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

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

Agradecimentos

Ninguem é tão sábio que não tenha algo para apren-
der e nem tão tolo que não tenha algo a ensinar.
— Blaise Pascal

Resumo

Elemento obrigatório, constituído de uma sequência de frases concisas e objetivas, em forma de texto. Deve apresentar os objetivos, métodos empregados, resultados e conclusões. O resumo deve ser redigido em parágrafo único, conter no máximo 500 palavras e ser seguido dos termos representativos do conteúdo do trabalho (palavras-chave). Deve ser precedido da referência do documento. Texto texto. Texto texto.

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.

Elemento obrigatório, elaborado com as mesmas características do resumo em língua portuguesa. De acordo com o Regimento da Pós-Graduação da USP (Artigo 99), deve ser redigido em inglês para fins de divulgação. É uma boa ideia usar o sítio www.grammarly.com na preparação de textos em inglês. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text. Text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text text.

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
FF	Feed-forward
MLP	Multilayer Perceptron
ReLU	Rectified Linear Unit
ELU	Exponential Linear Unit
GLP	Grad-Log-Prob
EGLP	Expected Grad-Log-Prob
IME	Instituto de Matemática e Estatística
USP	Universidade de São Paulo

Contents

Appendixes

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. Richard S. 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-explotation dillema. Richard S. 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 intuititve 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.
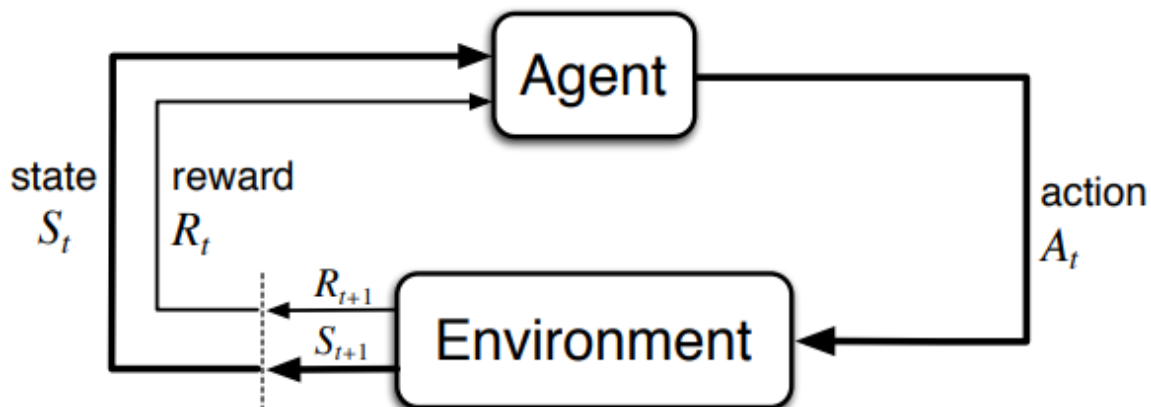

Figure 2.1: *Representation of the agent-environment interface from Richard S. SUTTON and BARTO, 2015.*

In other words, at each time step $t = 0, 1, 2, \ldots$, 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 $\pi(a \mid s)$ 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: $R_{t+1}, R_{t+2}, \ldots$. We define a *return* G_t, which is a function of that sequence. The simplest case is the sum of all of them:

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

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 breaked 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:

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

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 Richard S. 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 ou 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 a action taken at time t. We define the history sequence as $h_t = \{S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\}$. 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

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

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 $\langle S, A, P, R, \gamma \rangle$
- S is a finite set of all valid states
- A is a finite set of all valid actions
- $P : S \times A \to \mathcal{P}(S)$ is the transition probability function with $P[S_{t+1} = s' \mid S_t = s, A_t = a]$ being the probability of transitioning into state s' starting in state s and taking action a
- R is a reward function
  $R_s^a = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$
- γ is a discount factor such that $\gamma \in [0, 1]$

Other authors also define an MDP to be a tuple $\langle S, A, P, R, \rho_0 \rangle$, with ρ_0 being a distribution of starting states.

