Playing the Briscola game

Daniel Orzan

Università degli Studi di Trieste
Dipartimento di Matematica e Groscienze
Laurea Magistrale in Data Science and Scientific Computing

March 2024

Supervisor: Prof. Antonio Celani
Co-supervisors: Emanuele Panizon, Lorenzo Basile

Abstract

This thesis explores the application of the Decision Transformer (DT) model (Chen et al. 2021), a novel architecture inspired by the Transformer model, to the traditional Italian card game Briscola. Briscola is known for its strategic depth and large decision space, making it an interesting domain for studying Reinforcement Learning algorithms. The experiments analyze the model's ability to play against both human players and BriscolaBot (Lorenzo 2023), the agent that generated the dataset for our training. The results suggest that the Decision Transformer is a powerful framework for solving complex decision-making tasks, as the trained agent is able to achieve a winning rate of 60% against average human players. Unfortunately, it does not manage to surpass the performance of BriscolaBot. The code for the agent training, testing and human evaluation can be found in the GitHub repository BriscolaOfflineRL.

Acknowledgements

Ringrazio tutti coloro che mi sono stati vicini e mi hanno sostenuto durante questo percorso, senza mai perdere la fiducia che sarei arrivato a questo traguardo.

Ringrazio il mio relatore Antonio Celani, che ha avuto la pazienza di seguirmi per i lunghi mesi passati in cui la costanza non mi è sempre stata amica.

Ringrazio i miei correlatori Emanuele Panizon e Lorenzo Basile, che sono stati di fondamentale aiuto per la buona riuscita di questo progetto.

Infine, ringrazio Lorenzo Cavuoti, senza il quale questa tesi non sarebbe potuta esistere.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Briscola is one of Italy's most popular games for two to six players, played with a standard deck of forty cards. It requires players to make decisions based on incomplete information, predict opponents' moves, and adapt their strategies dynamically.
In this thesis, we explore the application of the Decision Transformer (Chen et al. 2021), a state-of-the-art variant of the transformer architecture, to the task of learning optimal strategies for playing Briscola. The DT model offers several advantages, including the ability to capture long-range dependencies, handle sequential data efficiently, and incorporate multi-head attention mechanisms for robust feature learning.
Our primary objective is to develop an autonomous agent able to play Briscola at a competitive level against human opponents. To achieve this goal, we exploit a large dataset of Briscola games as the basis for training our DT-based agent.

1.1 Briscola

The forty cards deck is divided into four suits: coins (Denari), swords (Spade), cups (Coppe) and batons (Bastoni); in every game one of these suits plays the role of the trump suit, called *briscola*.
The values on the cards range numerically from one through seven, plus three face cards in each suit: Knave (Fante), Knight (Cavallo), and King (Re) (Wikipedia 2024). Cards rank, beginning from the strongest, is as follows: ace, three, king, knight, and knave; point values are respectively 11, 10, 4, 3 and 2. The remaining cards have no point value, and are ranked in descending ordinal value, from seven to two. Overall, there are 120 card points and, to win a game, a player must accumulate more points than any other player. If two players, or teams, have the same number of points, which would be 60, the game is a tie.

A game is played as described next. At the start each player is dealt three cards and the next card is placed face up on the playing surface. This card is the *briscola* and represents the trump suit for the game. The player to the right of the dealer plays first and the other players subsequently play a card in turn, until everyone has played one card. The winner of the trick is determined as follows:

- if any *briscola* has been played, the player who played the highest valued *briscola* wins;
- if no briscole have been played, the player who played the highest card of the lead suit wins, where the lead suit is the first played card's suit.

Once the winner of a trick is determined, that player collects the played cards. Then, each player draws a card from the remaining deck, starting with the player who won the trick and will play the next one first. The last card collected in the game will be the up-turned *briscola.* At the end of the game, players calculate the total point value of cards in their own piles and the winner is chosen as explained above.

1.2 Reinforcement Learning

Reinforcement learning is an area of machine learning, alongside supervised learning and unsupervised learning, concerned with learning how an agent behaves, how it makes decisions and takes actions to maximize the cumulative reward.
A branch of RL is dedicated to the problem of finding solutions where experience only arises from the interaction with an environment. Its focus is then in finding a balance between exploration of uncharted territory and exploitation of current knowledge.
At its core, RL is inspired by the way humans and animals learn from trial and error. Just as a child learns to walk by taking steps, falling, and adjusting their movements based on the consequences of each step, RL algorithms enable machines to learn optimal behaviors by continuously exploring and adapting to their surroundings. This learning paradigm has found applications in a wide range of domains, from game playing to robotics and autonomous systems, where it has been successful in solving many complex decision-making tasks that were previously out of reach for a machine.

1.3 Transformers

Transformers are a type of neural network architecture that transforms or changes an input sequence into an output sequence. Leaving the details for the following chapters, a transformer is based on the multi-head attention mechanism, which allows it to determine the relative importance of different parts of a sequence, trying to select information that is “useful” when generating the output sequence.

All transformers have the same primary components:

- Tokenizers: a single embedding layer, which converts tokens and positions of the tokens into vector representations.
- Transformer layers: they carry out repeated transformations on the vector representations, extracting more and more linguistic information. These consist of alternating the attention mechanism and feedforward layers.

Transformer layers can be one of two types, encoder and decoder. In Vaswani et al. 2017 both of them were used, while later models included only one type of them (Ghojogh and Ghodsi 2020).

1.4 Games of imperfect-information

Imperfect-information games model settings where players have some private information (Sandholm 2015). They pose significant challenges for AI and decision-making, as they require agents to handle uncertainty, make probabilistic decisions, and adapt to opponents' strategies based on the available information. Classical solutions involve the Nash equilibrium, a decision-making theorem within game theory that states a player can achieve the desired outcome by not deviating from their initial strategy (Charles A. Holt 2004).
No-limit Texas hold'em is the most popular form of poker in the world and has been the primary benchmark challenge for AI in imperfect-information games (Noam Brown 2017).

1.5 Outline

The next chapter 2 provides a thorough background about Reinforcement Learning, from what an MDP is to the main techniques used in this branch of Machine Learning, such as dynamic programming, Monte Carlo methods, temporal-difference learning.
Chapter 3 continues the discussion about RL by focusing on its offline application. As a matter of fact, in this work training is not conducted making the agent interact with an environment but exploiting a dataset. In this way the task becomes a sequence modelling problem and the Decision Transformer used for solving it is illustrated in the following chapter 4 together with some fundamental concepts and architectures of Machine Learning and Deep Learning.
Chapter 5 starts by describing the above-mentioned dataset and continues with the original work of the thesis, from training and testing up to the results of the agent in

real games, with analyses of its gameplay behavior.
The last chapter 6 contains a discussion about the findings and some conclusions on the method used, suggesting how all this could be taken further trying different approaches or improving the presented one.

Chapter 2

Background

Reinforcement learning is a machine learning paradigm which aims at finding ways to teach machines how to act and learn in an environment.

It is distinguished by two fundamental characteristics: trial-and-error learning and delayed reward feedback. In reinforcement learning, an agent interacts with an environment, taking actions and receiving feedback in the form of rewards or penalties. Trial-and-error learning reflects the agent's iterative process of making decisions, beginning with random choices and gradually improving the strategy by checking action's consequences and building experience.
Delayed reward, on the other hand, represents the challenge of associating actions with their long-term consequences, as greater rewards might be received after a sequence of actions. Hence, agents must learn to make decisions that optimize cumulative rewards and not only immediate rewards, which may even be unfavorable.

Exploration and exploitation are other two pivotal concepts within the framework of reinforcement learning, tightly linked to the trial-and-error and delayed reward characteristics. Exploration represents the agent's need to try new actions or strategies to discover potentially better ones, which is essential for learning optimal policies. On the flip side, exploitation involves choosing actions that the agent believes will yield the highest immediate reward based on its current knowledge.
Balancing these two aspects is one of the main challenges in reinforcement learning. Too much exploration can make the agent move away from good solutions, while over-exploitation can prevent the discovery of better strategies. Effective exploration strategies, such as epsilon-greedy policies, are vital for finding the right trade-off between exploring uncharted space and exploiting known options.

2.1 Finite Markov Decision Processes

This section provides the core components of reinforcement learning, from the agent-environment interaction, rewards, returns, policies, to value functions.

2.1.1 The Agent–Environment Interface

Agent and environment are what reinforcement learning is built upon. The agent is a learning and decision-making entity, and it interacts with the environment, the "everything" outside the agent. The agent takes actions which influence the state of the environment, which in turn gives a feedback: the reward (Figure 2.1).

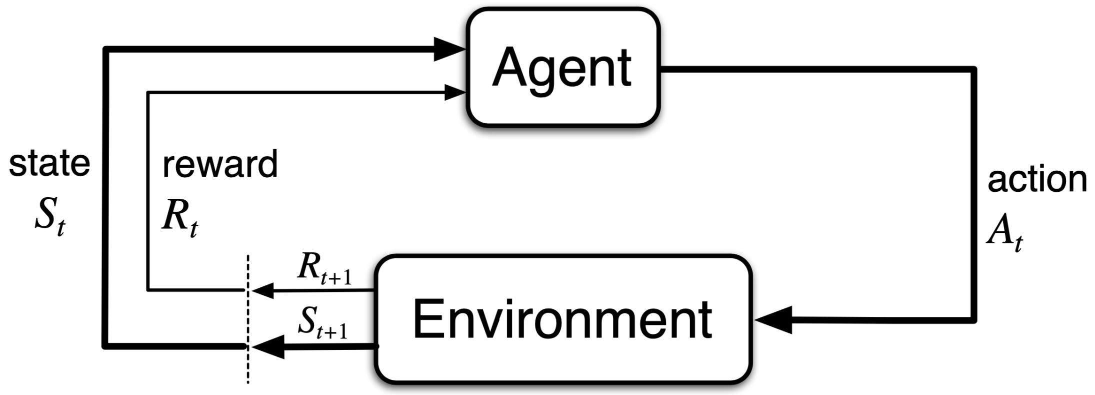


Figure 2.1: Agent-Environment Interface

The following is a formal explanation of this framework:

- States: The environment is modeled as a set of states, denoted as S, which represents all possible situations or configurations the environment can be in. These states capture all the relevant information about the environment at every time t.
- Actions: The agent interacts with the environment by selecting an action from a set of available actions, denoted as A. Thus, actions represent the decisions or choices made by the agent that carry the environment to a new state.
- Rewards: At each time step, after taking an action, the agent receives a numerical feedback called the reward, denoted as $R(s, a, s')$, where s, $s' \in S$ and $a \in A$. The reward represents the immediate consequence of the agent's action in the given state transition.
- Transitions: The environment may be stochastic, meaning that the outcome of an action is usually not predictable. The transition function $P(s'|s, a)$, where s,

$s' \in S$ and $a \in A$, defines the probability of transitioning from state s to state s' when the agent takes action a. All the transitions are subject to the Markov property:

A state $S_t \in S$ is said to possess the Markov property if the conditional probability distribution of future states depends only on the current state. In other words, for a state S_t to have the Markov property, the following condition must hold:

$$P(S_{t+1}|S_t, S_{t-1}, \ldots, S_0) = P(S_{t+1}|S_t) \tag{2.1}$$

The agent-environment interface, therefore, can be summarized as follows: the agent starts in an initial state, follows some policy to select an action, interacts with the environment, receives a reward, and transitions to a new state. This process repeats iteratively over discrete time steps creating a so called trajectory: S_0, A_0, R_1, S_1, A_1, R_2, ...

