Presenting UnLESS: Unsupervised Learning for Electoral Systems' Studying
a framework to search for optimal strategies using Deep Q-Learning with NetLogo and Keras

Michele Ciruzzi

Abstract

The goal of this paper is to describe UnLESS, a framework we have developed to study how different electoral systems influence the behaviour of parties. In order to minimize the *a priori* hypothesis on the parties' behaviours, we have chosen an agent-based approach, using reinforcement learning (particularly Deep Q-Learning) to find strategies that maximize votes received. The experimental setting was developed using Keras, to manage neural networks and learning, and NetLogo, to simulate elections and parties behaviour, linking them with Python.

1 Introduction

Electoral systems are an important topic in several fields of study: they were obviously a core topic in political science and in collective decisions theories in economics, but also mathematicians and sociologists have been interested in the topic. Each discipline has given a different contribute to study how electoral systems are able to represent people's ideas and guarantee stability in governments. Works like those by Arrow (1950) and Balinski and Young (2001) have been fundamental to set a bound to the legislators' expectations, thanks to their focus on how much accurate an electoral system could be in faithfully representing electors votes and preferences and their use of mathematical theory. On the other hand, scholars like Duvenger (1951) or Sartori (2004) have classified real elections to infer properties of electoral systems, working on empirical examples rather than on mathematical definitions. This two are the most common approaches in scientific literature and both have their flaws: both approaches don't investigate how electors and parties can adapt their behaviours to different electoral systems and the classifying one is restricted to real world situations, and so it is not able to study hypothetical cases. For this reason we have choose to pursue a different approach using an agent-based simulative model, which requires us neither to use real-world data (even if they can be a useful benchmark) nor to limit ourselves to problems with a simple enough mathematical

form, thanks to the use of unsupervised learning techniques instead of classical optimization theory. The proposed framework also aims to resolve some problems encountered by Bissey, Carini, and Ortona (2004). In particular, our model allows multidimensional distribution of electors and adaptive behaviour of parties, and eventually electors, focuses our attention less on what happens after the election, in terms of contractual power of parties, stability of the government and representativeness of electors, and more on what happens before, namely the strategies that parties and electors can adopt to reach the desired result.

2 Model

The model aims to explore the behaviour of parties under different electoral systems and electors' distributions. To do so we define two types of agents (parties and electors) which interact in an abstract metric space (the opinions' space). The parties' goal is to maximize the votes (or the seats) received by moving in the space (i.e. by changing their policies). Electors vote for the closest party, with a probability that is inversely proportional to the their distance, or abstain. Parties move in the space randomly or following the output of a neural network, which is trained to predict the votes received after a given move. We alternate simulations, to collect data, and neural network's trainings on these data. The general functioning of the model is sketched in algorithm 1, while more details about each step are given in the next sections.

2.1 Opinions' space

Classically political parties have been represented as a totally ordered set[1], whose extremes are the extreme Right and the extreme Left. In other words, the political positions of the parties has been represented on a line (i.e. a 1-dimensional space). However, such modelization is unable to distinguishing some important differences: for example far-right (neo-fascist parties) and far-left (URSS-inspired parties) have shown the same adversity toward democracy and civil rights and Brexit supporters were equally distributed both in the Labour party and in the Conservative party. For this reason, the model is generalized in n dimensions to account for different issues, each one with two extremes and an intermediate indifferent position. We could use any number of issue, as long as they are independent, i.e. each extreme of an issue is non contradictory (even if never realized in history) which each extreme of other issues. For example the 2 issues proposed by the Political Compass project[2] (an "economics" issue, whose extremes are deregulated free market and planned

[1] An order on a set is called total if for each couple of elements we can assign an order. In our case this means that for each couple of parties we can say which is more leftist and which is more rightist.