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, which can be either deterministic or stochastic. The first is a function $\mu : S \to A$ and the action at time t is

$$a_t = \mu(s_t)$$

and the latter is a probability distribution over $a \in A$ for each $s \in S$ denoted by π and the action at time t is sampled from π:

$$a_t \sim \pi(\cdot \mid s_t)$$

Moreover, in this work, we will use *parameterized policies*, whose outputs are computable functions that depend on a set of parameters θ, which can be adjusted using optimization algorithms. Parameterized policies are denoted by

$$a_t = \mu_\theta(s_t)$$
$$a_t = \pi_\theta(\cdot \mid s_t)$$

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

> **Definition 3.3: 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
>
> $$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right] \qquad (3.2)$$

and the action-value function.

> **Definition 3.4: 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
>
> $$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right] \qquad (3.3)$$

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:

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

and then we are able to formulate what is a optimal policy.

> **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 v_{π_*}.

> **Definition 3.5: Optimal State-Value Function**
>
> The optimal state-value function v_{π_*} or $v_*(s)$ is the expected return starting from state s and always acting according to the optimal policy.
>
> $$v_*(s) = \max_\pi v_\pi(s)$$

> **Definition 3.6: Action-Value Function**
>
> The optimal action-value function q_{π_*} or $q_*(s)$ is the expected return starting from state s, taking an arbitrary action a and then always act according to the optimal policy.
>
> $$q_*(s, a) = \max_{\pi} q_{\pi}(s, a)$$

Now, recall that the fundamental objective of reinforcement learning problems is to maximize rewards on the long run. One way to do that is using *Bellman Equations*, which enables us to calculate the optimal value functions defined above using recursive relationships and dynamic programming.

However, in this work, we'll focus on a particular class of methods called *Proximal Policy Optimization*, and for that, it isn't necessary to know how good a state/action is, but only how much better it is compared to others. We formalize this concept by defining the *advantage function*:

> **Definition 3.7: Advantage Function**
>
> The advantage function $A^{\pi}(s, a)$ describes how much better it is to take a specific action a in state s, over randomly selecting an action according to $\pi(\cdot \mid s)$ and following policy π forever after.
>
> $$A_{\pi}(s, a) = q_{\pi}(s, a) - v_{\pi}(s)$$

Chapter 4

Neural Networks

In this chapter, some concepts on *Neural Networks* (or NNs) will be discussed as well as a little bit of *Machine Learning*. As stated in the previous chapter, we'll be working with parameterized policies, and NNs will later be used as an approximator for said policies.

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 paterns 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.
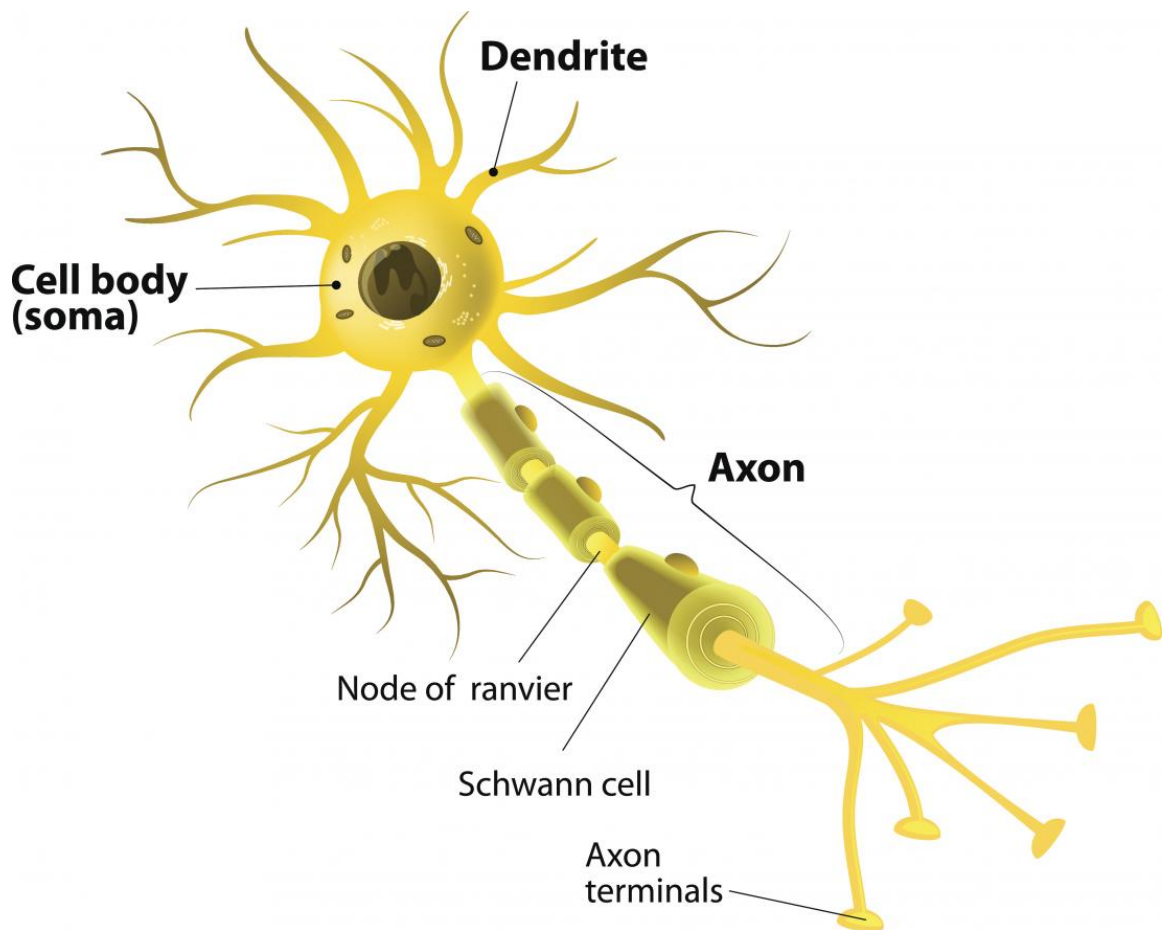