2.1.2 Goals and Rewards

A further critical concept is the reward, which is needed to formalize the agent's objectives. At each time step, the agent receives a numerical reward from the environment. The agent's ultimate goal is to maximize the total cumulative reward over time, guided by the reward hypothesis, which suggests that all objectives and intentions can be expressed as the maximization of the expected cumulative sum of the single rewards. This approach offers flexibility and wide applicability. For instance, it's used to teach robots to walk, escape mazes quickly, or collect items. Importantly, the reward communicates what the agent's goal is, not how to achieve it. The goal is a desired condition that signifies a successful outcome or accomplishment of the task, and the rewards must align with it, as the agent will aim to maximize them.

2.1.3 Returns and Episodes

Returns represent the cumulative sum of rewards an agent receives while interacting with the environment over a sequence of time steps. The expected return, denoted as G_t, is the sum of all future rewards, each discounted by a factor γ.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.2}$$

This expected return is a central idea in reinforcement learning as it expresses the agent's objective of maximizing cumulative rewards over time. The discount factor γ

is a parameter between 0 and 1, that allows the agent to balance the importance of immediate rewards versus future rewards. A γ value of 0 indicates a focus on immediate rewards, while a γ value of 1 treats all future rewards in the same way.

Another aspect worth mentioning is the distinction between episodic and continuing tasks. An episode is a sequence of interactions between an agent and its environment, which begins at a starting state, and unfolds over a finite series of time steps. In episodic tasks, the agent-environment interaction naturally divides into episodes, each ending in a so called terminal state. For these tasks, returns are well-defined as they sum over a finite number of time steps within each episode. In contrast, in continuing tasks, the interaction continues indefinitely, and the sequence of states cannot be divided into episodes. To handle such tasks the discounting is required in order to prevent the sum from diverging.

2.1.4 Policies and Value Functions

Policies are strategies or rules that the agent employs to decide its next action in different states of the environment. Policies can be deterministic, where they directly map states to actions, or stochastic, where they specify the probability of taking every action given the current state. They are defined as

$$\pi(a|s) = \mathbb{P}\Big[A_t = a|S_t = s\Big], s \in S, a \in A \tag{2.3}$$

They are a crucial component in reinforcement learning as they guide the agent's learning process.

On the other hand, Value Functions are essential for evaluating and comparing different states or state-action pairs and critical for understanding the quality of states and actions, aiding the agent in decision-making. The two primary value functions are:

- State-Value Function: it estimates the expected cumulative reward that an agent can achieve starting from a specific state $s \in S$ and following a given policy π. Its formulation is

$$v_\pi(s) = \mathbb{E}_\pi\Big[G_t|S_t = s\Big] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] \tag{2.4}$$

- Action-Value Function: it estimates the expected cumulative reward when the agent starts in state $s \in S$, takes action $a \in A$, and follows a policy π thereafter:

$$q_\pi(s,a) = \mathbb{E}_\pi\Big[G_t|S_t = s, A_t = a\Big] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{2.5}$$

The Bellman Equation is a recursive relation satisfied by the value function. It conveys the connections between the values of states and the values of the corresponding subsequent states.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\Big[G_t|S_t = s\Big] \\ &= \mathbb{E}_\pi\Big[R_{t+1} + \gamma G_{t+1}|S_t = s\Big] \\ &= \sum_a \pi(a|s)\sum_{s'}\sum_r p(s',r|s,a)\Big[r + \gamma\,\mathbb{E}_\pi\Big[G_{t+1}|S_{t+1} = s'\Big]\Big] \\ &= \sum_a \pi(a|s)\sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right], \forall s \in S \end{aligned} \tag{2.6}$$

2.1.5 Optimal Policies and Optimal Value Functions

An Optimal Policy is the ideal strategy that an agent should follow to maximize its cumulative rewards. This policy is the one that achieves the highest expected return among all possible policies:

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s), \forall s \in S \tag{2.7}$$

The Optimal Value Function, denoted as v_*, represents the maximum expected cumulative reward achievable starting from a specific state $s \in S$ when following an optimal policy π:

$$v_*(s) = \max_\pi v_\pi(s), \forall s \in (S) \tag{2.8}$$

The Optimal Action-value Function q_* symbolizes the maximum expected cumulative reward achievable when the agent starts in state $s \in S$, takes action $a \in A$, and follows the optimal policy π thereafter.

$$q_*(s,a) = \max_\pi q_\pi(s,a), \forall s \in (S), \forall a \in (A) \tag{2.9}$$

Since v_* is the value function of a policy, it must satisfy the Bellman Equation:

$$
\begin{aligned}
v_*(s) &= \max_a q_*(s,a) \\
&= \max_a \mathbb{E}_{\pi^*}\Big[G_t|S_t = s, A_t = a\Big] \\
&= \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_*(s')\right]
\end{aligned}
\tag{2.10}
$$

The above Bellman Optimality Equation describes the relationships between optimal value functions. This equation is key for finding optimal policies and values: it expresses how the value of a state relates to the values of its successors following an optimal policy.

Optimal Policies can be found by maximizing the value function associated with that policy. Different methods, such as Dynamic Programming, Monte Carlo methods, and Temporal-Difference learning, can be employed to approximate or calculate optimal value functions and derive optimal policies.

2.2 Dynamic Programming

Dynamic Programming is a main technique in Reinforcement Learning and offers a systematic approach for solving complex decision-making problems. It provides a framework for optimizing sequential decision processes by breaking them down into smaller, more manageable sub-problems. Despite being computationally expensive, especially as the state and action spaces grow in size, it is important because it lies the foundations for many other algorithms. Anyway, the computational cost could be reduced using Asynchronous DP algorithms, which basically update the states in an asynchronous manner, i.e. some states are updated more times than others.

Dynamic Programming is designed for problems that show the property of optimal substructure, meaning that an optimal solution to the overall problem can be constructed from optimal solutions to its sub-problems. This translates to finding an optimal policy for an agent navigating an environment by recursively determining the optimal actions to take at each state, under the conditions that the environment's dynamics are known and the problem can be modeled as a Markov Decision Process.

The goal of Dynamic Programming is to solve the Bellman Optimality Equation (2.10). The task is split into two fundamental approaches: Policy Evaluation and Policy Improvement.

Policy Evaluation involves estimating the value function, representing the expected cumulative reward under a given policy. Starting from an arbitrary v_0, every next iteration is obtained by

$$\begin{aligned} v_{k+1}(s) &= \mathbb{E}_{\pi}\Big[R_{t+1} + \gamma v_k(S_{t+1}|S_t = s\Big] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right], \forall s \in S \end{aligned} \tag{2.11}$$

Policy Improvement, on the other hand, seeks to refine the current policy by selecting actions that maximize the expected cumulative reward. By acting greedy with the selection of the action a in state s, which results in the best q value, a new policy π' is computed:

$$\begin{aligned} \pi'(s) &= \arg\max_a q_{\pi}(s,a) \\ &= \arg\max_a \mathbb{E}\Big[R_{t+1} + \gamma v_{\pi}(S_{t+1})|S_t = s, A_t = a\Big] \\ &= \arg\max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_{\pi}(s')\right] \end{aligned} \tag{2.12}$$

Policy Iteration is the combination of these two approaches, it allows to build a sequence of policies and value functions that improve monotonically:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

where $\xrightarrow{E}$ denotes a policy evaluation and $\xrightarrow{I}$ denotes a policy improvement. Because a finite MDP has only a finite number of policies, this process is guaranteed to converge to an optimal policy and the optimal value function in a finite number of iterations.

Value Iteration, on the other hand, streamlines the process by merging policy evaluation and improvement into a single step:

$$\begin{aligned} v_{k+1}(s) &= \max_a \mathbb{E}\Big[R_{t+1} + \gamma v_k(S_{t+1}|S_t = s, A_t = a\Big] \\ &= \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right], \forall s \in S \end{aligned} \tag{2.13}$$

The computational efficiency of Value Iteration becomes evident, presenting advantages over Policy Iteration in certain contexts.

Going further, Generalized Policy Iteration is a framework where the above algorithms can be unified (Figure 2.2). GPI presents policy evaluation and improvement as interconnected processes along a continuum, allowing them to interact with each other.

GPI stabilizes when both the evaluation and improvement processes stop producing changes: optimality has been reached. The stability of the value function relies on its consistency with the current policy, while policy stability hinges on its greediness concerning the current value function.

The interaction between evaluation and improvement in GPI can be conceptualized as two constraints or goals in a two-dimensional space, where the joint process converges towards optimality. Figure 2.3 illustrates how, in GPI, smaller, incomplete steps toward each goal collectively lead to the achievement of the overall goal of optimality, even when neither process explicitly aims for it.

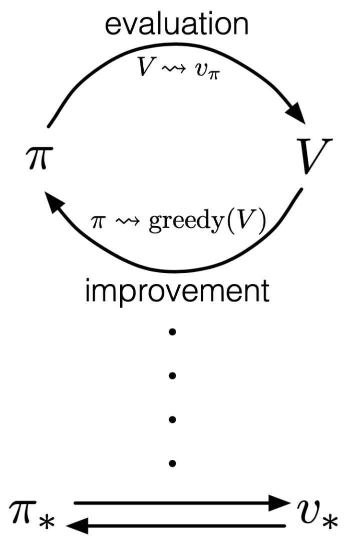

evaluation
$V \rightsquigarrow v_\pi$
π
V
$\pi \rightsquigarrow \mathrm{greedy}(V)$
improvement
π_*
v_*

Figure 2.2: Generalized Policy Iteration loop

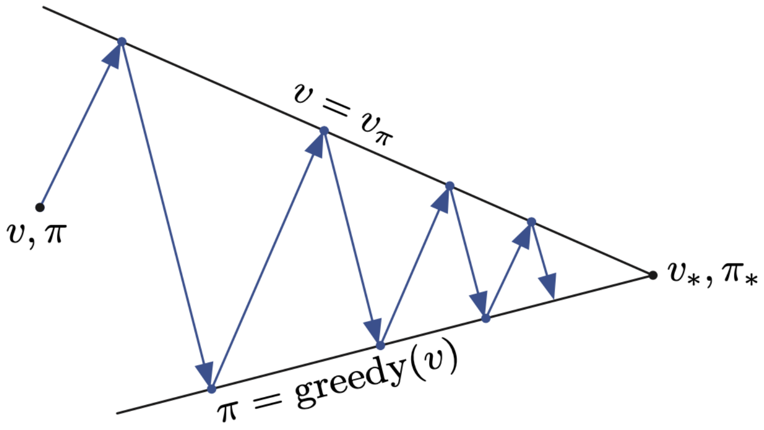


Figure 2.3: Alternative representation of GPI

2.3 Monte Carlo Methods

Monte Carlo methods are a class of algorithms widely used in Reinforcement Learning to determine optimal policies. They are particularly useful when an agent does not have complete knowledge of the environment's dynamics, making them applicable to a broader range of scenarios.
When dealing with predictions, unlike Dynamic Programming methods, Monte Carlo methods operate directly on sample sequences of states, actions, and rewards obtained through interaction with the environment. These methods operate exclusively on episodic tasks, updating value function and policy estimates only at the end of an episode. In addition, while estimating the action-value function $q_\pi(s,a)$, Monte Carlo

utilizes the concept of visits to state-action pairs: a pair is considered visited if action a was chosen in state s.

