

UNIVERSITY OF SÃO PAULO
INSTITUTE OF MATHEMATICS AND STATISTICS
BACHELOR OF APPLIED MATHEMATICS

**Simulation of a self-parking car using
deep reinforcement learning**

André Shumhei Kato

FINAL ESSAY
MAP 2010 — CAPSTONE PROJECT

Supervisor: Prof. Dr. Alexandre Megiorin Roma

São Paulo
2022

*The content of this work is published under the CC BY 4.0 license
(Creative Commons Attribution 4.0 International License)*

*Esta seção é opcional e fica numa página separada;
ela pode ser usada para uma dedicatória ou epígrafe.*

Agradecimentos

Ninguém é tão sábio que não tenha algo para aprender e nem tão tolo que não tenha algo a ensinar.

— Blaise Pascal

Resumo

André Shumhei Kato. **Simulation of a self-parking car using deep reinforcement learning**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2022.

[illegible]

Palavras-chave: Palavra-chave1. Palavra-chave2. Palavra-chave3.

Abstract

André Shumhei Kato. **Simulation of a self-parking car using deep reinforcement learning**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2022.

[illegible]

Keywords: Keyword1. Keyword2. Keyword3.

Lista de Abreviaturas

ML	Machine Learning
RL	Reinforcement Learning
MDP	Markov Decision Process
PPO	Proximal Policy Optimization
TRPO	Trust Region Policy Optimization
PG	Policy Gradient
ANN	Artificial Neural Network
NN	Neural Network
MLP	Multilayer Perceptron
IME	Instituto de Matemática e Estatística
USP	Universidade de São Paulo

Contents

1	Preliminaries	1
1.1	Introduction	1
1.2	Motivation	1
2	Introduction	3
2.1	Reinforcement Learning	3
2.2	Elements of a Reinforcement Learning Problem	4
2.3	Agent-Environment Interface	5
2.4	Goals and Rewards	5
3	Markov Decision Processes	7
3.1	Markov Processes	7
3.2	Policies	8
3.3	Optimality	9
4	Neural Networks	11
4.1	Machine Learning	11
4.2	Neural Networks	12
4.3	Perceptron	13
4.4	Multilayer Perceptron	15
5	Policy Gradient	19
6	Trust Region Policy Optimization	21
7	Experiments	23

Appendixes

A	Código-fonte e pseudocódigo	25
----------	------------------------------------	-----------

Annexes

A Perguntas frequentes sobre o modelo	27
--	-----------

References	29
-------------------	-----------

Index	31
--------------	-----------

Chapter 1

Preliminaries

1.1 Introduction

Reinforcement Learning is considered a subfield of Machine Learning, where learning occurs through an agent interacting with an environment. At each time step, the agent performs an action and the environment responds by producing a reward signal and transitioning to the next state. The goal of the agent is to maximize the total expected reward. [SUTTON and BARTO, 2015](#)

There are a lot of challenges that naturally arise from reinforcement learning problems that differ from the ones faced in classical machine learning. For example, balancing immediate rewards and future rewards: up to which point is it worth to sacrifice early rewards in exchange for bigger rewards in the future? An agent might be inclined to take actions that have been taken before because it has the knowledge of how much reward those actions will yield. Thus, limiting the agent to that specific set of actions and ultimately impairing it from further exploring the environment and possibly discovering new states and actions that could yield even more rewards. This is called the exploration-exploitation dilemma. [SUTTON and BARTO, 2015](#)

The focus of this work will be an application of the Proximal Policy Optimization (PPO) algorithm to train an agent able to park a car in a designated spot.

1.2 Motivation

With car crashes being more and more common, car manufacturers have started working on technologies that help mitigate crashes, ranging from simple proximity sensors that alarms the driver if a collision is imminent or stops the car by itself to fully fledged auto-driving systems. In the latter, automated parking is a key part in autonomous vehicle systems that allows cars to navigate through a parking lot completely unassisted.

This work aims to recreate the self-parking system inside a 3D virtual environment using deep reinforcement learning and studying how the algorithm performs in different parking situations.

Chapter 2

Introduction

2.1 Reinforcement Learning

Reinforcement learning is learning what to do - mapping situations to actions in order to maximize the reward received. The learner (or "agent") has no knowledge on what actions should be taken, as in many forms of machine learning. Instead, it must discover which ones yield the most reward by trying them and observing what happens. When taking an action, that action may affect not only the immediate reward but also the next reward and all the subsequent others. These two characteristics - trial-and-error search and immediate vs. delayed rewards are the most distinguishing aspects of a reinforcement learning problem.

The following examples illustrate how reinforcement learning concepts are applied in real life:

1. A chess player making a move. The move is informed both by planning - anticipating possible replies and counterreplies - and by intuitive judgements of what positions and moves are desirable.
2. A gazelle calf struggling to stand on its feet after being born and a few hours later being able to run.

Both examples involve an interaction between a decision-making agent and the environment, in which the agent seeks to achieve a goal, despite uncertainty about its environment. The agent's action may affect the future state of the environment, for example, the next chess move will affect the possible options and opportunities in the future. Taking the correct choice requires taking into account indirect and delayed consequences of actions, and thus requires planning.

Both examples have goals in which the agent can judge the progress towards it based on what it can sense directly. For example, the chess player could judge his progress by comparing his remaining pieces with the opponent's. The player also knows whether he wins.

In order to fully formulate a reinforcement learning problem, we need optimal control

of a Markov Decision Process, which will be discussed later, but the basic elements are shown in the next section.

2.2 Elements of a Reinforcement Learning Problem

Apart from the agent and environment, there are other subelements of a reinforcement learning system: a *policy*, a *reward signal*, a *value function* and, optionally, a *model of the environment*.

A *policy* is a mapping from perceived states to actions to be taken when in those states, that is, a policy is what defines the agent's behavior. Policies can be deterministic, being a simple function or a lookup table, or stochastic, with probabilities associated with each action.