Figure 4.1: *Source: MedicalNewsToday*

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

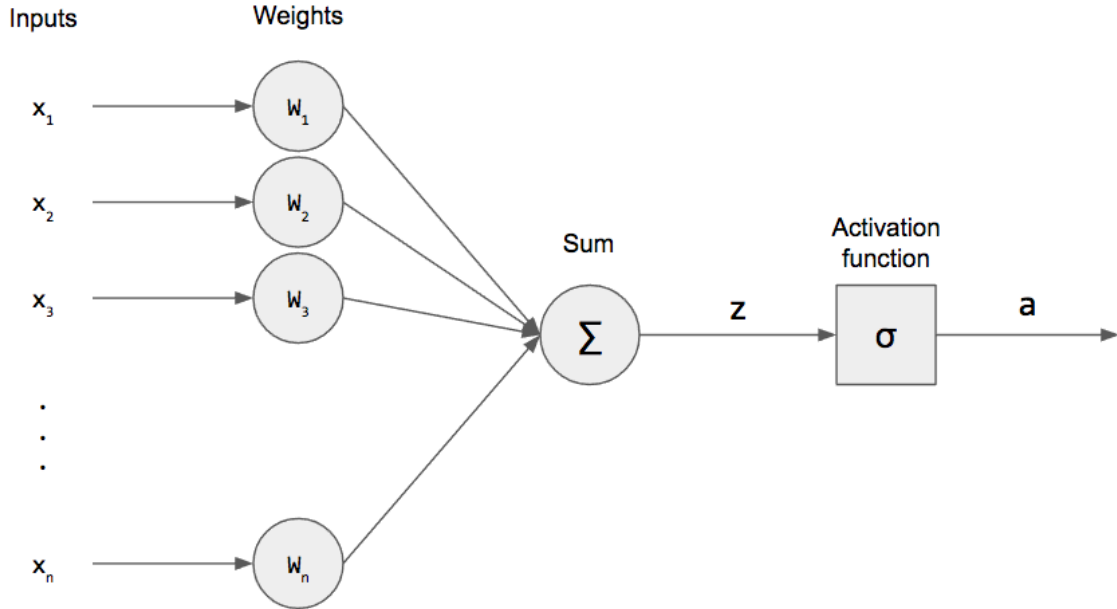probabilistic model for information storage and organization in the brain" ROSENBLATT, 1958.