Monte Carlo methods are split into two: prediction and control, which address different aspects of the learning process.

Monte Carlo Prediction employs methods such as the First-Visit and Every-Visit Methods to estimate value functions, while Monte Carlo Control focuses on finding optimal policies through the iterative process of policy evaluation and improvement, considering the challenge of exploring starts, that is, all episodes begin with state-action pairs randomly selected to cover all possibilities.

Monte Carlo Prediction aims to estimate the value function $v_\pi(s)$, i.e. the expected cumulative future discounted reward for a state s under policy π. In the First-Visit Method, a state s is considered visited when it appears for the first time in an episode. The estimation process involves tracking returns obtained after the first visit to s and averaging these returns over multiple episodes. As the number of visits increases, the average converges to the expected return.

Similar to the First-Visit Method, also the Every-Visit Method seeks to estimate the value function $v_\pi(s)$, but it takes into account all visits to a state, not just the first one. For each state s, returns obtained after every visit are tracked, and these returns are averaged over multiple episodes. Therefore, in this method the estimates are updated more frequently.

In the control problem, Monte Carlo methods draw from the idea of Generalized Policy Iteration. The modified policy iteration consists in observing an infinite number of episodes, starting with exploring starts, and alternating between evaluation and improvement on an episode-by-episode basis. The goal is finding an optimal policy π_* that maximizes the expected cumulative future discounted reward.

Through Policy Evaluation the value function $q_\pi(s, a)$, representing the expected cumulative future discounted reward for taking action a in state s under policy π, is estimated. The policy is then improved by selecting the action that maximizes the action-value function in a greedy manner:

$$\pi(s) = \arg\max_a q_\pi(s, a) \tag{2.14}$$

As mentioned above, an inherent challenge in traditional Monte Carlo methods is the assumption of exploring starts, where each episode begins with randomly selected state-action pairs to cover all possibilities. To address this, exploration is ensured by starting each episode with a random state-action pair, avoiding deterministic behavior. Alternatively, in Generalized Policy Iteration, exploration is maintained by alternating between policy evaluation and improvement on an episode-by-episode basis.

In order to remove the assumption of exploring starts, the introduction of on-policy and off-policy methods is necessary. On-policy methods aim to evaluate or improve the policy used for decision-making, shifting gradually from a soft to a deterministic optimal policy, while the off-policy approach evaluates a policy different from the one used to generate the data.

Importance sampling is a prevalent technique in all these methods, where returns are weighted by the ratio of the probabilities of taking observed actions under the two policies, allowing for more effective learning and exploration.

2.4 Temporal-Difference Learning

Temporal-Difference learning, also known as TD-Learning, represents a category of algorithms that combines elements of both Monte Carlo and Dynamic Programming methods. Similar to Monte Carlo, TD-learning learns directly from experience without the need of a model of the environment. Along Dynamic Programming, it updates the value function using already found estimates, employing a technique known as bootstrapping. This characteristic allows TD methods to be applicable to continuing tasks, enabling the updating of state values as soon as new information becomes available, without waiting for the completion of an episode.
TD learning, as Monte Carlo, can be divided into two main problems: prediction and control.

TD prediction methods aim to estimate value functions based on incomplete sequences of experiences, which means that value estimates are updated incrementally: instead of relying on the full return as in Monte Carlo methods, TD prediction uses the expected cumulative future discounted reward after a single time step. This expectation can be estimated by sampling episodes according to a given policy π. The algorithm updates the value function using a learning rate parameter α, controlling the influence of the new estimate compared to the previous one. The TD error, δ, quantifies the difference between the estimated value of the current state $V(S_t)$ and the better estimate $R_t + V(S_{t+1})$ obtained from the sampled episode.
While this update introduces bias in the estimate, it reduces variance, enabling TD methods to learn faster than Monte Carlo. The trade-off between bias and variance is influenced by the choice of the learning rate and plays a crucial role in the effectiveness of the algorithm.

In TD control, the goal is to estimate the action-value function $q_\pi(s, a)$ rather than a state-value function. Three main methods are commonly used: Sarsa, Expected Sarsa, and Q-learning.

- Sarsa (State-Action-Reward-State-Action): In Sarsa, the algorithm selects an action A_t in state S_t, observes the reward R_{t+1} and the next state S_{t+1}, and then selects the next action A_{t+1}. The value of the next state, $Q(S_{t+1}, A_{t+1})$, is a sample of $V(S_{t+1})$ since A_{t+1} is chosen according to policy π.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1,}A_{t+1}) - Q(S_t, A_t)\Big] \tag{2.15}$$

- Q-learning: It assumes that the next action A_{t+1} is chosen according to a greedy policy with respect to the action-value function $Q(S_{t+1}, A_{t+1})$. The learned action-value function approximates the optimal action-value function q_*, independently of the policy being followed. However, the behavioral policy π still influences which state-action pairs are visited.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big] \tag{2.16}$$

- Expected Sarsa: This method, a variant of the Sarsa algorithm, estimates the value of the next state by directly using its expected value $V(S_{t+1})$. This approach reduces the variance of the value estimate but increases computational cost.

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1)} - Q(S_t, A_t)\Big] \\ &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a Q(S_{t+1}, a)\pi(a|S_{t+1}) - Q(S_t, A_t)\Big] \end{aligned} \tag{2.17}$$

2.5 n-step Bootstrapping

n-step methods represent a class of algorithms that combine features of both Monte Carlo and Temporal-Difference learning. These methods are designed to estimate value functions and learn optimal policies through direct experience without requiring a complete model of the environment.

The primary focus of n-step methods is the prediction problem, which involves estimating the value function of a policy. Unlike single-step TD methods, which use only the immediate reward and the estimated value of the next state, n-step methods consider a sequence of rewards and states over multiple time steps. This sequence goes from the current time step up to n future time steps. Hence, their adaptability to different time horizons makes them valuable tools for learning in environments with complex, temporally extended dynamics.

The n-step return is defined as the sum of rewards obtained over the next n time steps, appropriately discounted:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) \tag{2.18}$$

The relationship between n-step returns and TD returns is established by varying the value of n.
In the case of n-step TD, where n is 1, the algorithm uses a one-step return. In contrast, for n greater than 1, the algorithm utilizes a multi-step return, allowing it to capture longer-term dependencies in the learning process. The estimate of the value function is updated based on the observed sequence of rewards and states:

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha \Big[G_{t:t+n} - V_{t+n-1}(S_t) \Big], \quad 0 \leq t < T \tag{2.19}$$

where T is the episode length. If n goes beyond T, then the n-step return is equal to the full return, i.e. Monte Carlo prediction.

The process of estimating the value function involves iteratively updating the value of a state, considering the n-step returns. This iterative update is guided by the learning rate parameter α, which determines the weight given to the new estimate compared to the existing estimate. The learning rate controls the rate at which the algorithm adapts to new information.

The n-step return introduces a trade-off between bias and variance. Larger values of n may reduce variance by providing a more accurate estimate of the true value, but this comes at the cost of increasing the bias. On the contrary, smaller values of n may have lower bias but higher variance: the choice of the appropriate n clearly depends on the specific characteristics of the environment and the learning task.

2.6 Multi-Agent Reinforcement Learning

Multi-Agent Reinforcement Learning (MARL) addresses scenarios involving multiple interacting agents within a shared environment. Unlike traditional RL, where a single agent learns to navigate an environment, each agents wants to develop a policy that maximizes its expected return over time. This multidimensional approach takes into account both environmental dynamics and the actions and behaviors of other learning agents.

In MARL, agents operate concurrently within a shared environment (Figure 2.4), and the actions of one agent have the potential to influence the observations and rewards of other agents. Therefore, each player must consider not only their own interaction with the environment, but also the actions of other players, making the learning more difficult.

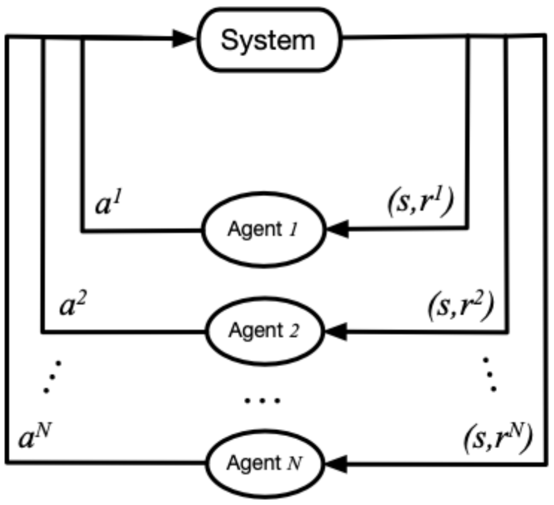


Figure 2.4: An example of the Multi-Agent setting

The interaction nature of agents can be either cooperative or competitive (Zhang, Yang, and Başar 2021): they might collaborate towards a common goal or compete for limited resources. In the former, the problem can be simplified to a single-agent framework, by summing individual rewards and considering the cartesian product of individual action spaces.
Communication may be allowed between agents in some scenarios, facilitating information sharing and improved coordination. However, designing effective communication protocols and ensuring optimal learning of communication strategies are complex tasks.

Each agent typically has access only to its private observations, providing a partial view of the overall state which puts together all the agent states. Observations may include information about the positions, actions, or policies of other agents, depending on the level of information sharing.
The joint action space represents the combination of actions selected by all agents. The choice of one agent's action has repercussions on the opportunities and constraints for other agents, introducing coordination and competition challenges. Hence, the design

of a reward structure becomes imperative in order to find a balance between individual and collective objectives.

MARL introduces challenges beyond single-agent RL, including the curse of dimensionality, particularly as the number of agents increases, the complexity of specifying suitable goals due to correlated agent returns, and setting rewards.
The exploration-exploitation dilemma is intensified, as agents must explore not only the environment but also other agents, trying to balance the acquisition of knowledge and the potential destabilization created by learning the counterparts.

