Neural Networks and Reinforcement Learning Applications in Economics

Davide Mattioli

May 31, 2024

1 History and Functioning of Neural Networks

1.1 Il Perceptron

The notion of an artificial neural network as an electronic brain dates back to 1943, when Warren McCulloch and Walter Pitts created an electronic circuit to simulate the functioning of neurons and their connections.[1] The actual application of a neural network as software, however, came in 1958, when Fran Rosenblatt hypothesised that it was possible to create a neural network capable of organising input information through mathematical functions, so he proposed the idea of Perceptron, i.e. a neuronal 'network' consisting of a single neuron from Warren McCulloch and Walter Pitts' model with two layers, the input layer and the output layer.[2]

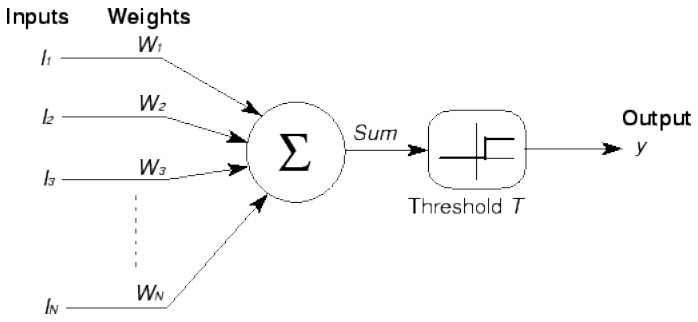

Figure 1: Neuron

The neuron performed a weighted sum of the inputs and included a function called 'step function' that returned the output in this way:

$$x = \sum_{i=1}^{N} I_i \cdot W_i + b_i$$

$$y = \begin{cases} 1 & \text{if } x > T \\ 0 & \text{if } x \leq T \end{cases}$$

Where I_i are the input values, W_i are the input weights, b is the bias value and y is the value returned by the neuron. The Perceptron was originally conceived not as a programme but as a custom-built machine, although its software was developed for the IBM 704 (and designed for image recognition). This combination proved successful, providing solid evidence that the algorithms and software could be transferred and used successfully on other similar computers. It should

be noted that Rosenblatt's main goal was not so much to build a computer capable of recognising and classifying images, but rather to gain insights into the workings of the human brain.

1.2 The limitations of the Perceptron

Despite the early successes with the Perceptron and consequently on research into artificial neural networks, there were many scholars who felt that these techniques had limited prospects. Notable among these were Marvin Minsky and Seymour Papert, whose 1969 book entitled "Perceptrons" was used to discredit research on artificial neural networks and focus attention on the alleged technical limitations of this field. One of the limitations pointed out most clearly by Minsky and Papert was that the Perceptron was unable to classify multidimensional input with non-linear patterns.

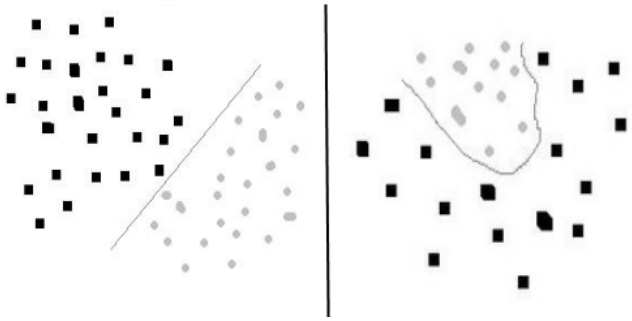

Figure 2: Left classification linearly separable, Right non-linearly separable.

The most striking example was the XOR problem, a classic problem in artificial neural network research. It involves using a neural network to predict the outputs of XOR logic gates given two binary inputs. An XOR function should return true if the two inputs are not equal and false if they are equal.

Input 1	Input 2	Output
1	0	True
1	1	False
0	1	True
0	0	False

At first sight, this might seem a simple problem, however, Minsky and Papert demonstrated that with the neural network architectures of the 1960s, this was not possible. This was because a single-layer perceptron used a linear activation function (or a step function, unit step function). This limited its use to single linear divisions of the input. This implied that the perceptron could only

classify inputs if they could be separated by a single straight line. The XOR problem required a non-linear division, so a single perceptron could not solve it effectively.[3]

The Perceptron's failure to handle non-linearly separable data did not imply an inherent flaw in the technology, but rather a matter of scale, in particular its two-layer structure did not allow it to solve more complex problems.[3]

progress was made in the 1970s when Seppo Linnainmaa theorised what was later called backpropagation, which made the creation of more complex networks possible.[4]

1.3 The Backpropagation

The first application came with the publication of an article by Rumelhart, Hinton and Williams, entitled "Learning Representations through Error Backpropagation," which allowed the parameters or weights of a neural network to be adjusted to reduce error.

In a multi-layer network, backpropagation follows the following steps:

1. Propagate the input data through the network, calculating the output of intermediate and final states.

2. Adjust the network weights to reduce the error with respect to the desired targets by calculating the gradient of the error with respect to the weights.

3. Adjust the weights to reduce the error.

4. Resume updating the weights until they converge in an attempt to cancel the error.

This process exploits the efficient application of Leibniz's chain rule and allows the gradient to be calculated, one layer at a time, by iterating backwards from the end towards the beginning.[5]

The term backpropagation refers specifically to the algorithm for calculating the gradient, but is often used more broadly to refer to the entire neural network training process. This approach contributed to the popularisation of backpropagation and initiated an active phase of research into multilayer neural networks.

Loss Function in Neural Networks

In the field of artificial neural networks, the loss function plays a fundamental role. This function measures the error between the predictions made by the neural network and the desired target values. The main objective is to minimise the error during the network training process.[6] Loss functions can take different forms, depending on the type of problem addressed:

Loss Quadratic

For regression problems, a common loss function is the loss quadratic, defined as:

$$\text{Loss}(y, \hat{y}) = (y - \hat{y})^2$$

Where y represents the target value and $\hat{y}$ represents the prediction of the neural network. This loss function measures the quadratic discrepancy between the prediction and the target.

Cross-Entropy Loss

For classification problems, the cross-entropy loss is widely used. Its form varies depending on the situation, but in general it measures the discrepancy between the probability distribution predicted by the network for the classes and the target probability distribution. This loss function significantly penalises incorrect predictions.

Hinge Loss

In the context of support vector machines (SVMs) and neural networks for binary classification, hinge loss is used to facilitate efficient separation between binary classes . It is defined as:

$$\text{Hinge Loss}(y, \hat{y}) = \max(0, 1 - y \cdot \hat{y})$$

Where y represents the target class (-1 or 1) and $\hat{y}$ is the network prediction.