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 $\phi : \mathbb{R} \to \mathbb{R}$ 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 $h : \mathbb{R}^n \to \mathbb{R}$ 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 mutiple 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 classifiation, 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.
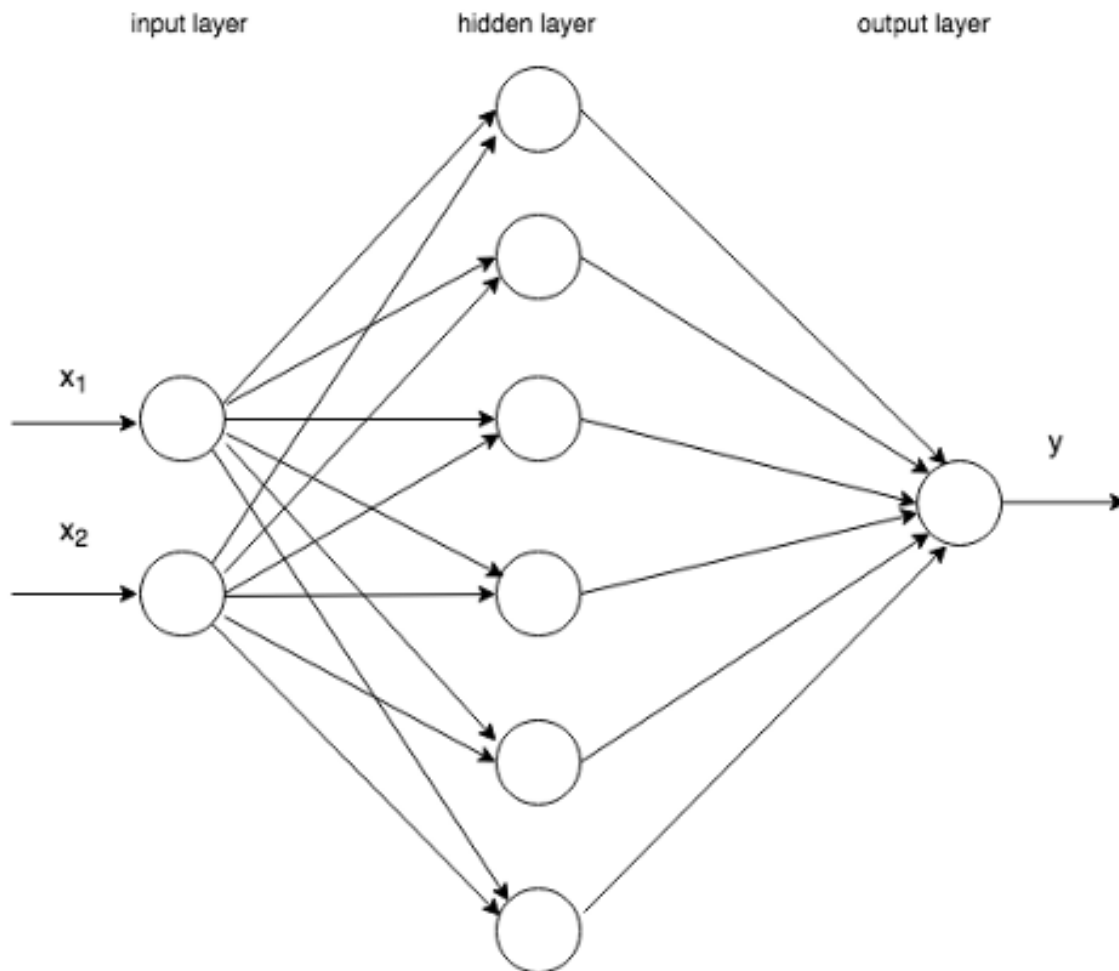

Figure 4.3: *Source: Medium*

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:

$$h_i^{(1)} = \phi^{(1)}\left(\sum_j w_{ij}^{(1)} x_j + b_i^{(1)}\right)$$

$$h_i^{(2)} = \phi^{(2)}\left(\sum_j w_{ij}^{(2)} h_j^{(1)} + b_i^{(2)}\right) \tag{4.1}$$

$$y_i = \phi^{(3)}\left(\sum_j w_{ij}^{(3)} h_j^{(2)} + b_i^{(3)}\right)$$

An important result for MLP models is the Universal Approximation Theorem CYBENKO, 1989:

Theorem 4.1: Universal Approximation Theorem

et $\sigma : \mathbb{R} \to \mathbb{R}$ 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 $i = 1, \ldots, N$ such that

$$F(x) = \sum_{i=1}^{N} \alpha_i \sigma(w_i x + b_i),$$

$$|F(x) - f(x)| < \epsilon$$

for all $x \in I_n$.