Chapter 3

Offline Reinforcement Learning

Offline reinforcement learning represents a paradigm within the broader field of RL where an agent learns from a pre-collected dataset of experiences rather than engaging in real-time interactions with the environment. This approach is in contrast to classic online RL, where the agent learns by actively exploring and exploiting the environment in a sequence fashion. The motivation behind offline RL emerges in scenarios where the cost, risk, or impracticability of real-time interaction make such interaction too challenging or undesirable (Levine et al. 2020). Indeed, offline RL uses to its advantage a fixed dataset, including past experience, to train and improve the agent's policies or value functions.

Offline RL, while promising, requires careful consideration of the challenges related to using a fixed dataset and adapting to changes in the environment: this is essential to ensure the success and robustness of offline RL approaches.
As research in RL continues to evolve, offline RL remains a crucial avenue for exploring the potential of historical data for efficient and effective learning in dynamic and complex environments (Nie et al. 2023).

3.1 Implementation

The offline RL process is carried out in several key stages. First, a dataset is generated through interactions with the environment based on a specific policy. This policy could be randomly chosen, heuristic, or derived from some initial exploration strategy (Schweighofer et al. 2022). The collected dataset encloses trajectories consisting of states, actions, rewards, and subsequent states; it must include a variety of scenarios, ensuring the model encounters a broad enough range of states and actions.

The dataset represents a snapshot of the environment based on the interactions with the initial policy and, once built, it is stored for the training phase. Therefore, unlike online RL, which learns by continuously updating the data, offline RL learns from a static dataset.
Training in offline RL involves using the stored dataset to update the agent's policies or value functions. Various RL algorithms can be adapted for batch learning, such as different forms of Q-learning, actor-critic methods, and value iteration (Levine et al. 2020). Indeed, the learning phase is usually conducted in a batch fashion: eventually the entire dataset is processed and the agent's understanding of the environment has hopefully improved.

Throughout the training process, continuous monitoring and validating against an evaluation dataset or the environment are critical to ensure the model's generalization to unseen scenarios and its alignment with the desired policy. This evaluation step helps assess the effectiveness of the learned policy or value function.

3.2 Algorithms

In terms of algorithmic choices, selecting an appropriate offline RL algorithm is pivotal, as each comes with its strengths and weaknesses.

Popular algorithms in offline RL assume that the dataset contains valuable information for training without requiring further interactions with the environment. Among these algorithms, Constrained Offline Policy Optimization (COPO) stands out as an influential method (Polosky et al. 2022). COPO leverages constrained policy optimization to ensure stability during training and effective utilization of the collected data. Another notable algorithm is Conservative Q-Learning, which introduces a conservative estimator to improve sample efficiency and enhance robustness in learning from offline data (Kumar et al. 2020).

Behavior regularization, on the other hand, tries to keep the learned policy faithful to the agent's conduct observed in the offline dataset (Wu, Tucker, and Nachum 2019; Hongchang et al. 2022). To address this, behavior regularization methods introduce mechanisms during the training to push the learned policy to mimic the behavior present in the dataset. These mechanisms often take the form of additional terms or constraints in the learning process, guiding the optimization towards policies that align with the actions taken in the offline data. In other words, the objective is to enhance the generalization of learned policies to real-world scenarios by constraining their behavior to be consistent with the offline dataset.

Once the algorithm is chosen, fine-tuning hyperparameters and conducting rigorous evaluations on the dataset become essential to optimize the model's performance.

3.3 Advantages and Challenges

Offline RL brings forth several advantages and challenges. On the positive side, it often proves to be more data-efficient than online methods since it makes functioning use of the collected samples without requiring real-time exploration. This characteristic makes offline RL particularly appealing for scenarios where collecting data is expensive, time-consuming, or may involve risks. By leveraging historical data, offline RL enables efficient utilization of past experiences, potentially accelerating the learning process. Moreover, it allows for policy improvement without additional exploration, making it suitable for applications with safety constraints (Levine et al. 2020).

One major difficulty is distributional shift: it consists in changes in the entire distribution of the data, including both the input distribution and the target distribution, rewards or returns. Hence, the model trained offline may struggle to generalize to situations not well-represented in the dataset, leading to sub-optimal performance in real-world scenarios. This challenge is worsen by the fact that the data collection policy might not be optimal, and the dataset may lack diversity or fail to cover critical edge cases.
To combat distributional shift, importance sampling is often employed. This technique assigns weights to samples based on the dissimilarity between the data collection policy and the target policy, helping the model to correct for biases introduced by the dataset (Z.-W. Hong et al. 2023). Advancements in offline RL include other distributional correction approaches, which aim to align the learned distribution with the true distribution of returns (Wu, Tucker, and Nachum 2019).

The lack of exploration during training poses another challenge, as the agent must rely exclusively on the information available in the fixed dataset. This can slow down the discovery of novel strategies or adapting to changes in the environment.
Researchers explore techniques like counterfactual data augmentation, a powerful tool where the dataset is artificially expanded by generating hypothetical trajectories that might have occurred but were not observed (Pitis et al. 2022). This augmentation helps expose the model to a more diverse set of states and actions, resulting in a better generalization. Meta-learning strategies are also investigated, enabling the agent to adapt quickly to new tasks or environments by leveraging knowledge from previous experiences (Mitchell et al. 2021).

Covariate shift, or non-stationarity, emerges from learning a policy that can make arbitrary mistakes in parts of the state space not covered by the expert dataset, which

is a consequence of a change in the input distribution, meaning the distribution of the states or observations encountered during training differs from those encountered during deployment (Chang et al. 2022). This can also impact the generalization of the learned model, making it challenging for the learned policy to adapt effectively. Ensemble methods, where multiple models are trained on different subsets of the dataset, provide a form of robustness against shifts in the underlying dynamics (Kidambi et al. 2021). These ensembles can capture diverse aspects of the environment, enabling the agent to adapt more effectively to changes.
Another approach involves incorporating uncertainty estimates into the learning process, allowing the model to express uncertainty in regions where the data is sparse or non-representative (J. Hong, Kumar, and Levine 2023).

3.4 Policy improvement

The effectiveness of an agent trained with offline reinforcement learning to learn a better policy than the one used to generate the dataset depends on various factors, and it is not guaranteed (Nie et al. 2023).
Several considerations influence whether the trained agent can surpass the policy used for data collection:

- Dataset Quality: The quality of the dataset is crucial. If the dataset provides diverse and informative experiences that cover a wide range of scenarios and challenges, it increases the likelihood of the agent to learn a better policy.

- Exploration Strategy: The exploration strategy employed during data collection is significant. If the initial policy used for exploration is too conservative or suboptimal, the agent might struggle to surpass it. A more effective exploration strategy often leads to a more informative dataset.

- Algorithm Choice: The choice of the RL algorithm for offline training plays a vital role. Some algorithms may be better suited for exploiting the information in the dataset and generalizing to unseen situations. Some techniques mentioned above, like distributional correction and importance sampling, can indeed impact the learning process.

- Model Complexity: The complexity of the model used by the agent can influence its capacity to learn a better policy. A well-designed and appropriately complex model may capture subtle patterns and dependencies in the data, enabling increased performance.

- Environment Dynamics: Changes in the environment or task requirements between data collection and training can pose challenges. If the environment remains relatively stable, the trained agent has a better chance of outperforming the original policy.
- Adaptability: Some offline RL algorithms are designed to handle distributional shifts and changes in the environment. Ensuring that the chosen algorithm has mechanisms for adaptation can positively impact the agent's ability to learn a superior policy.

Despite these considerations, there are no assurances that the learned policy will perform well in scenarios similar to those in the dataset and significant deviations might still be observed.

Chapter 4

Decision Transformer

In recent years, transformers have revolutionized the field of Natural Language Processing and have become the center of many state-of-the-art models, such as BERT (Devlin et al. 2019) and GPT (Radford et al. 2018). These models are pre-trained on a large corpus of text using unsupervised or semi-supervised learning and are fine-tuned afterwards on task-specific datasets, making them applicable to multiple NLP domains.

Transformers were originally proposed for sequence-to-sequence tasks, but they managed to find applications also in image recognition, reinforcement learning, and time-series forecasting (Lin et al. 2021). As a matter of fact, the transformer's ability to capture contextual information makes it suitable for tasks requiring an understanding of the relationships present inside the data.
Among the transformer variants, the Decision Transformer stands out as a powerful architecture designed specifically for sequential decision-making tasks (Chen et al. 2021).

In this chapter, a comprehensive view of the Decision Transformer functioning is presented, together with some fundamental concepts from Machine Learning, Deep Learning, and transformer architectures.

4.1 Natural Language Processing

Natural Language Processing is found at the intersection of computer science, artificial intelligence, and linguistics, with the goal of enabling machines to understand, interpret, and generate human language (Beslin Pajila et al. 2023). NLP embraces a broad range of tasks, from traditional problems, like language translation and speech recognition, to more advanced challenges such as sentiment analysis, question-answering, and language generation.

The primary objective of NLP is to bridge the gap between human communication and computer understanding, allowing machines to interact with users in a natural and meaningful way.

Different deep learning architectures are used in NLP, including recurrent neural networks, convolutional neural networks, and transformer-based models..

4.1.1 Embedding

A fundamental parameter in NLP is the embedding dimension, usually used in machine learning tasks that involve processing categorical data. It represents the size of the continuous vector space in which categorical inputs like words, tokens, or items, are represented as dense vectors, whose size is exactly the embedding dimension.
These dense representations are employed to give semantic and contextual information about the words which are being analysed.

The choice of the embedding dimension is a hyperparameter that depends on the specific task and the size of the dataset (Torfi et al. 2021). Larger embedding dimensions can capture more detailed information but require more memory and computational resources. On the other hand, smaller dimensions may result in a loss of information but are more memory-efficient.

4.2 Neural Networks

Neural networks, inspired by the biological neural networks of the human brain, consist of interconnected nodes, or neurons, organized in layers. Each neuron receives input signals, processes them using internal parameters, and produces an output signal that is passed on to other neurons in the network. Neural networks can learn to map input data to desired outputs, being able to solve a wide range of tasks.

The power of neural networks lies in their ability to automatically discover patterns and relationships in complex data, without being explicitly programmed (Schmidhuber 2015). This is achieved through iterative optimization algorithms, such as gradient descent, which adjust the network's parameters to minimize the difference between the prediction and the actual outputs.

Many neural network architectures have been developed to address specific problems and requirements, from feed-forward networks to recurrent neural networks and convolutional neural networks. Each of these architectures offers unique advantages for processing different types of data and capturing different types of patterns.