Huber Loss

The Huber loss is a robust loss function used in regression problems that can be affected by outliers. It combines quadratic and linear loss so that it is less sensitive to extreme errors generated by outliers than pure quadratic loss.

1.4 Optimisers

Optimisers are algorithms or methods used to modify attributes of your neural network, such as weights and learning rate. Optimisation algorithms or strategies are responsible for reducing errors and thus providing the most accurate results possible.[7]

- **Gradient Descent**

  Gradient Descent is the most basic but most widely used optimisation algorithm, and is notably used in backpropagation, linear regression and classification algorithms. the Gradient Descent is a first-order optimisation algorithm because it represents the first-order derivative of a loss function. It calculates in which direction the weights should be changed so that the function can reach a minimum. Through backpropagation, the error is transferred from one layer to another and the weights of the model are modified according to these in order to minimise them.

$$\text{algorithm: } \theta = \theta - \alpha \cdot \nabla J(\theta)$$

- **Stochastic Gradient Descent**

  This is a variant of Gradient Descent, which attempts to update the model weights more frequently. In this case, the weights are changed after the calculation of the error on each training data set.

$$\theta = \theta - \alpha \cdot \nabla J(\theta; x(i); y(i))$$

  Since the model weights are frequently updated, they have a large variance, generating fluctuations in the loss function.

1.5 The activation functions

An activation function determines whether a neuron should be activated, this implies that it decides whether or not the input of the neuron is relevant in predicting the output.

The role of the function is therefore to obtain an output from a set of input values provided to a neuron, it therefore performs a filtering function throughout the neural network being inherent to each neuron.

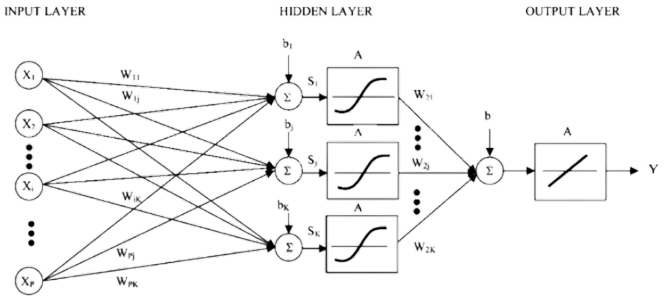

Figure 3: representation of activation functions in a neural network

In the biological neuron, elettrochemical signals are processed by the brain through a sequence of more or less powerful impulses. Similarly, this also occurs in learning in neural networks whose activation function is often called the Transfer Function in Artificial Neural Networks.[8]
The main role of the Activation Function is therefore to transform the input resulting from the weighed sum from the neuron into an output value to be fed to the next layer of neurons if one is present or as a final output. The development of activation functions has played a vital role in increasing the performance of deep neural networks, from a simple treshold function like those of the early Perceptron to complex and efficient functions like those of today.

Desirable Characteristics of an Activation Function

Desirable and necessary characteristics have been identified in the literature to consider an efficient activation function. These features are crucial to ensure that an activation function is suitable for use in deep neural networks and contributes to the successful training and overall performance of the model. [8]
Below, we will list some of the main features that an activation function should possess:

1. **The Absence of the Vanishing Gradient Problem** In neural networks, during the training process based on gradient descent, a problem known as Vanishing Gradient occurs. This occurs when the gradients of the weights of the initial levels become very small due to the repeated multiplication of values between 0 and 1 in the activation functions. This results in a significant reduction of the gradient value for the initial levels, preventing them from learning effectively. To address this problem, a desirable feature of an activation function is to prevent the gradient from tending to zero.

2. **Centred on Zero:**

   The output of the activation function should be symmetrical with respect to zero so that the gradients are not shifted in a particular direction. This symmetry is desirable to maintain training stability and network convergence.

3. **Computational efficiency:**

   Since activation functions are applied after each layer and must be calculated millions of times in deep neural networks, it is important that these functions are simple to calculate. This contributes to the overall efficiency of the network's training and inference process.

4. **Differentiation:**

   Since neural networks are trained using the Gradient Descent process, it is essential that the activation functions are differentiable or at least partially differentiable. This feature is essential so that the gradients can be calculated and used during back-propagation to update the weights.

Examples of activation functions

There are different types of activation functions, each with unique characteristics that make them suitable for specific tasks and different network architectures. The step-activation functions present in perception have already been discussed, and are generally considered extremely limited, as they can only activate or deactivate a neuron according to a fixed threshold. This makes it unsuitable for complex problems in which more than one activation gradient is required. Furthermore, its null gradient makes learning difficult and it is therefore rarely used in modern neural networks.

1.5.1 Linear Activation Function

The linear activation function is a function in which the activation is proportional to the input. This function makes no change to the weighted sum of the input, but simply returns the value supplied to it.

It can be represented as:

$$f(x) = x \tag{1}$$

Given its nature, it presents two main problems:

1. Backpropagation cannot be used, since the derivative of the function is a constant and has no relation to the input x.

2. All layers of the neural network will only collapse into one if a linear activation function is used, since the last layer will still be a linear function of the first . Consequently, a linear activation function essentially transforms the neural network into a single layer.

The linear activation function is only used in specific situations where a simple linear transformation is required and is not suitable for deep neural networks.

1.5.2 Non-Linear Activation Functions

Non-linear activation functions resolve the limitations of linear ones, since:

1. They allow backpropagation because the derived function being linked to the input, it is possible to trace and understand which neurons, through their weights, can provide a better prediction.
2. They allow the implementation of multiple layers of neurons since the output is now a non-linear combination of the input. In fact, the output can be represented as a combination of functional in the neural network.

Sigmoid or Logistics Activation Function

The Sigmoid activation function, also known as the logistic function, is widely adopted as it returns values between 0 and 1 and provides a smooth gradient, avoiding jumps in output values.