The theorem estabilishes 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 the *sigmoid* activation function, defined as:
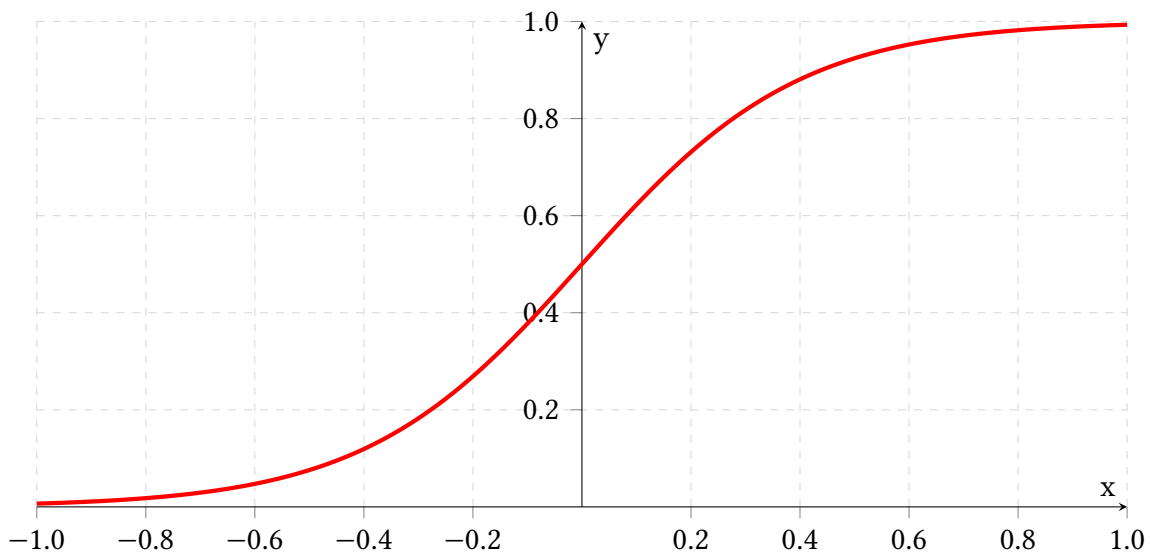
$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

Figure 4.4: *Graph for the Sigmoid function.*

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

- Hyperbolic Tangent

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

- Softsign

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

Recently, some unbounded activation functions have also proven to be very effective as approximators, even while not satisfying the theorem:

- 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.

Below, we have an implementation in Python of a MLP, using Pytorch, one of the most famous machine learning frameworks for Python. This is the code for the actual neural network we'll use later. We are using a simple network with only one hidden

---

Program 4.1 Simple feed-forward MLP

```python
import torch

class FeedForwardNN(torch.nn.Module):
    def __init__(self, in_dim, out_dim):
        """
            Initializes the network and set up the layers.
        """
        super(FeedForwardNN, self).__init__()

        self.layer1 = torch.nn.Linear(in_dim, 64)
        self.layer2 = torch.nn.Linear(64, 64)
        self.layer3 = torch.nn.Linear(64, out_dim)

    def forward(self, obs):
        """
            Runs a forward pass on the neural network.
        """
        if isinstance(obs, np.ndarray):
            obs = torch.tensor(obs, dtype=torch.float)

        activation1 = torch.nn.functional.relu(self.layer1(obs))
        activation2 = torch.nn.functional.relu(self.layer2(activation1))
        output = self.layer3(activation2)

        return output
```

---

layer with 64 units and a single input and output. As described in the official documentation, the torch.nn.Linear function applies the linear transformation $y = w^T x + b$ and torch.nn.functional.relu is the ReLU activation function.

Chapter 5

Proximal Policy Optimization

Proximal policy optimization (PPO) was proposed by Schulman, Wolski, *et al.*, 2017 as an alternative to already existing policy gradient (PG) methods, incorporating some concepts from trust region policy optimization (TRPO) methods, retaining some of its benefits while being significantly easier to implement.

5.1   Policy Gradient

For policy gradient, we consider parameterized policies, which can select actions without relying on a value function. The value function is still useful to learn the policy parameters, but it's not strictly necessary to select an action. This parameterization can be done in any way as long as the policy is differentiable with respect to its parameters.