A *reward signal* is what defines the goal in a reinforcement learning problem. At each time step, the environment sends a reward signal, which is nothing but a number. The agent's goal is to maximize reward received over the long run. In biological systems, rewards are analogous to feeling pleasure or pain. The reward received depends on the agent's current action and on the state of the environment. The agent cannot alter this process in any way, but it can influence it through its actions. The reward signal is the primary basis for altering a policy. If an action selected by the policy yields a low reward, then the policy may be changed to select another action in that situation in the future.

A *value function* tells us how much reward the agent should expect to receive over the future by starting in a specific state. While reward signals indicates whether a state is immediately desirable or not, the value function estimates the long-term desirability of that state by taking into account the states that are likely to follow the current one. For example, a low reward state might be followed by a high reward state or vice versa. It's values we are most concerned with when making and evaluating decisions - we seek for actions that yield maximum value, not reward. Note that this is equivalent to maximizing reward over the long run. The major problem with value is that it can be hard to estimate - rewards are given directly by the environment, but values must be re-estimated at each iteration from sequences of observations an agent makes over its lifetime. The most important component of almost all reinforcement learning algorithms is value function estimation.

A *model of the environment* is something that mimics the behavior of the environment itself. More generally, that allows for inferences about how the environment will respond to certain actions. For example, for a given pair of state and action, the model might predict how the environment will respond to that action in that particular state. Models are particularly useful for *planning*, which is deciding on a course of action by considering possible future situations. Methods that use models and planning are called *model-based* methods, as opposed to *model-free* methods that rely on trial-and-error to learn about the environment.

2.3 Agent-Environment Interface

The learner and decision-maker agent and the thing it interacts with, comprising of everything outside the *agent*, is defined as the *environment*. The agent and the environment interact continually, with the agent choosing actions and the environment responding to those actions, presenting new situations to the agent and rewarding it.

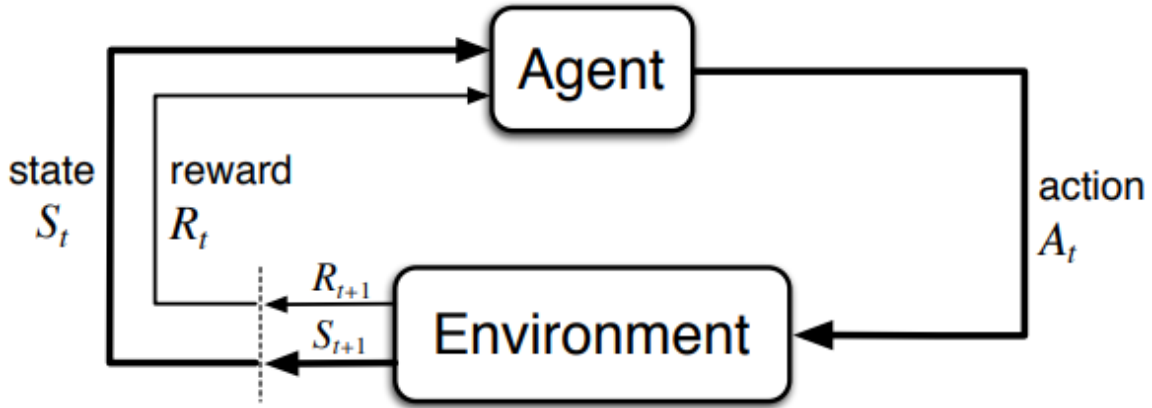


Figure 2.1: Representation of the agent-environment interface from *SUTTON and BARTO, 2015*.

In other words, at each time step $t = 0, 1, 2, \dots$, the agent receives some representation of the environment's state $S_t \in \mathcal{S}$, where \mathcal{S} is the set of all possible states, and selects an action $A_t \in \mathcal{A}(S_t)$, where \mathcal{A} is the set of all possible actions and $\mathcal{A}(S_t)$ the set of all possible actions in state S_t . At the next time step $t + 1$, the agent receives a reward $R_{t+1} \in \mathbb{R}$ and a new state S_{t+1} .

The action the agent takes is sampled from the agent's policy, denoted π , with $\pi(a | s)$ being the probability of taking action a when in state s . π is just an ordinary function defining a probability distribution over $a \in \mathcal{A}(s)$. Reinforcement learning methods specify how the policy is changed as the agent gathers more experience.

2.4 Goals and Rewards

The goal of an agent is formalized in terms of a reward signal $R_t \in \mathbb{R}$ that is passed to the agent by the environment. Consider the sequence of rewards received after time step t : R_{t+1}, R_{t+2}, \dots . We define a *return* G_t , which is a function of that sequence. The simplest case is the sum of all of them:

$$G_t = R_{t+1} + R_{t+2} + \dots + R_T$$

The agent's goal is to maximize the total amount of rewards it receives. In other words, the goal is to maximize the *expected return* $\mathbb{E}[G_t]$. In the above definition, we have the notion of a final time step, which comes very naturally when the agent-environment interaction can be broken down into subsequences, which we call *episodes* or *trials*. For example, plays of a game, trips through a maze or any sort of repeated interactions can be considered

episodes. Each episode ends in a special state called *terminal state*, followed by a reset to a standard state or to a sample of a distribution of starting states.

However, not all agent-environment interactions can be broken naturally into episodes, instead, it could go on without limit. That is, with slight abuse of notation, $T = \infty$ and the return G could easily be infinite as well. Thus, the concept of *discounting* arises, where we introduce a *discounting factor* γ , $0 \leq \gamma \leq 1$, to weigh immediate and future rewards:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Then, the sum is finite as long as $\gamma < 1$ and the reward sequence is bounded. Also note that if $\gamma = 0$, the agent is only concerned with immediate rewards and the action A_t will be chosen in such a way to maximize only R_{t+1} . If γ is close to 1, then the future rewards will be taken into account more strongly.