Mathematically, it can be represented as:

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

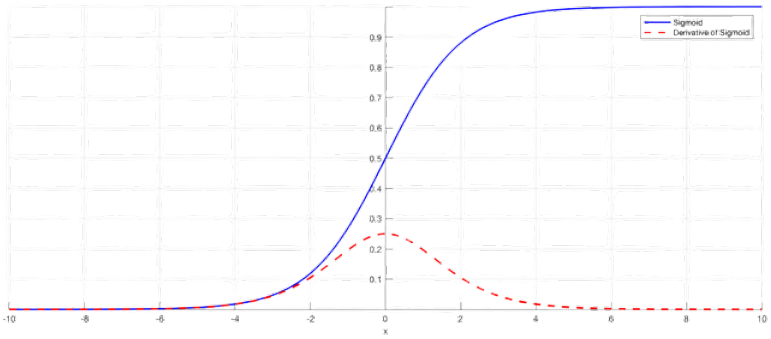

Figure 4: Sigmoid or logistic function and derivative

Derivative before the logistics function

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

As can be seen from the figure above, the gradient values are only significant in the range from -3 to 3. This implies that for values above 3 or below -3, the function will have very small gradients. This implies that when the value of the gradient approaches zero, the network stops learning and suffers from the "Vanishing Gradient" problem. Furthermore, the output of the logistic function is not symmetrical around zero, making the training of the neural network more difficult and unstable.

Hyperbolic Tangent Function (tanh)

The tanh function is very similar to the Sigmoid/logistic function, but with an output range between -1 and 1.

Mathematically, it can be represented as:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2}$$

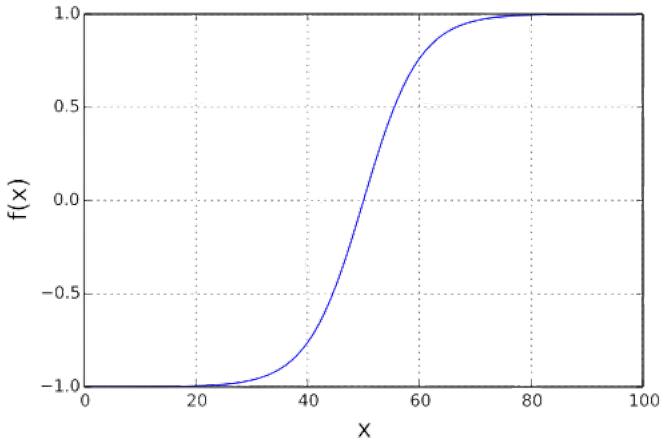

Figure 5: Hyperbolic Tangent Function

The advantages of using this activation function are:

- The output of the tanh trigger function is centred on zero, so output values can easily be mapped as negative, neutral or positive.
- It is usually used in the hidden layers of a neural network because its values lie between -1 and 1. Thus, the expected value turns out to be 0 or very close to it, allowing the data to be centred and simplifying learning.

The tanh function suffers from the problem of vanishing gradients like the sigmoidal function, but has a higher gradient.

ReLU Activation Function

The acronym ReLU stands for Rectified Linear Unit. Although it gives the impression of being a linear function, the ReLU has a derived function and therefore allows back propagation, and given its simplicity is computationally efficient. The main feature is that the ReLU function does not activate all the neurons at the same time; these will be deactivated if their output value is less than 0.

Mathematically, it can be represented as:

$$f(x) = \max(0, x)$$

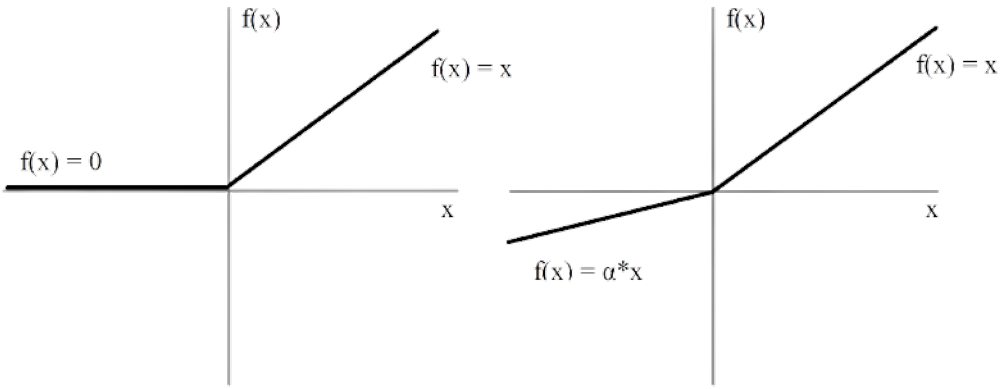

Figure 6: ReLu function on the left, leaky ReLU on the right

The advantages of using ReLU as a trigger function are:

- computational efficiency due to the fact that only a certain number of neurons are activated, which makes it much more efficient than the logistic function and tanh.
- rapid convergence to the global minimum, the ReLU in fact exhibits a rapid gradient descent to the global minimum of the loss function given its linear property.

The problem of dying ReLU

When x takes on negative values, the function has a gradient of zero. Because of this, during the backpropagation process, the weights and biases for some neurons are not updated, which can create inactive neurons that will never be activated. Furthermore, all negative input values immediately become zero, which decreases the model's ability to adapt or train correctly.

Leaky ReLU function

Leaky ReLU is an improved version of the ReLU function created to solve the problem of dying ReLU as it has a small positive slope when x takes on negative values.

Mathematically, it can be represented as:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ x \cdot \alpha & \text{if } x \leq 0 \end{cases} \text{ con } \alpha \neq 0$$

The advantages of the Leaky ReLU are the same as those of the ReLU, plus the fact that it allows backpropagation even for negative input values. Thus preventing inactive neurons.

1.6 Multilayer Neural Networks

An important step forward in neural network research was made with the use of multilayer neural networks. These solved the XOR problem and thus made it possible for neural networks to solve non-linear classification situations. The structure of a simple multilayer neural network is as follows:

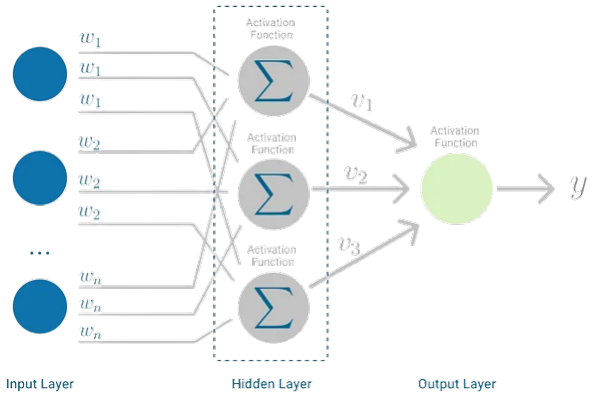

Figure 7: Mulitlayer neural network

The operation of a multilayer neural network can be described in terms of forward data propagation (forward pass) and error backpropagation (backpropagation). During forward pass, data flows through the network and each neuron calculates the linear combination of its inputs the result will be used in the activation function and used as input for subsequent neurons. This process is repeated through the layers until the output layer is reached.