[2] The homepage of the project is online at `https://www.politicalcompass.org/` and an explanation of the methodology can be found at `https://www.politicalcompass.org/analysis2`

Algorithm 1: General functioning of the model

```
loop
    simulate
        update parameters
        if new electors then
            set electors position
        foreach party do
            with probability ϵ do
                foreach move do
                    NN computes quality of move
                parties perform the best move
            else
                parties move randomly
        electors votes
        save state
    train
        update parameters
        train the NN with the state of the last m simulations
```

economy, and a "social" issue, whose extremes are authoritarianism and libertarianism) satisfy our hypothesis. The property of independence is necessary to assume the set of the issues as the basis of a vectorial space. Particularly each issue can be represented with a $[-1, 1]$ interval and so the opinion space will be represented by $[-1, 1]^n$. Given a vector space we can define a distance (and so a norm) to obtain a metric space (and be able to measure the distance between two points, i.e. two political position). We have chosen the maximum metric to measure distance (i.e. $d(x, y) = \max_i |x_i - y_i|$) to emphasize that is easier to find and agreement if the opinions are similar on each issue rather than are equal on each issue but one, on which they are opposite. Figure 1 shows the difference between maximum metric and "standard" euclidean metric.

2.2 Electors

The first type of agents in our model is the electors. Electors are created at the beginning of the simulation with a random position in the space (i.e. random opinion). A good distribution for the electors' positions is the Beta distribution, because it is defined on a closed interval ($[0, 1]$) and very general (could be either symmetrical or not, and either unimodal, uniform or bimodal, see figure 2). So for an elector E we can state that his position on the ith issue is

$$x_i^E = 2(Beta(\alpha_i, \beta_i)) - 1$$

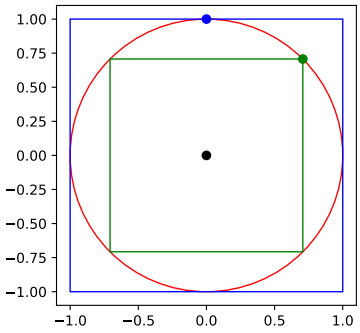


Figure 1: The squares represent the point at distance 1 (the blue one) and $1/\sqrt{2}$ (the green one) from the black point using the maximum metric. The red circle represents the points at distance 1 from the black point using the euclidean metric. We note that blue and green dots are at the same distance from the black dot using the euclidean metrics (that is they are indifferent from the black's point of view) while the green one is closer (i.e. preferable) using the maximum metric. The black point is the archetype of centrist, the blue one is an extremist (at least on one issue) while the green isn't. Using maximum metrics a big difference on one issue weights more than littler differences on many issues, which is (in the opinion of the authors) a more realistic preference criterion.

where $\{(\alpha_i, \beta_i)\}_i$ are the same for each elector (but not for each issue). At each time step of the simulation electors vote: each elector finds the closest party and votes it with a probability $\max(1-d, 0)$, where d is their distance, and abstains otherwise. In this version of the model electors don't change their opinion (i.e. don't move) and vote sincerely (i.e. don't cast strategic votes). Both features could be added in future versions of the model.

2.3 Parties

Parties are the main characters of this model. At the beginning of the simulation a certain number of them are created in a (uniformly) random position. At each time step, each party can move along one issues by a fixed step (along one issue at time in either direction) or remain at the same position. So the possible moves are $2n+1$, where n is the dimension of the opinions' space. With probability $1-\epsilon$ each party chooses a random move (with uniform probability among them). Otherwise it chooses the move with the best quality Q, which is proportional to the votes (or the seats) received after the move. The quality of a move is computed by a neural network (unique for all parties) which receives as input a proxy of the opinions' space, the position of the party and the move to be evaluated. The proxy could be the discretized distribution of electors, which is the discrete spatial empirical distribution of the electors obtained by dividing each dimension (i.e. a $[-1, 1]$ interval) in k equal subintervals and