According to [SUTTON and BARTO, 2015](#), although formulating goals in terms of reward signals appears to be limiting, in practice, it has proved to be flexible and widely applicable. For example, to make a robot learn how to escape from a maze, the reward is often -1 for every time step that passes prior to escape; this encourages the agent to escape as quickly as possible.

While designing how rewards should be given to the agent, it's crucial that it's done in such a way maximizing also makes the agent achieve the goal. That is, the reward signal must not be used to impart any prior knowledge to the agent about how to achieve the goal. For example, in a chess game, rewards should be given when the agent wins the game, and not when accomplishing subgoals, such as taking enemy pieces. If the agent gets rewarded for achieving subgoals, it might find a way to maximize the rewards by only taking enemy pieces and without actually winning. All in all, the reward signal is our way of communicating to the agent what needs to be done, not how to do it.

Chapter 3

Markov Decision Processes

3.1 Markov Processes

In the reinforcement learning framework, the agent makes decisions as a function of a signal from the environment called the **state**. In this section, we discuss what is required of the state signal and what information it does or does not convey.

The state signal should include an immediate sensation, but it could include more than that, including some memory from past states. In fact, in typical applications, it usually is expected the state to inform the agent of more than just immediate sensations. For example, we could hear the word "yes", but we could be in totally different states depending on what was the question that came before and can no longer be heard. In contrast, the state signal should not be expected to inform the agent about all of the past experiences or all about the environment. Ideally, we want the state signal to summarize well past experiences, in such a way all the relevant information is retained.

To formalize this idea, for simplicity, suppose there are a finite number of states and rewards. Also, consider that the environment responds at time $t + 1$ to an action taken at time t . We define the history sequence as $h_t = \{S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\}$. Assume the state is a function of the history, that is, $S_t = f(h_t)$. If the state signal does not have the Markov property, the response of the environment depends on everything that happened earlier. Otherwise, the environment's response depends only on the state and actions at time t .

Definition 3.1: Markov Property

A state signal is said to have the Markov property if and only if

$$P\{R_{t+1} = r, S_{t+1} = s' \mid h_t\} = P\{R_{t+1} = r, S_{t+1} = s' \mid S_t, A_t\} \quad (3.1)$$

for all r, s' and h_t . If 3.1 is satisfied, then the environment also has the Markov property.

If every state of an environment is Markov, then we define a Markov Decision Process

(MDP) as follows:

Definition 3.2: Markov Decision Process

A *Markov Decision Process* is a tuple $\langle S, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

- S is a finite set of states
- \mathcal{A} is a finite set of actions
- \mathcal{P} is a state transition probability matrix
 $\mathcal{P}_{ss'}^a = P[S_{t+1} = s' \mid S_t = s, A_t = a]$
- \mathcal{R} is a reward function
 $\mathcal{R}_s^a = E[R_{t+1} \mid S_t = s, A_t = a]$
- γ is a discount factor such that $\gamma \in [0, 1]$

3.2 Policies

A big part of reinforcement learning is estimating how much reward the agent is expected to get by being in a specific state. In order to do that, we introduce the *value function* - a function of a state or a state-action pair that estimates the expected future rewards (or expected return), which tells us how good it is to be in a certain state or how good it is to take a specific action while in a specific state. Accordingly, value functions are defined with respect to different ways of acting and these ways of acting are dictated by a *policy*.

A policy is a mapping of actions to states, defining a probability distribution over $a \in \mathcal{A}$ for each $s \in S$.

Definition 3.3: Policy

A *policy* π is a probability distribution over actions given states,

$$\pi(a \mid s) = p[A_t = a \mid S_t = s] \quad (3.2)$$

A policy fully defines the behavior of an agent, and is stationary, that is, $A_t \sim \pi(\cdot \mid S_t)$, for all $t > 0$.

Given an MDP $\mathcal{M} = \langle S, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and a policy π , we define the value function:

Definition 3.4: State-Value Function

The *state-value function* $v_\pi(s)$ of an MDP is the expected return starting from state s and following policy π afterwards

$$v_\pi(s) = E_\pi[G_t \mid S_t = s] = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right] \quad (3.3)$$

Definition 3.5: Action-Value Function

The *action-value function* $q_\pi(s, a)$ of an MDP is the expected return starting from state s , taking action a and following policy π afterwards

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \quad (3.4)$$

The value function v_π can be estimated using *Monte Carlo Methods* by keeping the average of the returns that followed each state and then the average will eventually converge to the state's true value as the number of times that state is encountered approaches infinity. Similarly, q_π can be estimated by the same method by keeping the average of each state and each action taken in that state.

3.3 Optimality

Solving a reinforcement learning problem often means finding a policy that maximizes reward over the long run. Value functions define a partial ordering over policies:

$$\pi \geq \pi' \text{ if } v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s \in \mathcal{S}$$

Theorem 3.1: Policy Optimality

For any MDP, there exists an optimal policy π_* such that $\pi_* \geq \pi, \forall \pi$. All optimal policies achieve the optimal value function v_{π_*} and the optimal action-value function q_{π_*} .

The optimal state-value function is often denoted simply as v_* , and is defined as

$$v_*(s) = \max_{\pi} v_\pi(s)$$

for all $s \in \mathcal{S}$.

Similarly, the optimal action-value function is defined as

$$q_*(s, a) = \max_{\pi} q_\pi(s, a)$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$.

There are various methods to find such optimal functions, but in this work, we'll focus on a particular class of methods called *Policy Gradient Methods*. Prior to that, we'll introduce some theory on *Neural Networks*.

Chapter 4

Neural Networks

In this chapter, we'll briefly discuss some concepts on *Neural Networks* (or NNs) and a little bit of *Machine Learning*. NNs will later be used as a function approximator in the Policy Gradient chapter and was also a subject of study during the making of this work.

4.1 Machine Learning

Nowadays, as the Internet continues to expand, we have more and more information available and stored everyday. Due to that large volume of data, analyzing it and extracting meaningful information has become a task increasingly difficult for humans to perform. From that challenge, the concept of *Machine Learning* arises.