Next, the error between the predicted output and the desired value is calculated. The error is then 'back-propagated' through the network, starting at

the output layer and going back, updating the connection weights based on the error using an optimisation algorithm.

This iterative learning cycle allows the network to adapt to the training data, gradually improving its representation capabilities and solving complex problems.

The depth of the network (the number of hidden layers) and the choice of activation functions influence the network's capabilities in modelling complex relationships in the data.Multi-layer neural networks have been shown to be particularly effective in solving problems that require non-linear solutions, such as the aforementioned XOR problem. [9]

The use of these networks is very varied due to their flexibility, they can have applications in many areas including:

- Data analysis: multilayer neural networks are commonly used in data analysis for tasks such as data visualisation, data compression and encryption. They are also used for predictive modelling, classification and clustering.
- Enterprise applications: multilayer neural networks are used in various enterprise applications such as data compression, streaming encoding, social media, music streaming and online video platforms. Vengono utilizzate anche per codificare i database, monitorare i dati di accesso e controllare la coerenza del database.
- Image and speech recognition: multilayer neural networks are used in image and speech recognition tasks such as object detection, face recognition and word-to-text conversion.
- Natural language processing: multilayer neural networks are used in natural language processing tasks such as sentiment analysis, language translation and text synthesis.
- Robotics: multilayer neural networks are used in robotics for tasks such as object recognition, motion planning and control.
- Learning: given their versatility, they are excellent artificial intelligences that can be used in reinforcement learning, creating agents capable of understanding and acting in a simulation.

2 Reinforcement Learning

Reinforcement Learning (RL), translated as 'Learning by Reinforcement,' is an area of machine learning that deals with how intelligent agents can take actions in an environment in order to maximise a reward function. RL is also one of the three main paradigms of machine learning, along with supervised learning, where both input and desired output are present in the data, and unsupervised learning, where the classes of the data are not known a priori but are learned automatically. Due to its generality, RL is studied in many disciplines, such as game theory, simulation-based optimisation and multi-agent systems.

2.1 Fuctioning of RL

In the context of agent-based Reinforcement Learning (RL) it is possible to create artificial intelligences called agents guided by a machine learning algorithm, in addition to the agent there is an environment which is defined by a succession of s states, which represent the characteristics of the environment at a given instant, the agent will then interact with it by performing a actions. the agent will use an algorithm which allows it to observe a s_t state from its environment at a t time. The agent will then interact with the environment by taking an action a_t in state s_t.

When the agent performs an action, the environment and the agent make the transition to a new state s_{t+1} based on the current state and the chosen action.

The state is therefore a sufficient statistic of the environment and includes all the information necessary for the agent to take the best action, which may also include information of the agent itself. In the literature on optimal control, states and actions are often referred to as x_t and u_t, respectively.

The best sequence of actions is determined by the rewards provided by the environment. Each time the environment makes the transition to a new state, it also provides a r_{t+1} scale reward to the agent for feedback.

The agent's goal is to learn a pi policy (a control strategy) that maximizes the expected return (the discounted cumulative reward). Given a state, a policy returns an action to be performed; an optimal policy is therefore any policy that maximizes the expected performance in a certain state of the environment.

In this context, the RL aims to solve the same problem as optimal control. However, the challenge in the RL is that the agent must learn the consequences of actions in the environment by trial and error. In addition, each interaction with the environment provides information, which the agent uses to update his knowledge.[10]

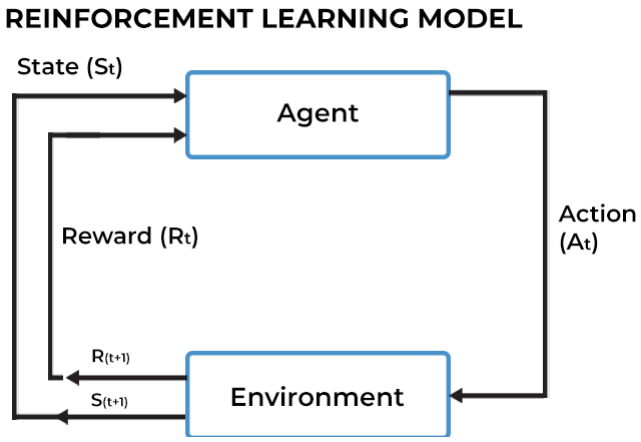

Figure 8: Reinforcement learning model

The environment is typically formulated in the form of a Markov decision-making process (MDP) that includes:

- A set of environmental and agent states S_t.
- a set of actions$A$ that the agent member may carry out
- A dynamic transition $T(s_{t+1}|s_t, a_t)$ to a state s_{t+1}, from a starting state s_t caused by an action performed at a time t, therefore the action a_t .
- A Reward Function $R(s_t, a_t, s_{t+1})$,a function that rewards the algorithm when performing an optimal action in the reference state.
- A Discount Factor $\gamma \in [0, 1]$, where lower values place greater emphasis on immediate rewards.

So a Reinforcement Learning (RL) algorithm has as its main objective the learning of an optimal or almost optimal policy by the agent. This policy aims to maximize a "reward function" or other reinforcement signal provided by the user, which accumulates through immediate rewards.

Policy e Rollout

In general, a policy (π) is to a probability distribution of the actions given the states:

$$\pi : S \rightarrow p(A = a|S)$$

If we consider an episodic Markov's (MDP) decision process, in which the state resets after each episode of T, then the sequence of states, actions, and rewards in an episode constitutes a "trajectory" or rollout of politics. Each rollout of a policy accumulates rewards from the environment, resulting in a return $R = \sum_{t=0}^{T-1} \gamma^t r_{t+1}$. The goal of the RL is therefore to find an optimal policy ($\pi*$) that maximizes the expected return of all possible states:

$$\pi^* = \arg\max_{\pi} E[R|\pi]$$

The Propety of Markov e POMDP

A key concept in RL is the property of Markov, which implies that only the current state affects the next state. However, this assumption may be unrealistic, since it requires states to be fully observable. A generalization of MDP is partially observable MDP (POMDP), where the agent receives an observation $o_t \in \Omega$. The distribution of observation

$$p(o_{t+1}|s_{t+1}, a_t)$$

depends on both the current state and the previous action. This is particularly relevant in control and signal processing contexts, where observation is described by a series of measurements/observations.