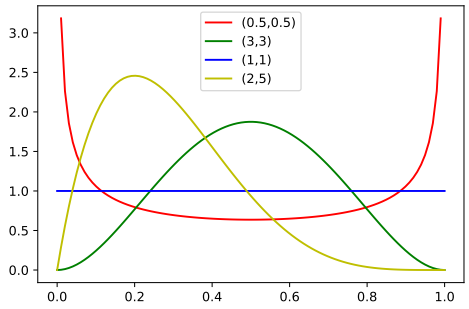


Figure 2: In this plot are represented the probability distribution for a Beta random variable for different parameters, reported in the legend as (α, β).

getting the number of electors in each n-dimensional subinterval, or a discretized distribution of votes, which is the spatial distribution of the parties weighted by the votes received discretized as above. The neural network's architecture is described in section 2.5. In this version of the model, parties don't sprout or die, while in future versions this could be taken into account. Particularly, parties could die when fail to get any seat for a certain amount of time, because of lack of funds and media exposure, and sprout randomly or when the abstained electors overtake a threshold.

2.4 Simulation

The simulations of the model are composed by an initial setup and a time step to iterate. During the initial setup the parameters of the simulation are set (see table 1) and parties and electors are created. Electors need to be initialized at the first iteration of the simulate-train loop and at each other iteration only if we are interested in learning on non constant opinion space, that is to say, if we want to verify if our learning model is able to generalize over different sets of electors (because they are a different realization of the same distribution or they are initialized from different distributions). Each time step is composed by four elements: first, parties can move as described in section 2.3, then electors are asked to vote as described in section 2.2, then seats are distributed according to the chosen electoral system and finally the current state of the simulation is saved. The current state includes a (discretize) electors' distribution, parties' positions, votes and seats in current and previous time steps (or equally at the beginning and the end of the time step) and the move chosen by each party. In this paper only first-past-the-post electoral system is used.

Parameter	Description
n	the number of dimensions of the opinions' space
$t_{\max}$	the number of time steps of the simulation
n_e	the number of electors at $t = 0$
$\{\alpha_i, \beta_i\}_{i=1\dots n}$	the parameters of the Beta distribution for the electors
n_p	the number of parties at $t = 0$
ϵ	the probability that a party act rationally (i.e. following the NN's predictions) each time step
Δx	the step of parties' moves
k	the number of the intervals used to discretize each dimension of the opinions' space
$I(\cdot)$	the input for the NN (particularly the chosen proxy for the opinions' space)
s	the total number of seats to be assigned
$S(\cdot)$	the chosen electoral system

Table 1: Parameters of the model's simulations

2.5 Neural network's architecture

The learning process is described in section 2.6 and it is based on the Deep Q-Learning algorithm (Mnih et al. 2015). The neural network is a multilayer feed-forward network. The network tries to evaluate the quality Q of each move $a_i \in A$ given the state of the system σ. More exactly, if $R(\sigma)$ represent the quantity to be maximized, which could be the votes or the seats received in the state σ, and $\sigma'(a_i)$ is the state after the move a_i has been performed, we have

$$Q(\sigma, a_i) = R(\sigma'(a_i)) + \gamma \max_{a_j \in A} Q(\sigma'(a_i), a_j)$$

where γ is a discount factor useful to keep into account also long-term rewards (in fact if $\gamma = 0 \to Q(\sigma, a_i) = R(\sigma'(a_i))$ while as γ grows also the state after the following moves is taking into account with increasing weight). So, it is evident that the last layer of the network must be composed by a single neuron with unbounded (i.e. ReLU or linear[3]) activation. The first layer instead will receive in input $2n + 1$ values for the moves (all 0 except one which is 1), k^n values to represent the opinion's space throughout the chosen proxy and n values for the position of the party who has to move, that is a $k^n + 3n + 1$ vector. Mnih et al. (2015) strongly recommend to use ReLU activation for hidden layers. We have arbitrarily used two hidden layers with slightly more neurons than the double of the input dimension.

[3]Linear activation maps $x \to x$ while ReLU (Rectified Linear Unit) activation maps $x \to \max(x, 0)$.