Machine learning can be defined as a set of techniques (or algorithms) that allows for a computer program to extract information, identify patterns and relationships in large amounts of data, in such a way a human cannot.

According to MITCHELL, 1997, "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T , as measured by P , improves with experience E ".

GOODFELLOW *et al.*, 2016 broke machine learning down to 3 main paradigms:

- In **Supervised Learning**, we have a dataset containing features, but each example is also associated with a *label* or *target*. For example, a dataset containing emails labeled as **spam** or **not spam**. A supervised learning algorithm can study this dataset and learn to classify whether an email is a spam or not.
- In **Unsupervised Learning**, we have a dataset containing many features, but no labels. Then, the goal is to learn useful properties of the structure of this dataset. In the context of deep learning, we usually want to learn the entire probability distribution that generated the dataset. Some other unsupervised learning algorithms perform other roles, like clustering, which consists of dividing the dataset into clusters of similar data.

- Then, there's **Reinforcement Learning**, where we don't have a fixed dataset. Instead, RL algorithms interact with the environment, forming a feedback loop between the learning system and what it has experienced.

4.2 Neural Networks

Neural Networks are a set of machine learning algorithms, whose structure is strongly related with the structure of the human brain.

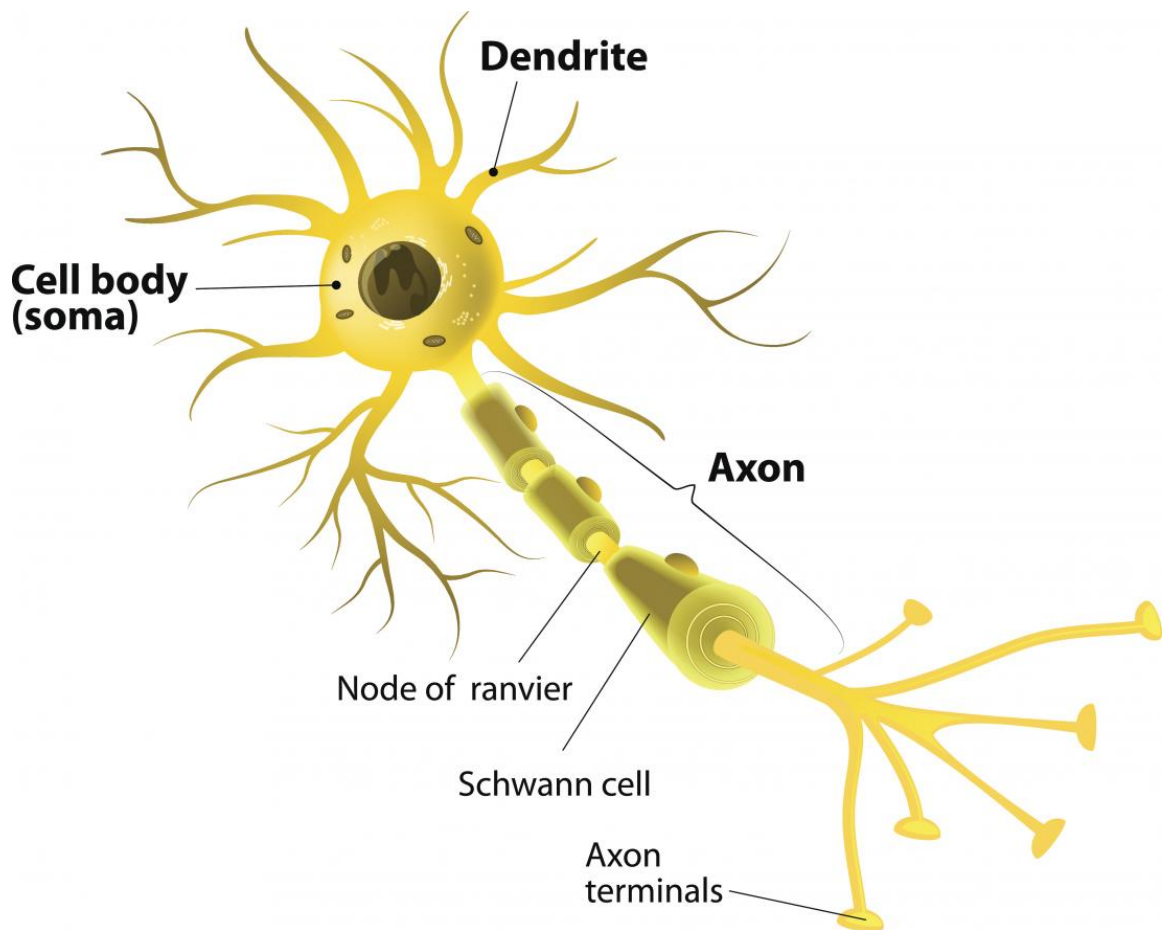


Figure 4.1: Source: *MedicalNewsToday*

The main structural cells responsible for processing information are divided into 3 main parts:

- **Dendrites:** filaments responsible for receiving and transporting stimulus coming from the environment or other cells of the body.
- **Body cell:** processes the information gathered by the dendrites.
- **Axon:** transmits the nervous impulses generated by the cell body to the other cells through the axon terminals.

In summary, upon receiving stimuli, the cell body processes those signals and if the result exceeds a certain threshold value, the neuron fires a nerve impulse indicating it reacted to the input signals, which are further transmitted by other neurons to other neurons or cells.

The human brain processes information processes in order 10^{-3} seconds, having a network of about 10 billion neurons densely connected, making it a huge, complex and efficient processing powerhouse, performing tasks like recognizing images and audios better than any machine. In an attempt to recreate a system that was able to operate like a human brain, the concept or *Artificial Neural Networks* was idealized.