4.2.1 Feed-forward Neural Networks

Feed-forward neural networks, also known as multilayer perceptrons, form the foundational architecture of deep learning. They are made of layers of neurons, where each neuron of a layer is connected to every neuron in the subsequent layer.
In FNNs, information flows in one direction, from the input layer through one or more hidden layers to the output layer, without any cycles or loops (Figure 4.1).

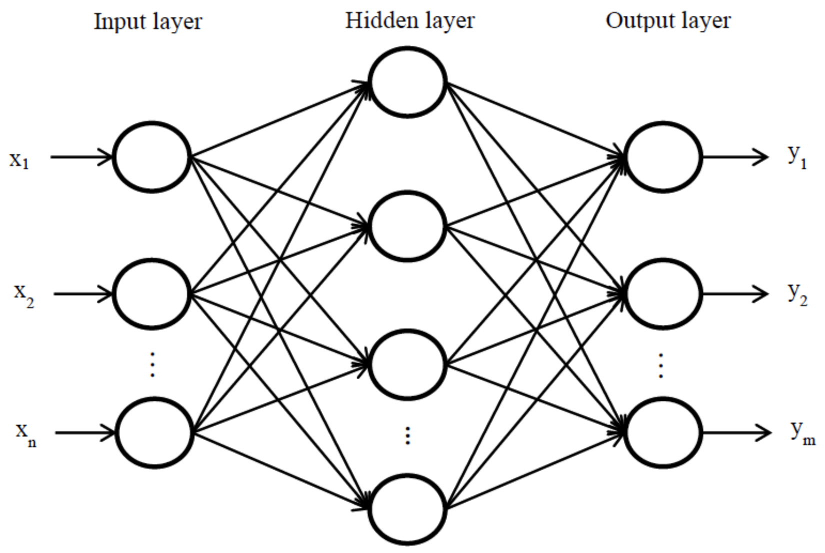


Figure 4.1: A multi-layer feed-forward neural network

Each neuron in an FNN performs a weighted sum of its inputs and adds a bias, followed by the application of an activation function to produce the neuron's output (Sazli 2006):

$$output = f\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{4.1}$$

where w_i represents the weight associated with the input x_i, f is the activation function, b is the bias term, and n is the number of inputs to the neuron.

This output then becomes the input to the neurons in the next layer and the process repeats throughout the network until the final output is generated. The weights and biases associated with each neuron are learned during the training process using techniques like backpropagation or gradient descent.

FNNs are widely used for tasks such as classification, regression, and function approximation. However, they may struggle with tasks involving sequential or temporal data, as they lack mechanisms to capture dependencies over time.

In spite of their simplicity compared to other more complex architectures like recurrent neural networks, FNNs are still an important tool in the deep learning framework thanks to their scalability and effectiveness in many real-world applications. Moreover, advancements such as the introduction of deeper architectures with multiple hidden layers, regularization techniques, and activation functions have further enhanced the performance and capabilities of this kind of neural networks.

4.2.2 Recurrent Neural Networks

Recurrent Neural Networks are a class of artificial neural networks specifically designed to handle sequential data, making them well-suited for tasks involving variable-length sequences and where understanding the order or timing of events is crucial. Unlike the feed-forward neural networks, RNNs are characterized by the ability to learn temporal dependencies by adding cycles that allow information to be transmitted back into itself (Figure 4.2) (Schmidt 2019).

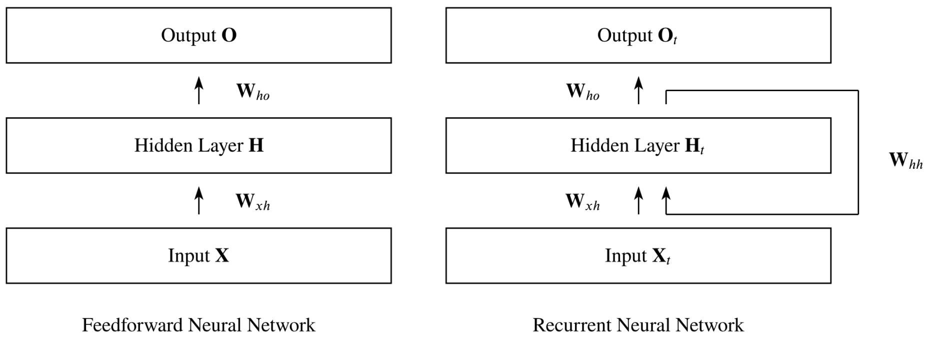


Figure 4.2: Visualisation of differences between Feedfoward NNs und Recurrent NNs

In the context of reinforcement learning, RNNs perform well in scenarios with partially observable environments, where past observations and actions influence the current state and decision-making (Schäfer 2008). The recurrent connections in RNNs enable them to maintain an internal memory, effectively encoding information about the agent's past experiences. This memory is fundamental for learning policies that consider the entire trajectory of an RL episode.

Despite their strengths, traditional RNNs suffer from challenges like vanishing or exploding gradients. Advanced variants, such as Long Short-Term Memory networks (Sherstinsky 2020), address these issues incorporating mechanisms to better gather information over extended sequences. Moreover, RNNs usually fail to remember long-range dependencies: they need to encode the information from the entire sequence in one single context vector and as the gap between two words grows, the model seems to "forget" the previous words.

4.2.3 Convolutional Neural Networks

Convolutional Neural Networks are another kind of networks employed for processing structured grid-like data, such as images and videos. They are particularly effective in computer vision tasks, where the input data has spatial relationships that are crucial for understanding the content. CNNs consist of multiple layers, including convolutional layers, pooling layers, and fully connected layers (Figure 4.3) (O'Shea and Nash 2015).

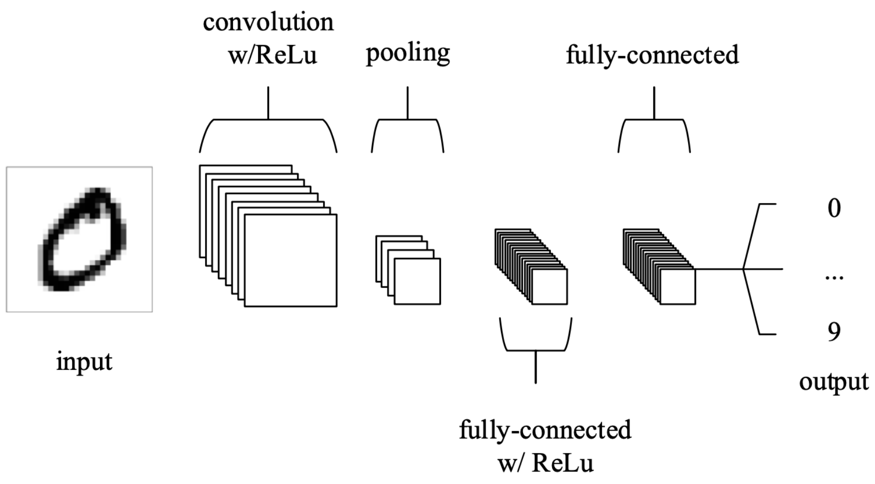


Figure 4.3: A simple CNN architecture, containing five layers

Convolutional layers are at the core of CNN. They apply convolution operations to the input data (Figure 4.4): these operations involve sliding small filters, or kernels, across the input data, performing element-wise multiplication with the overlapping regions, and then summing the results to produce the feature maps. By learning the parameters of these filters during training, CNNs can automatically extract hierarchical representations of the input data, obtaining features at different levels of abstraction.

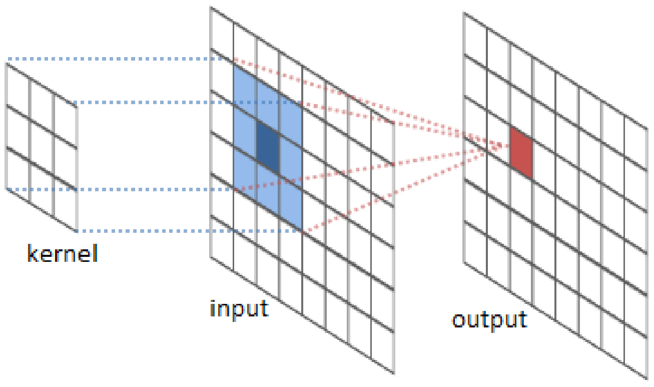


Figure 4.4: Convolution operation

Indeed, CNNs are known for their effectiveness in capturing local patterns and can be stacked to create deeper architectures that learn increasingly complex features.

On the other hand, the role of pooling layers is to reduce the spatial dimensions of the feature maps while preserving the most important information. Common pooling operations include max pooling and average pooling, which down sample the feature maps by taking respectively the maximum or average value within each pooling window. This helps reducing the computational complexity of the network and making it more robust to fluctuations in the input data.

Finally, fully connected layers are used to perform high-level reasoning and decision-making based on the extracted features. These layers connect every neuron in one layer to every neuron in the next layer, allowing the network to learn complex relationships among the extracted features and make predictions about the input data.

As mentioned above, this type of architecture developed for computer vision tasks, but it has found applications also in NLP (Wang and Gang 2018), particularly for tasks with fixed-length inputs such as text classification and sentiment analysis. In NLP, CNNs operate by applying convolutional filters over input sequences, extracting local features similar to how they capture spatial features in images. These filters highlight patterns at different levels of granularity, from individual characters or words to higher-level features representing phrases or sentences.

4.3 Transformers

A transformer is a deep learning architecture based on the multi-head attention mechanism. It has become a fundamental tool in the field of natural language processing, especially in tasks requiring contextual understanding. Indeed, this architecture allows to process sequential data and capture long-range dependencies.
In addition, unlike recurrent or convolutional neural network structures, transformers allow parallel training as their process is carried out on the whole input sequence simultaneously and not sequentially or locally: they are highly scalable and efficient (Lin et al. 2021).

4.3.1 Positional encoding

In the transformer architecture, since there are no recurrences or convolutions, positional encodings are added to the input embeddings to provide information about the positions of words in the sequence. These positional encodings enable the model to distinguish between words based on their positions, allowing it to understand the sequential order of the input data (Ghojogh and Ghodsi 2020).

One way to generate positional embeddings is using sinusoidal functions (Vaswani et al. 2017). This idea stands on the hypothesis that representing positions as sinusoids can provide the model with meaningful positional information in a continuous and regular manner.
The positional embedding for each position is computed using sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{4.2}$$

$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{4.3}$$

where:

- $PE_{(pos,2i)}$ and $PE_{(pos,2i+1)}$ represent the ith and $(i+1)$th dimensions of the positional embedding vector for position pos,
- pos is the position in the sequence,
- i is the dimension of the positional embedding,
- d_{model} is the dimensionality of the model.

4.3.2 Self-attention

The role of self-attention layers is to compute attention scores between all pairs of positions in the input sequence. Attention scores are nothing but an indication of how important is the corresponding element in the sequence (Ghojogh and Ghodsi 2020). Indeed, the attention mechanism eliminates the need to encode the full source sentence into a fixed-length vector as in RNNs. Rather, the decoder is allowed to focus on the most influential parts of the source sentence at each step of the output generation. Importantly, the model is the one deciding what to attend to based on the input sentence and what it has produced so far.