2.2 Q-Learning

Q-learning is an RL algorithm without model and without policy (off-policy) that aims to determine the best sequence of actions to be taken by an agent given its current status.
The agent experiences an action in a specific state and evaluates the consequences in terms of immediate reward or penalty received and the estimation of the value of the state in which it was brought. By experimenting with all actions in all states repeatedly, the agent learns which are the best on the whole, evaluated on the basis of the long-term reward. [11] Learning with the Q-learning method is a primitive form, but can form the basis for much more sophisticated learning models. Examples of the use of this methodology include several applications in industrial fields.[12]

2.2.1 Algorithm di Q-learning

The algorithm is defined by an array called Q containing all possible states and all possible actions for each state. Before learning begins, Q is initialized

with values called Q-values set arbitrarily (chosen by the programmer) generally randomly.

$$Q : S \times A \to R$$

Table 1: Q-table with States and Infinite Actions

State	**Q Values for Actions**			
	Action 1	**Action 2**	**Action 3**	**...**
State 1	$Q(s_1, a_1)$	$Q(s_1, a_2)$	$Q(s_1, a_3)$	...
State 2	$Q(s_2, a_1)$	$Q(s_2, a_2)$	$Q(s_2, a_3)$	...
State 3	$Q(s_3, a_1)$	$Q(s_3, a_2)$	$Q(s_3, a_3)$	...
State 4	$Q(s_4, a_1)$	$Q(s_4, a_2)$	$Q(s_4, a_3)$	...
State 5	$Q(s_5, a_1)$	$Q(s_5, a_2)$	$Q(s_5, a_3)$	...
State 6	$Q(s_6, a_1)$	$Q(s_6, a_2)$	$Q(s_6, a_3)$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Then, at every instant t, the agent selects a a_t, observes a $r_t$reward and enters a next state s_{t+1} (which can depend on both the previous state s_t and the selected action)and finally Q is updated giving more value to the shares that have generated a higher reward. The process of selecting an action action in a given state by:

$$\pi(s) = \arg\max_{a \in A} Q(s, a)$$

The values inside the table are updated according to the Bellman equation, which uses the weighted average of the current value and the new information obtained. The algorithm therefore has a function that calculates the quality of an action-state combination:[13]

$$Q_{\text{new}}(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot \left(r_t + \gamma \cdot \max_a Q(s_{t+1}, a) \right),$$

- α: The learning rate (learning rate) controls how much the new data will influence the current value.
- r_t: The reward earned instantly t.
- γ: The discount factor is a number between 0 and 1 ($0 \leq \gamma \leq 1$) and attaches greater importance to rewards received earlier than those received later, reflecting the value of a "good start".
- s_t:the current state.
- a_t:the selected action.
- s_{t+1}: The next new state.

Learning Rate

The learning rate, referred to as α, is a fundamental parameter in the Q-learning algorithm. This parameter adjusts the impact of new information on existing Q values. A learning rate of 0 indicates that the agent does not acquire any knowledge from the new experiences, continuing to rely solely on pre-existing information. In contrast, a learning rate of 1 implies that the agent attaches utmost importance to the latest information, completely ignoring previous information in order to explore new opportunities.

Discount Factor

The discount factor, represented as γ, is another key element in Q-learning, as it determines how much the future rewards should weigh compared to the immediate ones. A discount factor of 0 implies that the agent only considers the current rewards, completely ignoring future rewards. Instead, a discount factor close to 1 pushes the agent to maximize long-term rewards.

Initial Conditions (Q0)

Since Q-learning is an iterative algorithm, an initial condition is implicitly assumed before the first update. High initial values, known as "optimistic initial conditions," may encourage exploration. Regardless of the action selected, the update feature will cause lower Q values for the other alternatives, thus increasing the likelihood of choosing new actions.

Q-Learning and Neural Networks

An advanced version of Q-learning involves the use of artificial neural networks as function approximators. However, learning based on the Q function can result in the propagation of errors and instability when approximating the Q-function with an artificial neural network. In this case, you start with a lower discount factor and then gradually increase it towards its final value to accelerate the learning process.

2.3 Application of Q-learning in Price Theory

In the context of the Price Theory as the choice of an optimal price that maximizes the profit of an enterprise, it has long been discussed how this field is comparable to game theory, because it identifies the main characteristics of this field as the conflict between agents, where the subjects are in conflict with each other and must build a strategy based on the choices of others to win and the limited availability of resources in play .[14]

Table 2: Reward matrix for the price game

	Agent 2 low prices	Agent 2 high prices
Agent 1 low prices	$(3, 3)$	$(0, 5)$
Agent 1 high prices	$(5, 0)$	$(2, 2)$

Thus, it is possible to create a simulated environment where agents, businesses, operate in a theoretical reference market. One therefore has agents who are considered buyers allocating a budget for the purchase of a product offered by other agents who are considered sellers. It turns out to be possible to have an environment where a buying and selling process can take place between these. The process will then be repeated for a finite number of iterations where in each of these the buyers will purchase, if they can afford it, the product generated by the selling agents. While the sellers can create new products to put on the market to make a profit.
The entire code can be viewed at : https://github.com/mak8427/T1

2.3.1 Market Initialisation

The market on the demand side in this instance is defined by a gamma distribution described by the following formula:

$$f(x) = \begin{cases} \frac{1}{scale^{shape}\Gamma(shape)} x^{shape-1} e^{-\frac{x}{scale}} & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Where with:

$$Scale = 100$$
$$Shape = 6$$

It implies:

$$f(x) = \begin{cases} \frac{1}{100^6\Gamma(6)} x^5 e^{-\frac{x}{100}} & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Using this distribution, an array of 3,000 buyers is generated, who will have a budget available for the purchase of a single good, which will be used to satisfy the utility need.
Each product created will have, in addition to its selling price, a Utility value proportional to its production cost, described by a logarithmic function modified by the addition of a logarithmic function, This is necessary to incentivise the production of products with a higher price and thus giving a positive slope to the function:

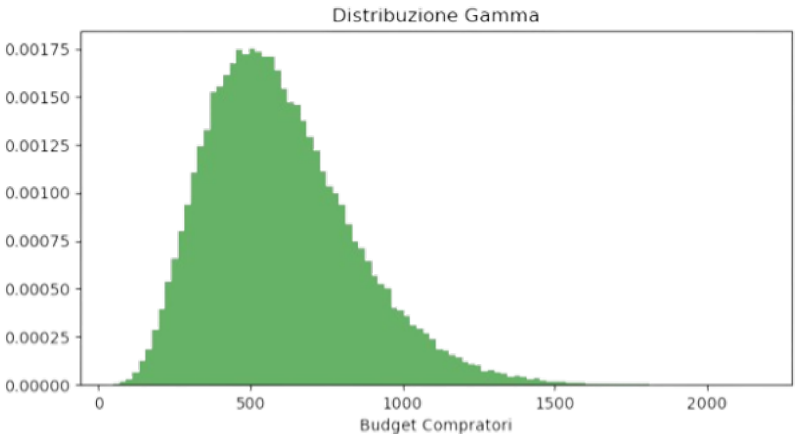

Figure 9: Gamma Distribution

$$U = \frac{1}{1 + e^{-w_x}} + \log_{10}(production\_price)$$

dove $w_x = -4.6 + 0.01 \cdot production\_price$, production_price is the attribute representing the production cost of the product. Utility will be one of the factors that buyers will consider when products are placed on the market, the goal of sellers will be to find an appropriate cost of production so as to satisfy the buyers' utility.
The optimal strategy, however, is not always to focus on achieving a higher utility than that provided by the products of other competitors, as such an approach would require an uncompetitive selling price and this would damage sales and consequently earnings.

In this type of simulation there will be two random agents who will choose the production and sales price following the budget distribution of the buyers, these will be the benchmark of the simulation, there will also be an intelligent agent who will learn following the Q-learning algorithm discussed above.

2.3.2 The Buyers

At each iteration, buyers will evaluate the products available on the market and purchase the product that meets the following conditions

- **Product Quality** The product with the most utility will be preferred over the others
- **Budget** The product must necessarily be within the buyer's budget, otherwise it will be discarded, the cheaper one will be chosen for equal utility

- **Affiliation** The buyer's previous purchases will also be considered in the decision-making process and an affiliation degree will be generated that will influence the purchase choice

Buyers will be initialised with each buying cycle.

2.3.3 The Sellers

Sellers at each iteration will be able to choose the following characteristics for the production of a product

- $production\_price \in [100, 2000]$ the production cost per product that will impact on its Utility
- $markup \in [10, 200]$ agents will be able to choose the net profit on the sale of each product
- $selling\_price = production\_price + markup$ it's the price that will be presented on the market
- $items\_produced \in [0, 2000]$ it's the number of products each agent will produce and place on the market
- $items\_sold <= items\_produced$ it represents the number of products sold on the market

Sales agents will have the following statistics used as an indicator of their performance.

- $profit = selling\_price \cdot items\_sold - items\_produced \cdot production\_price$ profits (losses) generated by product sales
- $comulative\_profit = \sum_{i=0}^{n} \text{profit}_i$ it's the sum of all profits (losses) of a simulation

2.3.4 The Simulation

at the beginning of the simulation the three agents are initialised, agent 1 and agent 2 will be the benchmark agents, who will choose their product values randomly following the budget distribution of the buyers. agent 3 on the other hand will be the agent who will use reinforcement learning to choose the price of products according to those of the other two agents.
First the Q-table of agent three is created with random gamma values $0 < \gamma < 1$ the structure of the Q-table will be as follows:

Where $p_x^{a_n}$ describes the price x of agent n. 40 possible prices have been chosen, these will be divided into fifty: $[50, 100, 150, ..., 2050]$ so the matrix will have the following form:

$$Q : 2000 \times 40 \to R$$

The Q function will be as follows:

$$Q_{\text{new}}(s_t, p_t) \leftarrow (1 - \alpha) \cdot Q(s_t, p_t) + \alpha \cdot \left(r_t + \gamma \cdot \max_a Q(s_{t+1}, p) \right),$$

Table 3: Q-table of infinite states and actions

combinations of price and agents 1 and 2	**Q Values for agent 3**			
	Price 1	**Price 2**	**Price 3**	**...**
$(p_1^{a_1}, p_1^{a_2})$	$Q(s_1, p_1)$	$Q(s_1, p_2)$	$Q(s_1, p_3)$	...
$(p_2^{a_1}, p_1^{a_2})$	$Q(s_2, p_1)$	$Q(s_2, p_2)$	$Q(s_2, p_3)$	...
$(p_3^{a_1}, p_1^{a_2})$	$Q(s_3, p_1)$	$Q(s_3, p_2)$	$Q(s_3, p_3)$	...
$(p_4^{a_1}, p_1^{a_2})$	$Q(s_4, p_1)$	$Q(s_4, p_2)$	$Q(s_4, p_3)$	...
$\vdots$	$Q(s_5, p_1)$	$Q(s_5, p_2)$	$Q(s_5, p_3)$	...
$(p_4 0^{a_1}, p_1^{a_2})$	$Q(s_{40}, p_1)$	$Q(s_{40}, p_2)$	$Q(s_{40}, p_3)$	...
$(p_1^{a_1}, p_2^{a_2})$	$Q(s_{41}, p_1)$	$Q(s_{41}, p_2)$	$Q(s_{41}, p_3)$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

- $\alpha = 0.95$: The learning rate will be chosen very high to favour exploration
- r_t: The reward obtained at instant t will be proportional to the agent's profit at the given instant .
- $\gamma = 1$: The discount factor will be 1 to give relevance to immediate gains
- s_t: The current state.
- p_t: The selected price.
- s_{t+1}: The next state which will not have much relevance as each state will be stand-alone apart from the degree of affiliation with consumers which will allow targeting a specific part of the distribution.

The simulation will then proceed as follows:

- Initialise Buyer and Seller Agents
- initialise Q-tables and Q-functions

The buying and selling cycle will then start and it will iterate for i cycles

- Product creation by sellers
- Buyer budget generation
- Each agent will choose if possible the product with the highest utility at a lower cost of the budget
- The profit (loss) for the selling agents will be calculated
- The Q-function will be updated

2.3.5 Agents Performance

The simulation was run for 10000 total cycles and the following data emerged: the unit of measurement of the price, cost and earnings variables will be expressed in euros.