HAYKIN, 2009 cites the following useful properties and capabilities:

- **Nonlinearity:** an artificial neuron can be linear or nonlinear. Therefore, a neural network, made up of an interconnection of nonlinear neurons, is itself nonlinear. Nonlinearity is a highly important property, particularly if the underlying physical mechanism responsible for generation of the input signal (e.g., speech signal) is inherently nonlinear.
- **I/O Mapping:** neural networks perform very well in supervised learning tasks, where we have labeled data. Each piece of data is used to update the synaptic bias of the network such that the difference between the expected output and the actual output is minimized.
- **Adaptivity:** neural networks have the capability to adapt their synaptic weights to changes in the surrounding environment.
- **Evidential Response:** in particular, a neural network trained to operate in a specific environment can be easily retrained to deal with minor changes in the operating environmental conditions.
- **Contextual Information:** in the context of pattern classification, a neural network can be designed to provide information not only about which particular pattern to select, but also about the confidence in the decision made. This latter information may be used to reject ambiguous patterns, should they arise, and thereby improve the classification performance of the network.
- **Uniformity of Analysis and Design:** neural networks enjoy universality as information processors, in a sense that the same notation is used in all domains involving the application of neural networks, making it possible to share techniques and theories between models with different purposes.
- **Neurobiological Analogy:** the design of a neural network is motivated by analogy with the brain, which is a living proof that parallel processing is not only physically possible, but also fast and powerful.

4.3 Perceptron

In order to create a model that would function similarly to a human brain, the psychologist and neurobiologist Frank Rosenblatt proposed the *perceptron* model. In his words, "a

probabilistic model for information storage and organization in the brain" [ROSENBLATT, 1958](#).

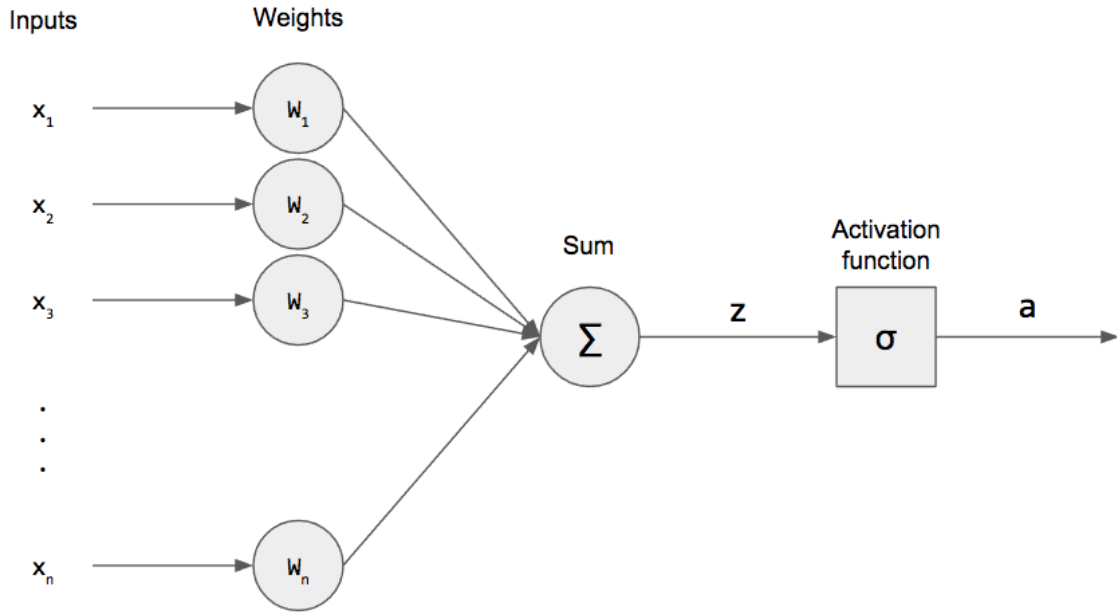


Figure 4.2: Source: [Medium](#)

In this artificial neuron, we have the following structures:

- An input vector $x \in \mathbb{R}^n$
- The synaptic weights $w \in \mathbb{R}^n$ responsible for weighing the values from the input vector. Large and positive values indicate higher relevance. Conversely, small and negative values indicate lower relevance.
- A bias $b \in \mathbb{R}$, which can be interpreted as how easily a neuron is activated.
- A linear combinator Σ weighing the input signals into a single scalar value:

$$z = \Sigma(x) = \sum_n^{i=1} w_i x_i + b$$

- An activation function $\phi : \mathbb{R} \rightarrow \mathbb{R}$ mapping the weighted sum z to an output $\phi(z)$.

In general, given $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$, the perceptron model can be defined as a function $h : \mathbb{R}^n \rightarrow \mathbb{R}$ such that

$$h(x \mid w, b) = \phi(w^T x + b)$$

From the perceptron, multiple models have been derived, but we are particularly interested in a specific *feed-forward* neural network, called *multilayer perceptron*.

4.4 Multilayer Perceptron

In a multilayer perceptron, the perceptrons (neurons) are stacked in multiple layers in such a way that every node on each layer is connected to all other nodes on the next layer, without any cycles, characterizing the *feed-forward* nature of the model.

The first layer is the input layer, and its units take the values of the input vector. The last layer is the output layer, and it has one unit for each value the network outputs. In the context of classification, it could have a single unit in the case of binary classification, or K units in the case of K -class classification. All the layers in between these are called hidden layers. It's called hidden because we don't know ahead of time what these units should compute, and this needs to be discovered during learning [GROSSE, 2021](#).