Each attention layer employs learned linear projections to achieve dimensionality reduction, thereby mitigating computational complexity and it incorporates the positional encodings to provide information about the relative or absolute position of tokens in the input sequence.

The attention mechanism operates through a set of queries, keys, and values. For each element in the input sequence, a query is used to weigh the relevance of all other elements (keys) in the sequence, and the weighted sum of values is computed. This process is repeated for every element, creating an attention distribution (Vaswani et al. 2017).

The multi-head self-attention mechanism consists of several heads working in parallel, where each head operates as a separate attention mechanism. The following is an overview of what happens in a single self-attention head:

- Query, Key, and Value Vectors: The input to the self-attention mechanism consists of three sets of vectors: query Q, key K, and value V vectors. These vectors are computed from the input data using learned linear transformations.

- Attention Scores: Each head computes attention scores by taking the dot product between the query vector of a position and the key vector of all other positions in the input sequence. These scores indicate how much attention should be paid to each position in the sequence when processing the current position.

- Attention Weights: The attention scores are scaled and passed through a softmax function to obtain attention weights. These weights determine the importance of each position in the input sequence for the current position, with higher weights indicating higher importance.

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right) \tag{4.4}$$

where d_K is the dimension of K.

- Weighted Sum: The attention weights are used to compute a weighted sum of the value vectors. This weighted sum represents the output of the self-attention mechanism for that head.

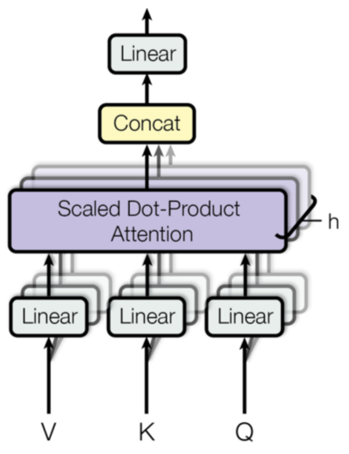


Figure 4.5: Multi-Head Attention

This process is repeated for each head, with each head having its own sets of query, key, and value vectors. The outputs from all the heads are concatenated and transformed again through another linear layer to produce the final self-attention output (Figure 4.5).

The idea behind having multiple heads is that different heads may focus on different aspects or patterns in the input data simultaneously and this allows the model to capture a more diverse and rich set of relationships between elements in the sequence.

4.3.3 Encoder and Decoder

The transformer is composed of two macro parts: encoder and decoder (Figure 4.6). These components are responsible for processing input sequences and generating output sequences respectively.

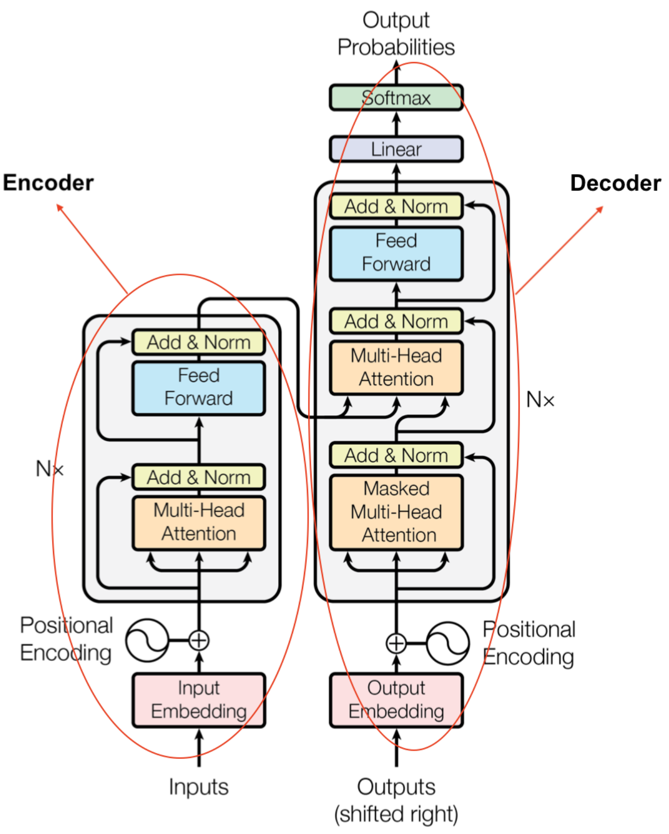


Figure 4.6: Model architecture of a transformer

In the encoder, each position in the input sequence attends to all other positions to capture the relationships and dependencies between different parts of the input sequence. Through the self-attention mechanism it generates a context-aware representation of the input sequence, by linearly combining attention weights and positional encodings and concatenating the context vector with the output of the previous time step (Vaswani et al. 2017).

In the decoder each position in the generated output sequence attends to all positions in the input sequence. This mechanism enables the decoder to focus only on the relevant parts of the input sequence when generating each element of the output sequence, which will be consequently aligned with the relevant information in the input.

Generally, the transformer architecture can be used in three different ways (Lin et al. 2021):

- Encoder-Decoder: The full architecture is used, which is typically done in sequence-to-sequence modeling.
- Encoder only: Only the encoder is employed and its outputs are utilized as a representation for the input sequence. This is usually used for classification or sequence labeling problems.
- Decoder only: Just the decoder is included and the encoder-decoder cross-attention module is removed, typically used for sequence generation, such as language modeling.

4.3.4 LayerNorm

LayerNorm is a technique used in neural networks to normalize the activations of neurons within each layer of the network. It is applied to ensure that the mean activation of each neuron is close to zero and that the standard deviation is close to one across the training items in a batch, in order to provide the subsequent layer with standardized inputs. This process helps stabilizing the training by reducing the internal covariate shift problem, which consists in a shift of the activations distribution during training, making it harder to optimize the network (Ba, Kiros, and Hinton 2016).
In the context of transformers, LayerNorm is applied after each self-attention or feed-forward network in both the encoder and decoder components (Vaswani et al. 2017).

The mathematical formulation of layer normalization is as follows:

$$LayerNorm(x) = \gamma \frac{x - \mu}{\sigma} + \beta \tag{4.5}$$

where:

- x is the input tensor to be normalized,

- μ and σ are the mean and the standard deviation of the input tensor along the normalization axis,
- γ and β are the learnable scale and bias parameters that allow the model to respectively scale and shift the normalized activations.

4.3.5 Transformer Block

All the single elements of the transformer architecture discussed above are then assembled to form the transformer block (Turner 2024).
The block itself comprises two stages: one operating across the sequence and one operating across the features. The first stage improves each feature independently taking into account the relationships between tokens across the sequence i.e. how much a word in a sequence at position n depends on previous words at position n'. Instead, the second stage refines the features representing each token. By repeatedly applying the transformer block the representations of token n and feature d get shaped through the information recovered from token n' and feature d'.

The structure of a block unfolds as follows:

- Multi-Head Self-Attention Layer: This layer computes attention scores between all pairs of words in an input sequence. The outputs from all the heads are concatenated and transformed through a linear layer to produce the final self-attention output.
- Feed-forward Neural Network: After the self-attention layer, the transformer block applies a feed-forward neural network to the self-attention output. This layer consists of two linear transformations with a non-linear activation function, such as ReLU, in between.
- Residual Connections: To facilitate training deep transformer architectures, residual connections are added around both self-attention and feed-forward layers. These connections allow gradients to flow more easily during training, mitigating the vanishing gradient problem.
- Layer Normalization: Layer normalization is applied after each sub-layer, as the residual connection, to stabilize the training process and improve convergence.

Both the encoder and decoder blocks share the above structure (Figure 4.6), with the addition in the latter of a third sub-layer to perform multi-head attention over the encoder's output. Moreover, the attention mechanism of the decoder is modified to prevent positions from attending to subsequent positions. This modification ensures that the model only attends to previously generated elements during decoding, preserving its autoregressive property.

4.3.6 Reinforcement Learning apllication

Traditional RL methodologies encounter difficulties when dealing with partially observable environments or complex sequential decision-making tasks, reason why transformers have been widely employed in RL to develop more sophisticated agents capable of handling sequential and structured data (W. Li et al. 2023). Transformers manage to modify the representation and the processing of information, treating RL problems as sequence-to-sequence tasks. In addition, thanks to the self-attention mechanism, the model can focus on relevant elements in the input sequence, discerning temporal dependencies and understanding the context of an RL environment.

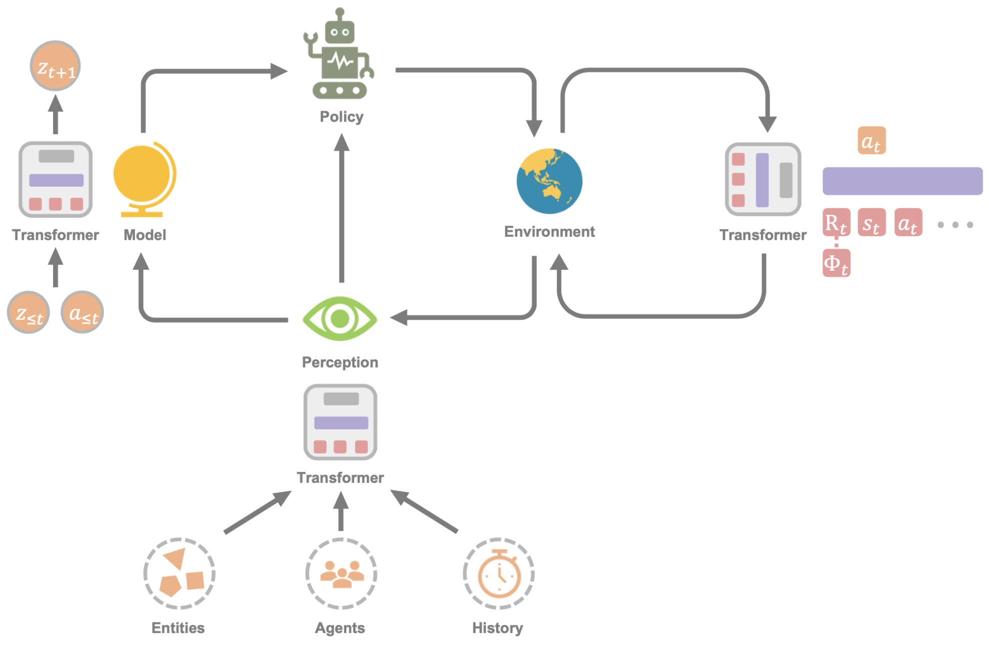


Figure 4.7: An illustrating example of a transformer in the RL setting

4.4 Decision Transformer

The Decision Transformer is an architecture that casts the problems of reinforcement learning as conditional sequence modeling. What it does is simply output the optimal actions by leveraging a causally masked transformer and, by conditioning an autoregressive model on the desired return, past states, and actions, it can generate future actions that achieve the desired return (Figure 4.8). Moreover, it incorporates action embeddings and multi-head attention mechanisms, which enable it to model dependencies between actions and states efficiently.