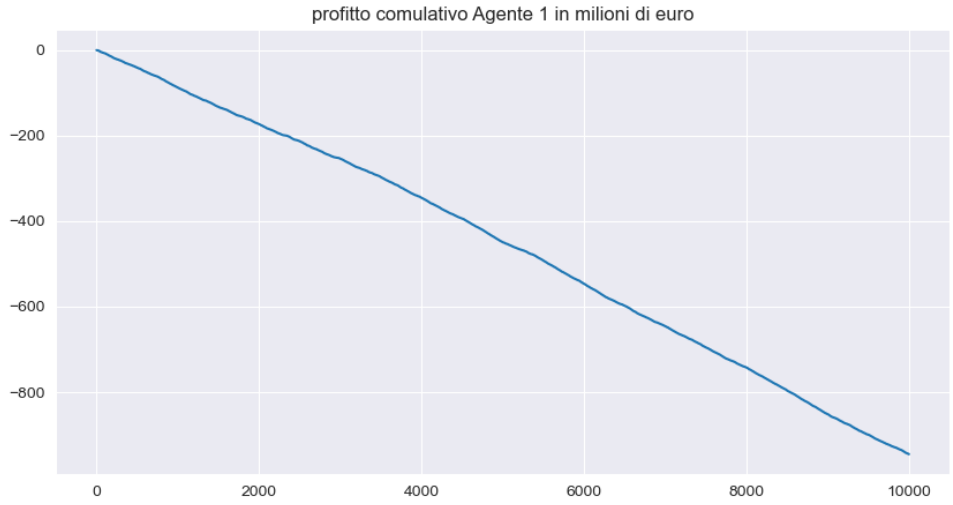

Figure 10: y-axis = accumulated profit (loss) from iteration 0 to iteration i , x-axis = iteration number

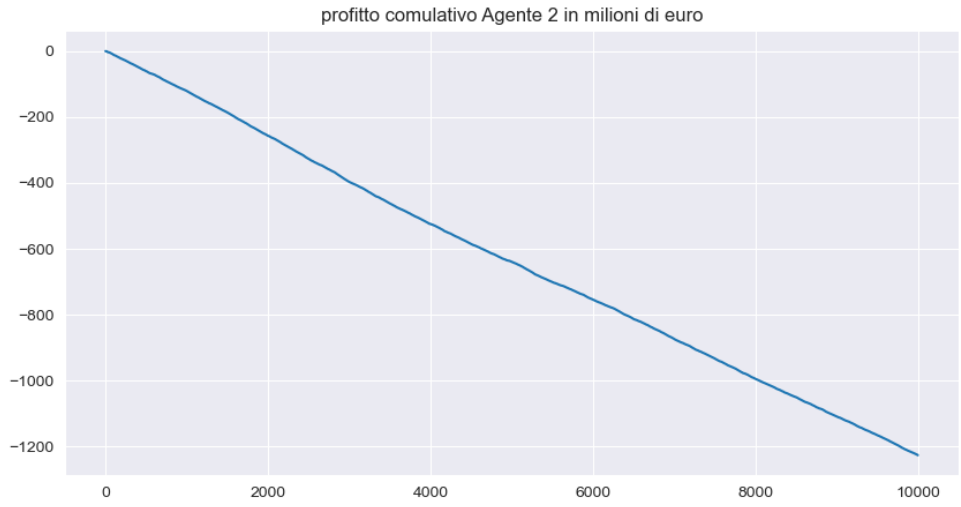

Figure 11: y-axis = accumulated profit (loss) from iteration 0 to iteration i , x-axis = iteration number

As can be seen from the graphs, the two random agents perform badly, showing a decreasing monotonic function, failing to accumulate any kind of profit during the iterations. The third agent on the other hand at the beginning of

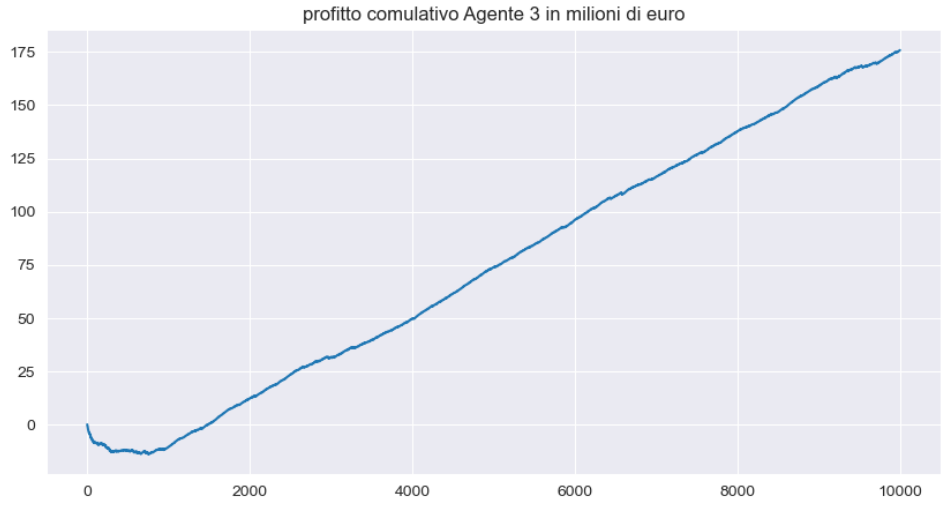

Figure 12: y-axis = accumulated profit (loss) from iteration 0 to iteration i , x-axis = iteration number

the cycles is at a loss, this is due to the fact that in the first cycles the Q-table has random values, this makes agent 3 comparable to agents 1 and 2. In the next part of the graph, from about iteration 1000 onwards, a positive trend can be seen which is maintained for all the other iterations, this occurs because the algorithm learns to choose the best price according to the one chosen by the other two agents in order to ensure a greater profit. This is confirmed by the following figure
The graph of the third agent as also confirmed by the previous graph starts at a loss and then remains positive throughout the rest of the simulation.