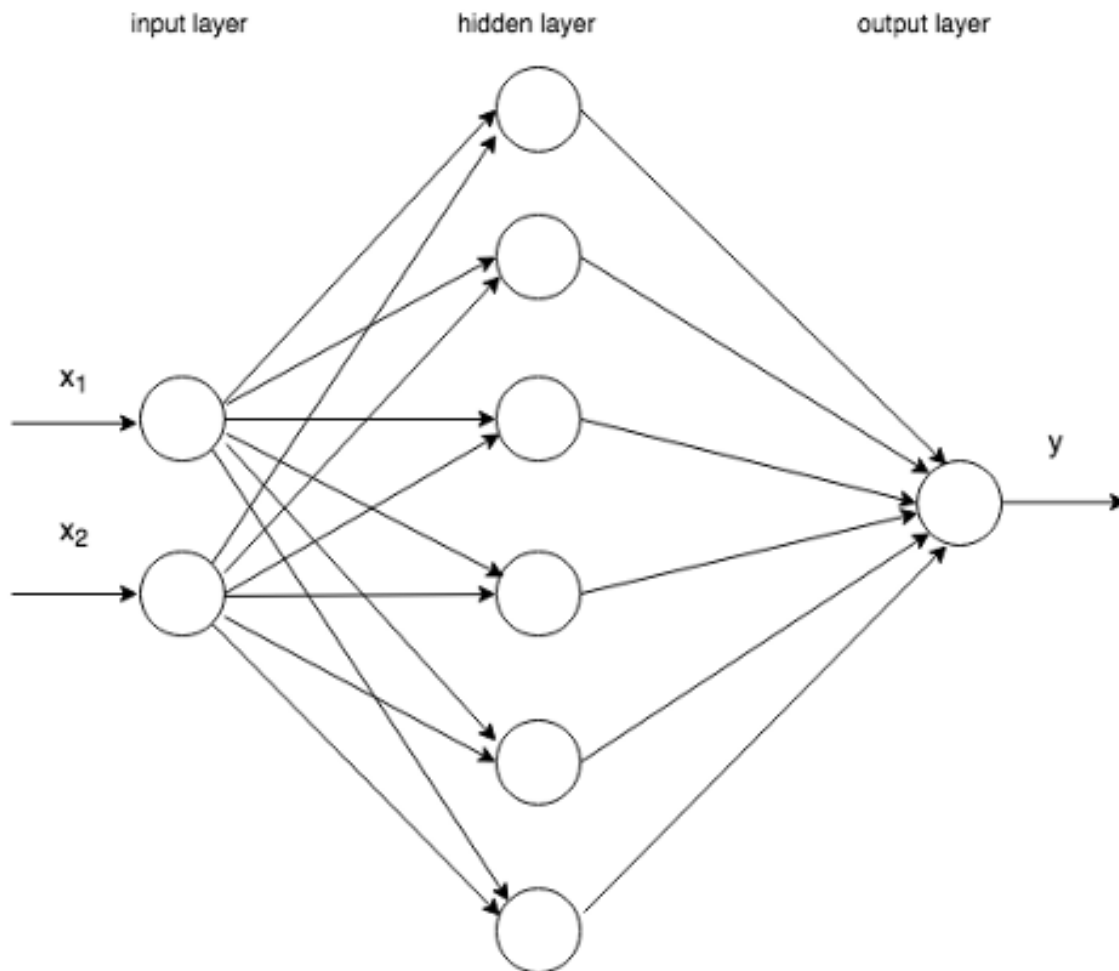


Figure 4.3: Source: *Medium*

Such as the perceptron, the MLP can also be defined as a function. Denote by $h_i^{(l)}$ the i -th unit in the l -th hidden layer and by y the output unit. Note that each unit has its own bias, and there's a weight for every pair of units in two consecutive layers. Therefore, the

network's computation can be expressed as:

$$\begin{aligned} h_i^{(1)} &= \phi^{(1)} \left(\sum_j w_{ij}^{(1)} x_j + b_i^{(1)} \right) \\ h_i^{(2)} &= \phi^{(2)} \left(\sum_j w_{ij}^{(2)} h_j^{(1)} + b_i^{(2)} \right) \\ y_i &= \phi^{(3)} \left(\sum_j w_{ij}^{(3)} h_j^{(2)} + b_i^{(3)} \right) \end{aligned} \quad (4.1)$$

An important result for MLP models is the Universal Approximation Theorem [CYBENKO, 1989](#):

Theorem 4.1: Universal Approximation Theorem

Let $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ be a continuous, bounded, non-constant and monotonically decreasing function. Let $I_n = [0, 1]^n$ be the n -dimensional hypercube and $C(I_n)$ the space of continuous functions in I_n . Then, given $f \in C(I_n)$ independent of σ and $\epsilon > 0$, there exists $N \in \mathbb{N}$, $w_i \in \mathbb{R}^n$, $b_i \in \mathbb{R}$ and $\alpha_i \in \mathbb{R}$, where $i = 1, \dots, N$ such that

$$\begin{aligned} F(x) &= \sum_{i=1}^N \alpha_i \sigma(w_i \cdot x + b_i), \\ |F(x) - f(x)| &< \epsilon, \end{aligned}$$

for all $x \in I_n$.

The theorem establishes that for any continuous function f on a compact subset of I_n can be approximated by a feed-forward neural network with only one hidden layer and finite number of units.

It's important to note that this doesn't imply that one neural network can accurately approximate any arbitrary continuous function under any circumstances. It is required that the neural network and its parameters (number of hidden units, number of learning iterations etc.) be adjusted for each unique function. Then, with the right parameters, it is possible to achieve any desired accuracy ϵ .