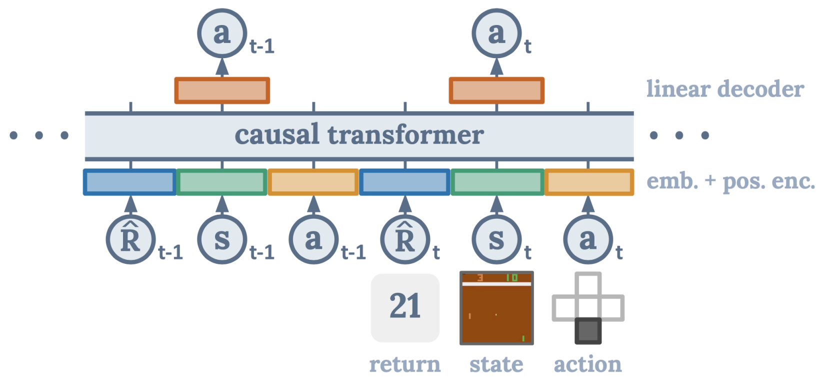


Figure 4.8: Decision Transformer architecture

The Generative Pre-trained Transformer is a transformer-based model characterized by a stack of transformer layers, which enable it to capture relationships and dependencies in texts (Radford et al. 2018). It modifies the original transformer architecture with a causal self-attention mask to enable autoregressive generation, replacing the softmax over the full sequence of tokens with only the previous tokens in the sequence. This is done as the primary focus of GPT is generative tasks, i.e. predicting the next word in a sequence given the context of previous words, and the Decision Transformer relies on its architecture.

At the core of the Decision Transformer there is a decoder only architecture. It processes the input sequence of states, actions, and rewards and predicts future actions given both current and historical information.

Token embeddings of states, actions and rewards are obtained by learning a linear layer, which projects raw inputs to the embedding dimension, followed by layer normalization. The same strategy is adopted to embed the time steps, generating the positional embedding that lets the Decision Transformer effectively capture the sequential relationships and dependencies in the input data, enabling it to make informed decisions based on both the token identities and their positions in the sequence.

The sequence has to be represented in such a way to enable the transformer to learn meaningful patterns and allow to conditionally generate actions at test time. The model should generate actions based on future desired returns, rather than past rewards. Therefore, instead of supplying the action rewards directly, the model is fed with the returns-to-go, i.e. the sum of the future rewards. At test time the desired performance, which is the total return that the agent is expected to gain throughout the episode, can be specified as the first token of the trajectory and, as the episode evolves, this target return is decreased by the action rewards. In this way, during the game the agent is constantly prompted with the goal it needs to accomplish, conditioning its every action choice. If the agent manages to abide by the initial request, at the end of the episode the returns-to-go should be equal to zero, meaning that it indeed reached the desired return.

Chapter 5

Original work

In this chapter we explain and discuss the approach that was applied to make an agent learn the game of Briscola. The code implementation is available on GitHub at BriscolaOfflineRL.

The general idea is to treat the game as a sequence of tokens, where the tokens are returns-to-go, states and actions, which compose the trajectory. All the games available in the dataset at our disposal are going to be converted to trajectories and used to train a Decision Transformer, which will then be evaluated against the agent that generated the dataset.

5.1 Dataset

The dataset is composed of a total of 10^6 games played by a trained agent against himself: BriscolaBot (Lorenzo 2023). This is a reinforcement learning agent that learnt to play Briscola using the Proximal Policy Optimization (PPO) algorithm, being trained for around 2.5×10^6 games.

The 10^6 games of the dataset are translated to a corresponding number of trajectories by taking one player as the agent and the other player as the opponent. However, this number could be doubled if we were to consider every game from the opponent's point of view, resulting in an entire different game in the eyes of the agent. This data augmentation was not performed as the training time given the available resources would have become prohibitive.

5.1.1 Encoding

The forty cards that compose the deck are encoded with numbers from 0 to 40: from 0 to 9 the first suit, from 10 to 19 the second and so on; 40 is used to indicate an unknown card.
We made no distinction among the four suits as only the suit of briscola, which is chosen at the beginning of every game, makes a difference, while the other three suits are left at the same level.
There again, we then had to use a hot-encoding of the state cards in order to feed them into the model.

5.2 Trajectory

5.2.1 State

The state is composed of 6 cards: first the three cards in the hand of the agent; then the card played by the opponent, which is known only if the agent is playing second, otherwise is left unknown; the card of the opponent played in the previous trick and finally the card of briscola. We are including the opponent's card of the previous trick because in a trick where the agent plays first, it needs to see the response of the opponent. Therefore, adding that card, which would otherwise remain unknown, to the next state allows the agent to have all the information as in a real game.

Having delineated the representation of a state, we can compute the space size:

- $C(40, 3) \times {}^{37}P_3$ when the agent plays second during the first 18 tricks of a game,
- $C(40, 3) \times {}^{37}P_2$ when the agent plays first during the first 18 tricks of a game,
- $C(40, 2) \times {}^{38}P_3$ when the agent plays second during the 19th trick of a game,
- $C(40, 2) \times {}^{38}P_2$ when the agent plays first during the 19th trick of a game,
- $C(40, 1) \times {}^{39}P_3$ when the agent plays second during the last trick of a game,
- $C(40, 1) \times {}^{39}P_2$ when the agent plays first during the last trick of a game,

for a total of approximately 5×10^8, which is quite large, especially considering that our dataset contains only approximately 2×10^7 states.

In addition, at the end of the trajectory we added an extra state, representing the terminal state: $(40, 40, 40, 40, x, b)$, where b is the card of briscola, which is constant in all the state of a game, and x is the opponent's card of the last trick. This was done because otherwise x would remain unseen in the case where the agent played first in the last trick. Once again, this was done to provide the agent with all the information as it would happen in a real game.

5.2.2 Returns-to-go

In our context we do not focus on rewards but on returns-to-go: at the beginning of the trajectory it is the total points made throughout the game by the agent, and as the game evolves it decrease by the amount of points made by the agent when winning a trick, ending with zero points which correspond to the actual points at the beginning of the game. Therefore, the returns-to-go is allowed to assume values between 0 and 120, even though the upper and lower limits are very hard to reach and actually never reached in the dataset. As a matter of fact, the average winning points is $71.0416 \approx 71$ and consequently the average losing points is around 49.

A second option, which we tried during training, is to consider also the points made by the opponent during the game: the reward starts at 0 and as the game evolves it shifts up or down by the amount of points made by the agent and the opponent respectively, resulting in a value centered around zero. The returns-to-go is then this sequence of rewards turned upside down, beginning with the final value obtained which then decreases when the agent makes points and increases when the opponent does.
Being in the realm of reinforcement learning, our idea was that the agent would learn more easily if an action is not a good decision by seeing a shift in the wrong direction of the returns-to-go. However, after training on both kinds of trajectory, the evaluation did not show a significance difference between the two. Therefore, being the first setting much simpler to understand and implement, we decided to adopt that strategy throughout the learning process.

5.2.3 Action

The action that the agent is allowed to take is among the cards in his hand, i.e. a number from 0 to 40. We are including 40 because in the last two tricks the agent has only two and one card and the missing cards are indicated in the state as an unknown with the number 40. In addition, due to the architecture implementation, the model requires the number of states, actions and returns-to-go in a trajectory to be equal; hence, having added an extra state, we also added an extra "terminal action" which is again indicated with 40.

5.2.4 Episode trajectory

A trajectory is therefore a sequence of returns-to-go $\hat{R}_i$, states s_i and actions a_i unfolding as $\hat{R}_1, s_1, a_1, \hat{R}_2, s_2, a_2, ..., \hat{R}_T, s_T, a_T$, where $T = 21$ is the termination in which

$\hat{R}_T = 0$, while the state and the action are as explained above, for a total of 63 tokens.

5.3 Network architecture

The network architecture we used is a simplification of the original Decision Transformer (Chen et al. 2021) and divides into three main modules.

5.3.1 Masked Causal Attention

This module implements the attention mechanism. It consists of three different linear layers that applied to the input data generate queries, keys and vectors. Then the weights are computed by multiplying queries and keys and keeping only the relevant part of the resulting matrix by applying the causal mask. Finally, softmax is applied to normalize the weights.
After that, the values are multiplied by the normalized weights and the result undergoes first to a drop out layer, then to a linear layer and another drop out layer.

5.3.2 Block

This module represents a single block in the transformer architecture. Each block contains a Masked Causal Attention with a residual connection followed by LayerNorm and a Multi-Layer Perceptron layer once again with a residual connection and followed by LayerNorm. The MLP applies non-linear transformations to the output x of the attention module. In particular, it utilizes the GELU activation function (Hendrycks and Gimpel 2023):

$$\mathrm{GELU}(x) = x * \Phi(x) \tag{5.1}$$

where $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution.

5.3.3 Decision Transformer

The Decision Transformer is the main component of the network. It consists of a number of blocks stacked together, 12 in Radford et al. 2018 and from 1 to 4 in this thesis, and takes as input the current timestep, states, actions, and returns-to-go. The model embeds the inputs into a high-dimensional space using linear transformations, for states and returns-to-go, and embeddings, for timesteps and actions.
We had to think carefully how to embed the state, as being a six entries vector we could not apply the embedding directly. Therefore, we decided to divide the state into four chunks: the agent's cards, the opponent card, the opponent previous card and the

briscola. These four chunks were then hot-encoded and fed to four different linear layers that mapped them to a fourth of the embedding space. In this way the chunks could then be concatenated into one entity which results embedded in the same dimensional space of the other inputs, as desired.
In particular, we could three-hot-encode the agent's card because they are not ordered in any specific way, they are actually all at the same level in the eyes of the agent, making this configuration representative of the reality.

The embedded inputs are then concatenated and, after applying LayerNorm, passed through the transformer blocks.

After processing through the transformer blocks, the model predicts the next state, action weights, and the next returns-to-go value by means of three linear layers that downsize back the output of the blocks from the embedding dimension to their actual sizes.
Finally, the model uses a softmax function to predict the probabilities of different actions.

5.4 Training

We started the training by trying to understand the best architecture we could implement given the available computing resources: a standard 4-core computer, specifically Quad-Core Intel Core i5.

5.4.1 Loss function

In order to compute the loss of the predicted action, we used the cross-entropy function (Agarwala et al. 2020):

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C}\mathbf{y}_i[c]\log(\mathbf{p}_i[c]) \tag{5.2}$$

where:

- N is the sample size,
- C is the number of action classes,
- $\mathbf{y}_i$ is the ground truth one-hot encoded label for action i, where $\mathbf{y}_i \in \{0,1\}^C$,
- $\mathbf{p}_i$ is the predicted probability distribution over the C classes.