2.6 Learning

The goal of the learning process is to predict correctly the quality Q of each possible move, or in other words to minimize the difference between the predicted Q and the observed $\hat{R}$. We call this difference *Loss* and we can state

$$L = \mathbb{E}[(Q(\sigma, a_i) - \hat{R}(\sigma'(a_i)) - \gamma \max_{a_j \in A} Q(\sigma'(a_i), a_j))^2]$$

The data collected during simulation give as a tuple $(\sigma, a_i, \sigma'(a_i), \hat{R}(\sigma'(a_i)))$ on which we can perform the learning. The update of the network's weights has been done with the Adam algorithm, proposed in Kingma and Ba (2014), to avoid possible numerical problems which can appear using the Stochastic Gradient Descending method. In this version of the model the learning dataset is composed by all the states from the last m simulations, proposed to the learning algorithm for a certain number of epochs (repetition on the entire dataset) in random order. Future versions of the model could improve how the learning dataset is composed, selecting what simulations have to be kept in memory or using only a sample of the dataset, and how long train the network on the same dataset, setting a loss to be reached instead of a fixed number of epochs. Talking about loss and quality, it is important to remember that $Q \sim \frac{1}{1-\gamma} R$ (by the properties of geometric series) and that the absolute mean error has the same order of magnitude of $\sqrt{L}$. Even if a neural network is able to generalize, the predictions are way more accurate (and so is better the behaviour of the agents) for the states of the system which are similar to those have been included in the training dataset: an ϵ parameter too high in the first simulations leads the agents to learn to only follow some paths with the downside to ignore everything that is not along those paths and to act in fact randomly in unexplored areas (overfitting). For this reason the first simulations have to be explorative (i.e. with $\epsilon \leq 0.5$) and only after those ϵ can grow up to reinforce the agents' knowledge (this is called the exploitative phase).

3 Software architecture

In this section we will describe the implementation of the model that we have realized[4], following the flow chart in figure 3. The main piece of software is a Python script which loads a JSON file with all the instructions and parameters. Generally speaking, there are two types of instructions: those related to the simulation and those related to the learning. Those about simulation are executed by NetLogo thanks to the pyNetLogo library[5] (Jaxa-Rozen and Kwakkel 2018) which allows to load a NetLogo model and call its functions from Python. NetLogo is an optimal solution to quickly develop a model, but requires some

[4] All the software is available at `https://github.com/TnTo/UnLESS/`, free to use and reuse. We please you to cite the source in case. The version in the repo is the same used to obtain the result presented in section 4 and has some very experimental feature not showed here.

[5] At the time I write the extension is not compatible with NetLogo versions newer than the 6.0.4 one.

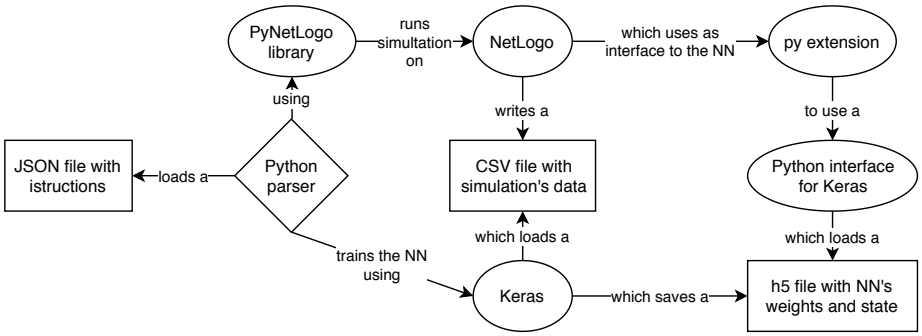


Figure 3: The flow chart above represents the interactions between the different pieces of software.