Denoting by $\theta \in \mathbb{R}^d$ the policy parameter vector, the probability of selecting action a at time t given that the environment is in state s with parameter θ is

$$\pi_\theta(a \mid s, \theta) = P\{A_t = a \mid S_t = s, \theta_t = \theta\}$$

We also need to define a performance measure $J(\theta)$ to quantify how good a policy is. Policy gradient algorithms search for a local maximum in J using gradient ascent:

$$\Delta\theta = \alpha\nabla_\theta J(\theta)$$

where $\nabla_\theta J(\theta)$ is the policy gradient defined as

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix}$$

and α is a step size parameter, commonly called *learning rate*.

In our case, we define such measure to be $J(\theta) = v_{\pi_\theta}(s_0)$, where s_0 denotes the starting state and v_{π_θ} is the true value function for the parameterized policy π_θ. That is, the measure is the expected sum of discounted rewards.

For that particular choice of $J(\theta)$, we have a convenient way of expressing its gradient using the Policy Gradient Theorem (adapted from Richard S SUTTON *et al.*, 1999):

Theorem 5.1: Policy Gradient Theorem

Given an MDP $\langle S, A, P, R, \gamma \rangle$ and a parameterized policy π_θ, the gradient of the expected return $J(\theta)$ is given by

$$\nabla_\theta J(\theta) \propto \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} q_{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(a \mid s) \tag{5.1}$$

Where $d^{\pi_\theta}(s) = \lim_{t \to \infty} P\{s_t = s \mid s_0, \pi\}$ is the stationary distribution of states under policy π, which is assumed to be independent of the starting state s_0. Furthermore, we can use the following identity, commonly called the "log derivative trick", to rewrite the expression for the gradient:

$$\nabla_\theta \pi_\theta(a \mid s) = \pi_\theta(a \mid s) \frac{\nabla_\theta \pi_\theta(a \mid s)}{\pi_\theta(a \mid s)}$$
$$= \pi_\theta(a \mid s) \nabla_\theta \log \pi_\theta(a \mid s)$$

Then, using the identity above, we are able to express the gradient as an expectation:

$$\nabla_\theta J(\theta) \propto \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} q_{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(a \mid s) \tag{5.2}$$
$$= \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} q_{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(a \mid s) \frac{\pi_\theta(a \mid s)}{\pi_\theta(a \mid s)}$$
$$= \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} q_{\pi_\theta}(s, a) \pi_\theta(a \mid s) \frac{\nabla_\theta \pi_\theta(a \mid s)}{\pi_\theta(a \mid s)}$$
$$= \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} q_{\pi_\theta}(s, a) \pi_\theta(a \mid s) \nabla_\theta \log \pi_\theta(a \mid s)$$
$$= \mathbb{E}_\pi [q_{\pi_\theta}(s, a) \nabla_\theta \log \pi_\theta(a \mid s)] \tag{5.3}$$

where $\mathbb{E}_\pi$ is the expectation when $s \sim d^{\pi_\theta}$ and $a \sim \pi_\theta$, i.e. both state and action distributions follow policy π_θ. Expressing the gradient as an expectation means we can estimate it using a sample mean. We let the agent interact with the environment following a policy π_θ and collect its *trajectory* $\tau_i = \{s_0, a_0, \dots, s_{T+1}\}$ over N episodes, obtaining a set $\mathcal{D} = \{\tau_i\}_{i=1,\dots,N}$ of trajectories. Then, the policy gradient is estimated as:

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} q_{\pi_\theta}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \tag{5.4}$$

That is, we compute the expression inside the expectation in each episode and take the sample mean as an estimator for the gradient, allowing us to take an update step.

The policy gradient, while elegant, has shown to be problematic in practical problems:

Sample inefficiency. In order to run policy gradient, we need to sample from our

policy millions and millions of times, since the estimation is done using Monte Carlo, averaging over a number of trial runs. Summing over all steps in a single trajectory could be very expensive computationally depending on the environment. It is also worth noting sample inefficiency is not a problem exclusive to policy gradient, it is an issue that has long plagued a lot of other RL algorithms.

Slow convergence. Sampling millions of trajectories is already inherently slow, and the high variance makes optimization very inefficient.