Cybenko proved the theorem specifically for *sigmoid* activation functions $Sigmoid(x) = \frac{1}{1 + e^{-x}}$

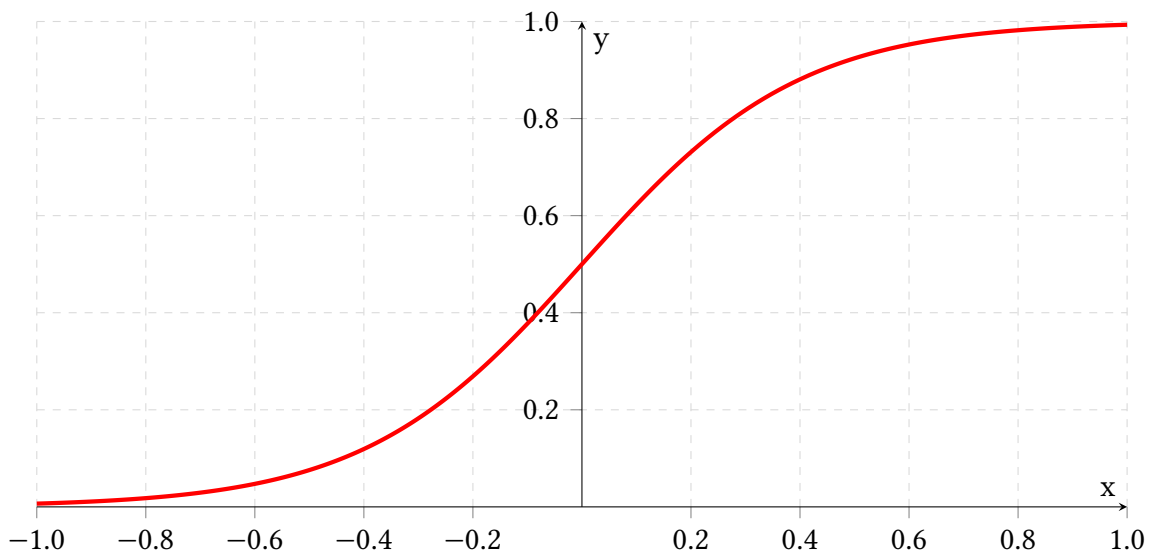


Figure 4.4: Graph for the Sigmoid function.

Other activation functions have also been shown to satisfy the theorem, such as:

- Hyperbolic Tangent

$$\phi(z) = \frac{2}{1 + e^{-2z}} - 1 = 2\text{Sigmoid}(2z) - 1$$

- Softsign

$$\phi(z) = \frac{z}{1 + |z|}$$

Recently, other activation functions have also proven to be very effective as approximators, but don't satisfy the theorem, since they're not bounded:

- Rectified Linear Unit (ReLU)

$$\phi(z) = \max\{0, z\}$$

- Softplus

$$\phi(z) = \ln(e^z + 1)$$

- Exponential Linear Unit (ELU)

$$\phi(z) = \begin{cases} \alpha(e^z - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

for $\alpha \in \mathbb{R}$.

The theorem also has an extension for the case with multiple hidden layers and outputs, but will not be addressed here.

Chapter 5

Policy Gradient

Chapter 6

Trust Region Policy Optimization

Chapter 7

Experiments

Appendix A

Código-fonte e pseudocódigo

Com a *package listings*, programas podem ser inseridos diretamente no arquivo, como feito no caso do Programa ??, ou importados de um arquivo externo com o comando `\lstinputlisting`, como no caso do Programa A.1.

Program A.1 Máximo divisor comum (arquivo importado).

```

1  FUNCTION euclid( $a, b$ )  $\triangleright$  The g.c.d. of  $a$  and  $b$ 
2       $r \leftarrow a \bmod b$ 
3      while  $r \neq 0$   $\triangleright$  We have the answer if  $r$  is 0
4           $a \leftarrow b$ 
5           $b \leftarrow r$ 
6           $r \leftarrow a \bmod b$ 
7      end
8      return  $b$   $\triangleright$  The g.c.d. is  $b$ 
9  end
```

Trechos de código curtos (menores que uma página) podem ou não ser incluídos como *floats*; trechos longos necessariamente incluem quebras de página e, portanto, não podem ser *floats*. Com *floats*, a legenda e as linhas separadoras são colocadas pelo comando `\begin{program}`; sem eles, utilize o ambiente `programruledcaption` (atenção para a colocação do comando `\label{}`, dentro da legenda), como no Programa A.2¹:

Program A.2 Máximo divisor comum (em português).

```

1  FUNCAO euclides( $a, b$ )  $\triangleright$  O máximo divisor comum de  $a$  e  $b$ 
2       $r \leftarrow a \bmod b$ 
3  enquanto  $r \neq 0$   $\triangleright$  Atingimos a resposta se  $r$  é zero
4       $a \leftarrow b$ 
5       $b \leftarrow r$ 
```

cont \longrightarrow

¹ listings oferece alguns recursos próprios para a definição de *floats* e legendas, mas neste modelo não os utilizamos.

```

→ cont
6       $r \leftarrow a \bmod b$ 
7      fim
8  devolva  $b \triangleright$  O máximo divisor comum é  $b$ 
9  fim

```

Além do suporte às várias linguagens incluídas em listings, este modelo traz uma extensão para permitir o uso de pseudocódigo, útil para a descrição de algoritmos em alto nível. Ela oferece diversos recursos:

- Comentários seguem o padrão de C++ (`//` e `/* ... */`), mas o delimitador é impresso como “ \triangleright ”.
- “`:=`”, “`<>`”, “`<=`”, “`>=`” e “`!=`” são substituídos pelo símbolo matemático adequado.
- É possível acrescentar palavras-chave além de “if”, “and” etc. com a opção “`morekeywords={pchave1,pchave2}`” (para um trecho de código específico) ou com o comando `\lstset{morekeywords={pchave1,pchave2}}` (como comando de configuração geral).
- É possível usar pequenos trechos de código, como nomes de variáveis, dentro de um parágrafo normal com `\lstinline{blah}`.
- “`$....$`” ativa o modo matemático em qualquer lugar.
- Outros comandos \LaTeX funcionam apenas em comentários; fora, a linguagem simula alguns pré-definidos (`\textit{}`, `\texttt{}` etc.).
- O comando `\label` também funciona em comentários; a referência correspondente (`\ref`) indica o número da linha de código. Se quiser usá-lo numa linha sem comentários, use `/// \label{blah}`; “`///`” funciona como `//`, permitindo a inserção de comandos \LaTeX , mas não imprime o delimitador (\triangleright).
- Para suspender a formatação automática, use `\noparse{blah}`.
- Para forçar a formatação de um texto como função, identificador, palavra-chave ou comentário, use `\func{blah}`, `\id{blah}`, `\kw{blah}` ou `\comment{blah}`.
- Palavras-chave dentro de comentários não são formatadas automaticamente; se necessário, use `\func{}`, `\id{}` etc. ou comandos \LaTeX padrão.
- As palavras “Program”, “Procedure” e “Function” têm formatação especial e fazem a palavra seguinte ser formatada como função. Funções em outros lugares *não* são detectadas automaticamente; use `\func{}`, a opção “`functions={func1,func2}`” ou o comando “`\lstset{functions={func1,func2}}`” para que elas sejam detectadas.
- Além de funções, palavras-chave, strings, comentários e identificadores, há “`specialidentifiers`”. Você pode usá-los com `\specialid{blah}`, com a opção “`specialidentifiers={id1,id2}`” ou com o comando “`\lstset{specialidentifiers={id1,id2}}`”.