compromises to use at their best its built-in functions; for this reason the model has been realized only for $n = 2$, $k = 21$ and a $[-10.5, 10.5]$ space instead of the $[-1, 1]$ one. Future versions could replace NetLogo with Python either using an existing library like SLAPP[6] or building a simulator from scratch. NetLogo simulation needs to interface with the neural network which chooses the parties' moves: the bridge is realized with the Python Extension[7] which allows us to call a Python script to, using Keras, load the neural network and perform the computation needed. NetLogo saves the state of the system at each time step in a csv file, which is later loaded by Python and used to train the network. The other set of actions includes the creation and training of the neural network. All the operations on the network are performed using Keras (Chollet et al. 2015) with the TensorFlow back-end. NetLogo data are loaded and prepared using Pandas (McKinney 2010). Between a training and the next simulation it is important to save the neural networks weights in a file which can be loaded by the NetLogo Python Extension. For this purpose, Keras built-in function to save to h5 files has been used.

4 Results

To test our model and its implementation we have designed three different experiments[8]: a first one in which a single agent is trained on a constant electors' distribution, a second one in which the empirical distribution of the electors changes at each simulation but is extracted always from the same theoretical distribution and a third one in which two agents are trained only against the

[6] `https://github.com/terna/SLAPP`

[7] `https://github.com/NetLogo/Python-Extension`

[8] The experiments are in the *ex* directory of the repository named as *ex*+number of experiment+progressive letter+.*json*: at each letters is associate a different random seed. The ten versions of each experiment have been analysed together, to lower the probability that the results obtained are strongly influenced by the particular realization of the theoretical distribution of the electors or the initial positions of the parties during the simulations.

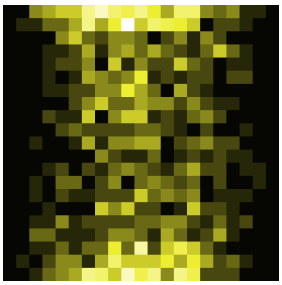


Figure 4: The figure above represents the density of the electors initialized with $\alpha_x = 3$, $\beta_x = 3$, $\alpha_y = 0.5$ e $\beta_y = 0.5$. This is a possible realization of the random variable, different initial seeds will generate a slightly different empirical distribution.

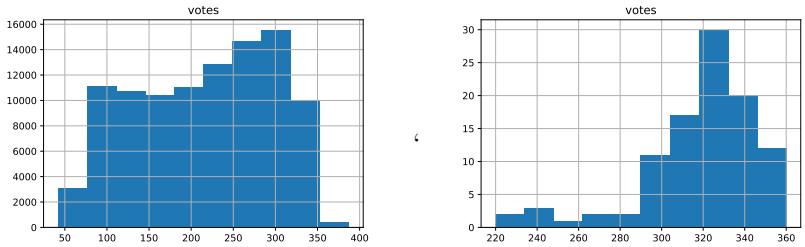
votes received by each one with a constant distribution of electors. The number of electors was fixed at 1000.

4.1 First Experiment

The goal of the first experiment is to check if the learning process works on a very simple case in which one party knows the distribution of the electors. We initialize electors only once at the beginning of the learning process with parameters $\alpha_x = 3$, $\beta_x = 3$, $\alpha_y = 0.5$ e $\beta_y = 0.5$. We have trained the neural network with a first long simulation with $\epsilon = 0$ (i.e. totally random), a set of 10 simulation with $\epsilon = 0.5$ and a set of 10 simulation with $\epsilon = 0.9$. During the set of simulations, we have trained the network not only with the data provided by the last simulation, but also with those from all the previous ones in the set (and the last of the previous set because of how we have implemented the memory management). We have repeated the experiment with 10 different random seeds and for each of these we have saved the position at the end of 10 simulations with $\epsilon = 1$. As input we have used the discretized electors distribution, that is a quasi-perfect knowledge of the electorate by the party. It is very evident by the data that the party learns which are optimal positions and how to reach them: at the end of the simulation the parties averages 318 votes (sd 26.3), while in a random run it averages only 215 votes (sd 80.8) (see figure 5). Furthermore, we observe that the distribution of the absolute position on the x axis is very near to 0 (mean 1.4, sd 0.9) while the absolute position on y has more variance (mean 3.9, sd 2.27) but is strongly unimodal with mode near the mean. Which suggests us that the optimal position could be near to $(0, \pm 4)$, which is also coherent with data reported in figure 6.