High variance. The high variance comes from the fact that, in RL, we are often dealing with very general problems. In our case, teaching a car to navigate through a parking lot. When we sample from an untrained policy, we are bound to observe highly variable behaviors, since we begin with a policy whose distribution of actions over states is effectively uniform. Of course, as the policy improves, the distribution is shaped to be unimodal or multimodal over some successful actions given a state. But in order to get there, we need the model to observe the outcomes of many different actions for each possible state. If we consider the action and state spaces to be continuous, the problem is even worse, since visiting every action-state pair possible is computationally intractable.

Equation 5.3, also known as *grad-log-prob*, gives rise to an important result, which was used to derive a lot of other methods as an improvement over the *vanilla* policy gradient.

Lemma 5.1: Expected Grad-Log-Prob (EGLP)

Suppose that P_θ is a parameterized probability distribution over a random variable x. Then:

$$\underset{x\sim P_\theta}{\mathbb{E}}\left[\nabla_\theta \log P_\theta(x)\right] = 0 \tag{5.5}$$

Proof. First, recall that probability distributions are normalized:

$$\int_x P_\theta(x)\,dx = 1 \tag{5.6}$$

By taking the gradient on both sides, we get:

$$\nabla_\theta \int_x P_\theta(x)\,dx = \nabla_\theta 1 = 0 \tag{5.7}$$

Now, we can use the log derivative trick:

$$\nabla_\theta \int_x P_\theta(x)\,dx = 0$$
$$\int_x \nabla_\theta P_\theta(x)\,dx = 0 \tag{5.8}$$
$$\int_x P_\theta(x)\nabla_\theta \log P_\theta(x)\,dx = 0$$

$\square$

As an consequence of the above lemma, Schulman, Moritz, *et al.*, 2015 proposed a more general form for policy gradients:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \Phi_t \nabla_\theta \log \pi_\theta(a \mid s) \right] \tag{5.9}$$

where Φ_t may be any of these functions:

- $\sum_{t=0}^{\infty} r_t$: sum of total rewards.

- $\sum_{t'=t}^{\infty} r_{t'}$: reward following action a_t.

- $Q^\pi(s_t, a_t)$: state-action value function.

- $A^\pi(s_t, a_t)$: advantage function.

Schulman, Moritz, *et al.*, 2015 lists all the possible functions. For trust region methods and proximal policy optimization, Φ_t is chosen to be the advantage function $A_\pi(s_t, a_t)$.

The formulation of policy gradients with advantage functions is rather common, but we then need a way to estimate it. In this work, we will use *generalized advantage estimation* as described in Schulman, Moritz, *et al.*, 2015.

5.2   Trust Region Methods

TRPO updates policies by taking the largest step possible to improve performance, while satisfying a special constraint on how close the new and old policies are allowed to be. The constraint is expressed in terms of KL-Divergence, a measure of (something like, but not exactly) distance between probability distributions.

This is different from normal policy gradient, which keeps new and old policies close in parameter space. But even seemingly small differences in parameter space can have very large differences in performance—so a single bad step can collapse the policy performance. This makes it dangerous to use large step sizes with vanilla policy gradients, thus hurting its sample efficiency. TRPO nicely avoids this kind of collapse, and tends to quickly and monotonically improve performance.

Chapter 6

Experiments

6.1   Experiment 1: Fixed Spawns

6.2   Experiment 2: Randomized Spawns

6.2.1   Randomized Car Spawn and Fixed Parking Spot

6.2.2   Randomized Car and Parking Spot Spawns

6.3   Experiment 3: Parallel Parking

Appendix A

Código-fonte e pseudocódigo

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).

[Schulman, Moritz, *et al.* 2015]   John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2015. doi: 10.48550/ARXIV.1506.02438. url: https://arxiv.org/abs/1506.02438 (cit. on p. 22).

[Schulman, Wolski, *et al.* 2017]   John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms*. 2017. doi: 10.48550/ARXIV.1707.06347. url: https://arxiv.org/abs/1707.06347 (cit. on p. 19).

[Richard S Sutton *et al.* 1999]   Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in Neural Information Processing Systems*. Ed. by S. Solla, T. Leen, and K. Müller. Vol. 12. MIT Press, 1999. url: https: //proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f- Paper.pdf (cit. on p. 20).

[Richard S. 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).