Annex A

Perguntas frequentes sobre o modelo²

- **Não consigo decorar tantos comandos!**

Use a colinha que é distribuída juntamente com este modelo (gitlab.com/ccsl-usp/modelo-latex/raw/master/pre-compilados/colinha.pdf?inline=false).

- **Por que tantos arquivos?**

O preâmbulo \LaTeX deste modelo é muito longo; as partes que normalmente não precisam ser modificadas foram colocadas no diretório extras, juntamente com alguns arquivos acessórios.

- **Estou tendo problemas com caracteres acentuados.**

Versões modernas de \LaTeX usam UTF-8, mas arquivos antigos podem usar outras codificações (como ISO-8859-1, também conhecido como latin1 ou Windows-1252). Nesses casos, use `\usepackage[latin1]{inputenc}` no preâmbulo do documento. Você também pode representar os caracteres acentuados usando comandos \LaTeX : `\a` para á, `\c{c}` para cedilha etc., independentemente da codificação usada no texto³.

- **Aparece uma folha em branco entre os capítulos.**

Essa característica foi colocada propositalmente, dado que todo capítulo deve (ou deveria) começar em uma página de numeração ímpar (lado direito do documento). Se quiser mudar esse comportamento, acrescente “openany” como opção da classe, i.e., `\documentclass[openany,...]{book}`.

- **É possível resumir o nome das seções/capítulos que aparece no topo das páginas e no sumário?**

Sim, usando a sintaxe `\section[mini-titulo]{titulo enorme}`. Isso é especialmente útil nas legendas (*captions*) das figuras e tabelas, que muitas vezes são demasiadamente longas para a lista de figuras/tabelas.

² Esta seção não é de fato um anexo, mas sim um apêndice; ela foi definida desta forma apenas para servir como exemplo de anexo.

³ Você pode consultar os comandos desse tipo mais comuns em en.wikibooks.org/wiki/LaTeX/Special_Characters. Observe que a dica sobre o pingão `\i` não é mais válida atualmente; basta usar `\i`.

- **Existe algum programa para gerenciar referências em formato bibtex?**

Sim, há vários. Uma opção bem comum é o JabRef; outra é usar Zotero ou Mendeley e exportar os dados deles no formato .bib.

- **Posso usar pacotes L^AT_EX adicionais aos sugeridos?**

Com certeza! Você pode modificar os arquivos o quanto desejar, o modelo serve só como uma ajuda inicial para o seu trabalho.

- **Como faço para usar o Makefile (comando make) no Windows?**

Lembre-se que a ferramenta recomendada para compilação do documento é o latexmk, então você não precisa do make. Mas, se quiser usá-lo, você pode instalar o MSYS2 (www.msys2.org) ou o Windows Subsystem for Linux (procure as versões de Linux disponíveis na Microsoft Store). Se você pretende usar algum dos editores sugeridos, é possível deixar a compilação a cargo deles, também dispensando o make.

References

- [CYBENKO 1989] George CYBENKO. “Approximation by superpositions of a sigmoidal function.” In: *Math. Control. Signals Syst.* 2.4 (1989), pp. 303–314. URL: <http://dblp.uni-trier.de/db/journals/mcss/mcss2.html#Cybenko89> (cit. on p. 16).
- [GOODFELLOW *et al.* 2016] Ian GOODFELLOW, Yoshua BENGIO, and Aaron COURVILLE. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016 (cit. on p. 11).
- [GROSSE 2021] Roger GROSSE. *Lecture 5: Multilayer Perceptrons*. URL: https://www.cs.toronto.edu/~rgrosse/courses/csc311_f21/. 2021 (cit. on p. 15).
- [HAYKIN 2009] Simon S. HAYKIN. *Neural networks and learning machines*. Third. Pearson Education, 2009 (cit. on p. 13).
- [MITCHELL 1997] Tom M. MITCHELL. *Machine Learning*. McGraw-Hill, 1997 (cit. on p. 11).
- [ROSENBLATT 1958] Frank ROSENBLATT. “The perceptron: A probabilistic model for information storage and organization in the brain”. In: *Psychological Review* 65 (1958) (cit. on p. 14).
- [SUTTON and BARTO 2015] Richard S. SUTTON and Andrew G. BARTO. *Reinforcement Learning: An Introduction*. The MIT Press, 2015 (cit. on pp. 1, 5, 6).

Index

C

Captions, *see* Legendas
Código-fonte, *see* Floats

E

Equações, *see* Modo matemático

F

Figuras, *see* Floats
Floats
 Algoritmo, *see* Floats, ordem
Fórmulas, *see* Modo matemático

I

Inglês, *see* Língua estrangeira

L

Legendas, [27](#)

M

Mendeley, [28](#)

P

Palavras estrangeiras, *see* Língua estrangeira

R

Rodapé, notas, *see* Notas de rodapé

S

Subcaptions, *see* Subfiguras
Sublegendas, *see* Subfiguras

T

Tabelas, *see* Floats

V

Versão corrigida, *see* Tese/Dissertação, versões
Versão original, *see* Tese/Dissertação, versões

Z

Zotero, [28](#)