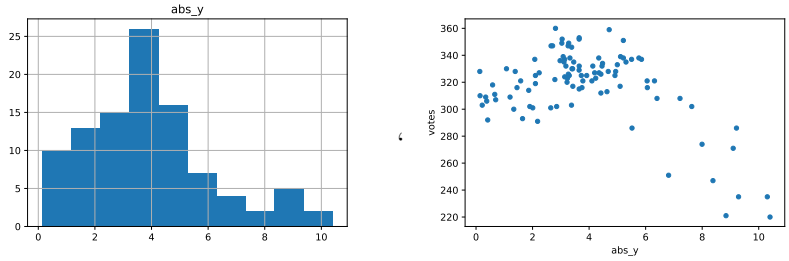
4.2 Second Experiment

During the second experiment we have tested if the learning process is able to generalize. The second experiment has been equal to the first one in all its features except that before each simulation we reinitialize electors. The results



(a) Random ($\epsilon = 0$) (b) After learning ($\epsilon = 1$)

Figure 5: Votes received by party during a random run (left) and after learning (right).



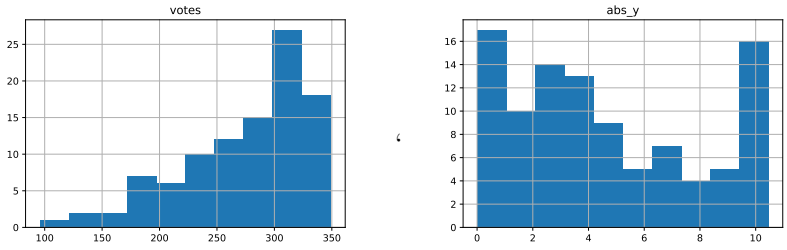
(a) Absolute y position (b) Votes against absolute y position

Figure 6: Absolute position on y axis and votes received by party plotted against the absolute position on y axis.

are worse than in the first case but are, however, comparable. In figure 7 are represented the distribution of votes and absolute positions along y axis at the end of the simulations (with $\epsilon = 1$) realized after the end of the learning. The effectiveness of learning process is still evident and it should not surprise us that an harder task (be able to generalize on similar thus non identical electors' distributions) would need longer, and better tuned, learning processes.

4.3 Third Experiment

With the third experiment, instead, we have tested the ability of two agents to adapt theirs each other's behaviour and to find the optimal position receiving as input only the votes received. To do this, we have initialized the electors only once and we have passed as input to the neural network the discretized distribution of votes. We have also used a longer number of simulations and trainings before collecting data for the analysis (20 with $\epsilon = 0.5$ and no memory, 20 with $\epsilon = 0.5$ and memory and 30 with $\epsilon = 0.9$ and no memory). The results are good but not optimal, probably because of a too simple learning algorithm particularly in memory management. The two parties together are able to catch up more votes than the single one of the first experiment (figure 8e) even if it



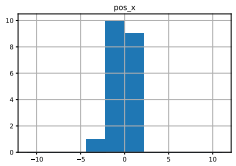
(a) Votes (b) Absolute y position

Figure 7: Votes received by party (left) and absolute position on y axis (right) at the end of the simulations, after learning.

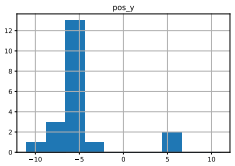
is no longer true if we consider each party individually (figure 8d), which is a pretty expected consequence of the competition. In figure 8f we see that the two parties are able to gain more votes when they occupied opposite positions on the y axis (i.e. when a left-right dynamic appears). The reason why the best output is not the more frequent one can have two possible explanations: the first one is that the learning was too short or too simple to allow parties to learn how to migrate to the other side if the other party is near to them; the second one is that when in the discretized representation of the opinion's space two parties are in the same position if $\epsilon = 1$ (like in the simulations that provide these data) the neural network suggests the same move to both parties until the end of the simulation. To avoid the last issue we should use $\epsilon < 1$ (like 0.95) which allows parties to separate from each other.

