron with a sigmoid function at the end. This model gives the best accuracy by far but its loss is really close to the one of the linear model. Second, the Mean Squared Error gives the best result compared to the binary accuracy, but the two results are really close. Finally, the most important is for me that the best results that I ever got on the test base was an accuracy of 61 % : this may look a poor precision, but, for me, it only means that the result of such a complex game cannot be simply deduced by the team composition : luck, concentration of the players and

other contingencies may always change the result of a game. And this cannot be better modelized by any entry or model.
Maybe an increase of the training set or new entry parameters would have slightly increase this performance, but I'm really not sure of it…

6) How to run the code:

About the way to run the code, there is a Jupyter notebook with all the functions and the code to create a dataset, to create the models and to use them to make predictions. There is also the main database used to train the models and the better model that I was able to run in the zip file. In the notebook, every section has comments so it is quite easy to see what each part corresponds to. There is also a public link to an Hugging face interface to see the final result and to make predictions based on the best model I was able to train: ***https://clement13430-riot-game.hf.space/***