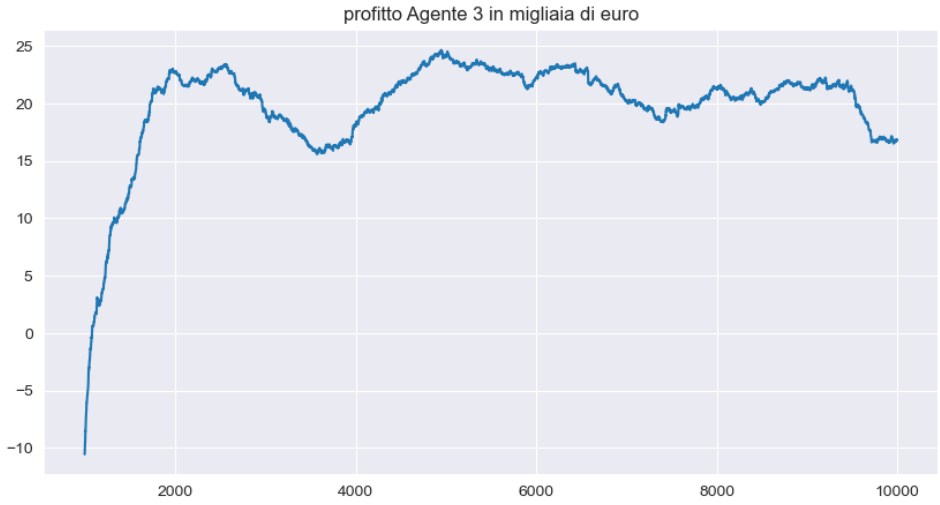

Figure 13: y-axis = profit(loss) per iteration , x-axis = iteration number

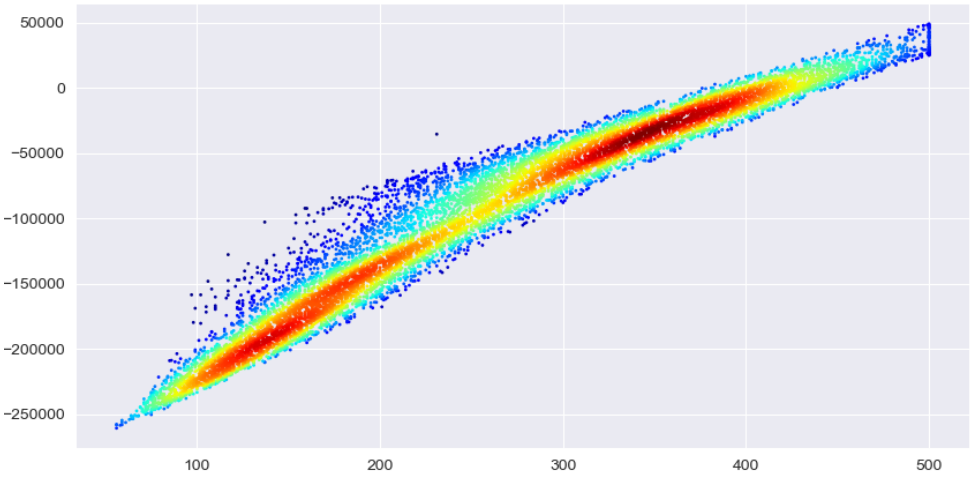

Figure 14: profit(loss) distribution with units sold y-axis = profit(loss) , x-axis = units sold

These two graphs show the distribution of sales with the relative earnings of the agents per iteration, as can be seen, agent 2 presents a more scattered distribution and is generally concentrated in the negative part of the profits. There is also an important correlation between profits and number of sales with a $R \approx 0.85$ this value however is variable and unique for each simulation. Agent

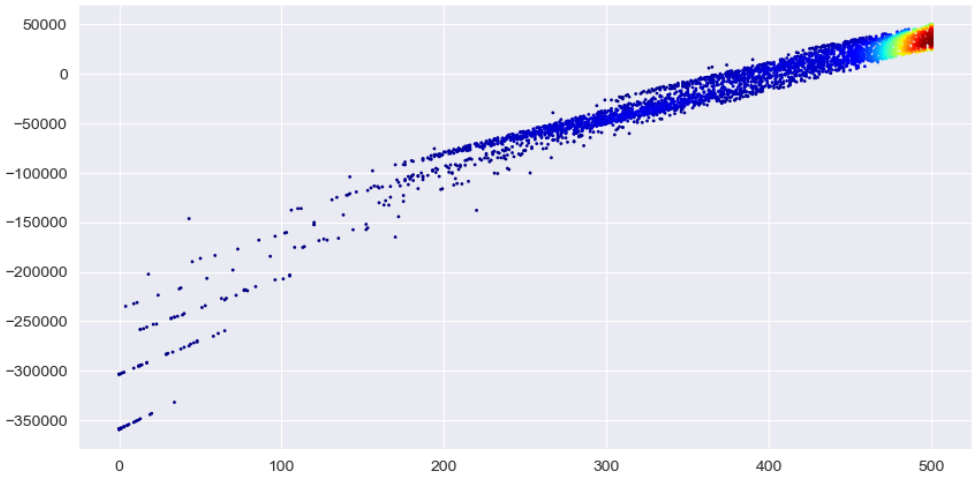

Figure 15: profit(loss) distribution with units sold y-axis = profit(loss) , x-axis = units sold

3 shows a high concentration in the positive part of the graph with almost all the units sold, in fact about 99.7 per cent of the data are in the positive part of the graph.

2.3.6 Conclusions

This simulation, although very crude and theoretical, shows how it is possible to create RL algorithms capable of competing in the decision-making process for choosing a product price that can be competitive in an environment with other competing agents. It could be expanded with the creation of multiple intelligent agents competing with each other, using adaptive neural networks that try to predict the budget distribution of buyers; the biggest problem lies in making agents unaware of the actions taken by others in a given state and trying to predict what their next move might be considering the factors in play that have already occurred previously. Research in the area of Deep RL applied to economics is relatively new and interesting developments in subsequent simulation models are to be expected.

3 Bibliography

References

[1] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[2] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[3] Zhao Yanling, Deng Bimin, and Wang Zhanrong. Analysis and study of perceptron to solve xor problem. In *The 2nd International Workshop on Autonomous Decentralized System, 2002.*, pages 168–173, 2002.

[4] Seppo Linnainmaa. *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors.* PhD thesis, Master's Thesis (in Finnish), Univ. Helsinki, 1970.

[5] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[6] Katarzyna Janocha and Wojciech Marian Czarnecki. On loss functions for deep neural networks in classification. *arXiv preprint arXiv:1702.05659*, 2017.

[7] Dami Choi, Christopher J Shallue, Zachary Nado, Jaehoon Lee, Chris J Maddison, and George E Dahl. On empirical comparisons of optimizers for deep learning. *arXiv preprint arXiv:1910.05446*, 2019.

[8] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *Towards Data Sci*, 6(12):310–316, 2017.

[9] Ken-Ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192, 1989.

[10] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

[11] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[12] Rui Nian, Jinfeng Liu, and Biao Huang. A review on reinforcement learning: Introduction and applications in industrial process control. *Computers & Chemical Engineering*, 139:106886, 2020.

[13] Deepshikha Pandey and Punit Pandey. Approximate q-learning: An introduction. In *2010 Second International Conference on Machine Learning and Computing*, pages 317–320, 2010.

[14] Robert A. Shumsky Praveen Kopalle. Game theory models of pricing. *Handbook of Pricing Management,*, 2010.