5 Conclusions

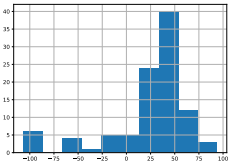
The deep learning approach to our problem seems really promising: this first version of the framework is already able to find a good solution also in a non trivial case with two agents and very little information. It will be necessary to improve the learning technique, particularly the management of the memory of previous simulations, to obtain stabler results in these first cases. To achieve this, it should be necessary to rewrite the simulator with Python to achieve a better integration with Keras and Pandas. As stated in other sections, this framework is also very extendible in future: with little efforts it is possible to introduce other actions for parties (like splitting or merging), complex behaviours for electors (like tactical voting) and other electoral systems. On the other hand, some interesting benchmarks can be found using game theory and probability theory: for example the optimal position of the first and the third experiment are the points in the opinions' space which maximize an integral function, a problem which can be solved with numerical techniques. All in all, this is a promising approach which is able to study how parties and electors adapt to electoral systems, giving a new point of view to this field of study.



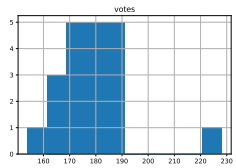
(a) Position along x axis



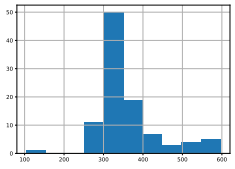
(b) Position along y axis



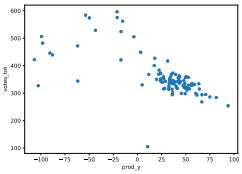
(c) Product of positions of the two parties along y axis



(d) Votes received by each party



(e) Sum of votes received by the two parties



(f) Sum of votes against product of y positions

Figure 8: Summary of third experiment: product of y position allows us to determinate if the two parties are on the same side (i.e. both $y > 0$ or both $y < 0$) or on opposite sides of the x axis.

References

Arrow, Kenneth J. (Aug. 1950).
"A Difficulty in the Concept of Social Welfare".
In: *Journal of Political Economy* 58.4, pp. 328–346. DOI: 10.1086/256963.

Balinski, Michel L. and H. Peyton Young (Aug. 23, 2001).
Fair Representation. Brookings Institution. 192 pp.

Bissey, Marie-Edith, Mauro Carini, and Guido Ortona (2004).
"ALEX3: A simulation program to compare electoral systems".
In: *Journal of Artificial Societies and Social Simulation* 7.3.

Chollet, François et al. (2015). *Keras*. https://keras.io.

Duvenger, Maurice (1951). *Les partis politiques*. A. Colin, Paris.

Jaxa-Rozen, Marc and Jan H. Kwakkel (2018).
"PyNetLogo: Linking NetLogo with Python".
In: *Journal of Artificial Societies and Social Simulation* 21.2.
DOI: 10.18564/jasss.3668.

Kingma, Diederik P. and Jimmy Ba (Dec. 22, 2014).
"Adam: A Method for Stochastic Optimization". In:
arXiv: http://arxiv.org/abs/1412.6980v9 [cs.LG].

McKinney, Wes (2010).
"Data Structures for Statistical Computing in Python".
In: *Proceedings of the 9th Python in Science Conference*.
Ed. by Stéfan van der Walt and Jarrod Millman, pp. 51–56.

Mnih, Volodymyr et al. (Feb. 2015).
"Human-level control through deep reinforcement learning".
In: *Nature* 518.7540, pp. 529–533. DOI: 10.1038/nature14236.
Sartori, Giovanni (2004).
Ingegneria costituzionale comparata. Strutture, incentivi ed esiti. Il Mulino.