We tried to predict also the returns-to-go, but its loss goes to 0 straightaway and does not have any impact on the total loss as Chen et al. 2021 also reports. In addition, being the state stochastic we did not consider its prediction either.

5.4.2 Architecture

Taking the parameters setting of Chen et al. 2021 as a starting point, including the embedding dimension set to 128, we compared the performance of different numbers of blocks and heads by training the transformer for 2 epochs (Table 5.1; Table 5.2).

The different resulting models were evaluated both against a random player and against BriscolaBot, recording also the % of unfeasible actions, meaning that the predicted action was not present in the agent's card. For example, one unfeasible action in 50 games corresponds to 0.1%.

Blocks	**Random**	**BriscolaBot**	**Unfeasible**
1	78.9	25.1	1.05
2	85.5	32.2	0.46
3	86.0	34.6	0.44
4	86.9	36.6	0.25

Table 5.1: Winning % against random agent and BriscolaBot, obtained playing 1000 games each, and % of unfeasible actions for varying block number.

Looking at the different blocks configuration in Table 5.1 we can notice that 4 blocks yield the best overall performance. This means that a too small architecture is not able to capture fully the complexity of the game.
Nevertheless, since there is not an enormous difference between using 2 and 4 blocks, we opted to tune the number of heads and the parameters using a 2-block architecture. The reason is that training this model is approximately 30% faster than training one with 4 blocks, which is a straightforward consequence of the number of parameters that need to be learnt: 4×10^5 versus 8×10^5.

Heads	**Random**	**BriscolaBot**	**Unfeasible**
1	86.9	32.2	0.63
2	84.2	31.3	0.64
4	85.6	33.6	0.38
8	86.7	33.3	0.60

Table 5.2: Winning % against random agent and BriscolaBot, obtained playing 1000 games each, and % of unfeasible actions for varying head number.

With regards to the number of heads, there is not much to say: 4 heads are the right choice having a much lower number of action mistakes.
It is important to notice that the winning % cannot be used as a definite metric for comparison because it is somewhat stochastic, having the game also a random component when drawing cards from the deck. Throughout the experiments, we noticed that it can shift up or down by more or less one point percentage.

5.4.3 Embedding dimension

Once we decided what the most promising architecture was, we delved into hyper-parameters tuning and firstly we focused on the embedding dimension (Table 5.3).

Embedding	**Random**	**BriscolaBot**	**Unfeasible**
128	86.1	33.6	0.45
256	88.0	39.8	0.13
384	88.6	38.6	0.06

Table 5.3: Winning % against random agent and BriscolaBot, obtained playing 1000 games each, and % of unfeasible actions for varying embedding dimension.

The greater the dimension, the smaller the number of mistakes. This is probably due to the fact that with a bigger dimensional space the model can distinguish more easily between two states, whose representation is the most complex to discern in the trajectory (Guo and Berkhahn 2016). With regards to performance, it seems that the middle model can outperform the biggest, especially against BriscolaBot. We speculate that with a smaller embedding dimension, the model could end up grasping more of the agent's strategy present in the dataset games and focusing less on making feasible actions.

Anyway, the greatest embedding dimension requires a training time 75% longer with respect to 256. Indeed, the number of learnable parameters more than doubles going from 256 to 384.
Therefore, we were forced to choose the smaller dimension between these two.

5.4.4 Batch size and learning rate

The batch size refers to the number of samples, in our case trajectories, processed by the model in one forward pass.
Smaller batch sizes provide noisier gradient estimates due to the limited number of samples, whereas larger batch sizes offer more accurate estimates. Therefore, larger batch sizes often lead to more stable training dynamics because they provide smoother

updates to the model parameters.
Nevertheless, smaller batch sizes, by exposing the model to different subsets of the training data more frequently, may encourage better generalization by preventing overfitting (Smith 2018).

In addition, we had to consider also the hardware capacity, as a too large batch size could create a bottleneck by preventing the CPU to handle the data efficiently, leading to out-of-memory errors or reduced performance due to swapping (Fu et al. 2023). By monitoring the CPU utilization, we decided that the maximum batch size could be 64. Hence, we tested three different values: 16, 32 and 64.

With regards to the learning rate, smaller values often lead to more stable, though slower, training, as they result in smaller updates to the model parameters; instead, high values can speed up the training process by making larger updates, especially in the early stages of training (Smith 2018). In complex optimization landscapes large updates might overshoot the optimal solution, but at the same time they can help the optimization algorithm escape local minima by encouraging exploration of the parameter space, potentially leading to better generalization or finding better solutions.

We decided to adopt a learning rate warm-up: starting with a low value that increases gradually for a number of warm-up steps and then settles to the max value, which was set to 1, for the remaining training steps. The number of warm-up teps in Chen et al. 2021 is set to a fixed value 10^5, but by changing the batch size and the number of epochs we can end up with a different number of updates. Therefore, we employed a proportional warm-up value, precisely one third of the total training steps.

Considering batch size and learning rate together, for larger batch sizes higher learning rates are often used to compensate for the reduced variance in the gradient estimates, whereas smaller batch sizes usually require lower learning rates to prevent miss the minimum.

Batch size	**Learning rate**	**Random**	**Unfeasible**
16	0.001	87.9	0.14
32	0.001	88.0	0.22
64	0.001	89.0	0.18
16	0.0001	87.9	0.16
32	0.0001	86.2	0.35
64	0.0001	85.3	0.45

Table 5.4: Winning % against random agent, obtained playing 1000 games each, and % of unfeasible actions for varying batch size and learning rate.

We can clearly notice in Table 5.4 the correlation between the two parameters. In addition, the best generalization is obtained using a batch size = 16, while the best strategy, i.e. the winning %, is obtained with a bigger learning rate. Being the generalization more important than the strategy, at least in an initial stage where the agent must learn what are the feasible actions, we decided to conduct the training with this combination of parameters: 16 and 0.001.

5.4.5 Weight decay and temperature

Weight decay is regularization a technique to prevent overfitting. An additional term $R(w)$ is added to the loss function in order to penalizes large weights w in the model:

$$R(w) = \lambda \frac{1}{2} ||w||_2^2 \tag{5.3}$$

where λ is the weighting term controlling the importance of the regularization over the consistency (Kukačka, Golkov, and Cremers 2017).
Too small a value of λ may not provide sufficient regularization, leading to overfitting, while too large a value may result in underfitting. We decide to test for two values, a small value which is the one used by Chen et al. 2021 and a bigger one. From the results in Table 5.5 it is clear that 0.001 is a making the model learn more efficiently.

λ	**Random**	**Unfeasible**
0.001	88.0	0.38
0.0001	85.3	0.46

Table 5.5: Winning % against random agent obtained playing 1000 games, and % of unfeasible actions for varying weight decay importance.

Throughout the tuning of the parameters we noticed that after the first epoch the loss was already approaching a plateau, and during the second epoch the model improvement was negligible. This so called plateau phenomenon can occur when the model has learned as much as it can from the available training data, or when it has reached a local minimum in the optimization landscape where further improvements are difficult to achieve (Ainsworth and Shin 2020).

Therefore, we opted to add a parameter that allows for more explorartion to inspect if it would help find a different, possibly better, solution or escape a local minimum: the temperature. This parameter, indicated with τ, is found inside the softmax of action x_i (He et al. 2018):

$$\text{Softmax}(x_i) = \frac{\exp(\frac{x_i}{\tau})}{\exp(\sum_j(\frac{x_i}{\tau}))} \tag{5.4}$$

It pushes for more exploration by increasing the entropy of the output, or in other words it spreads the probabilities across all the possibilities, resulting in more uniform action distributions. When equal to 1, it corresponds to the standard softmax function.

τ	**Random**	**Unfeasible**
1	84.5	0.91
1.5	84.9	0.48
2	87.1	0.32

Table 5.6: Winning % against random agent obtained playing 1000 games, and % of unfeasible actions for varying temperature values.

The values in Table 5.6 show a clear distinction, particularly in the model generalization, between using and not using the temperature parameter. For the next experiments we decided to adopt the biggest value of τ, as it yields also the highest winning percentage.

5.4.6 Tuned training

After setting architecture and parameters (Table 5.7), we conducted a long training evaluating the agent only against a random player. The model was trained for 12 epochs, which took approximately 86 hours, to find out if it was possible to escape from the plateau.

Parameter	**Initial value**	**Selected value**
Blocks	12	2
Heads	1	4
Embedding dimension	128	256
Batch size	64	16
Learning rate	0.0001	0.001
Weight decay	0.0001	0.001
Temperature	1	2

Table 5.7: Comparison between initial parameters (Chen et al. 2021) and selected parameters found by training the transformer for 2 epochs on a set of arbitrarily chosen parameters.

With this configuration of parameters it actually took around 250000 steps, approximately 4 epochs, to just enter the plateau, and afterwards the loss sill keeps decreasing, but at a very slow rate (Figure 5.1).

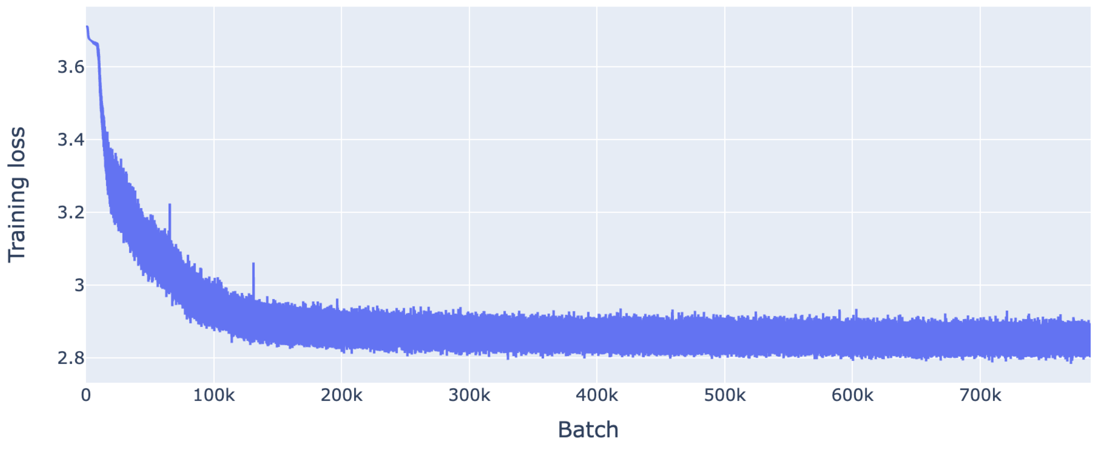


Figure 5.1: Cross entropy loss during training computed for every processed batch.

With regards to evaluation (Figure 5.2), we can see a similar pattern. During training, around the same time as the loss, the model reaches a plateau and does not increase its performance anymore: another indication that the improvement from the fourth epoch onwards becomes very slim.

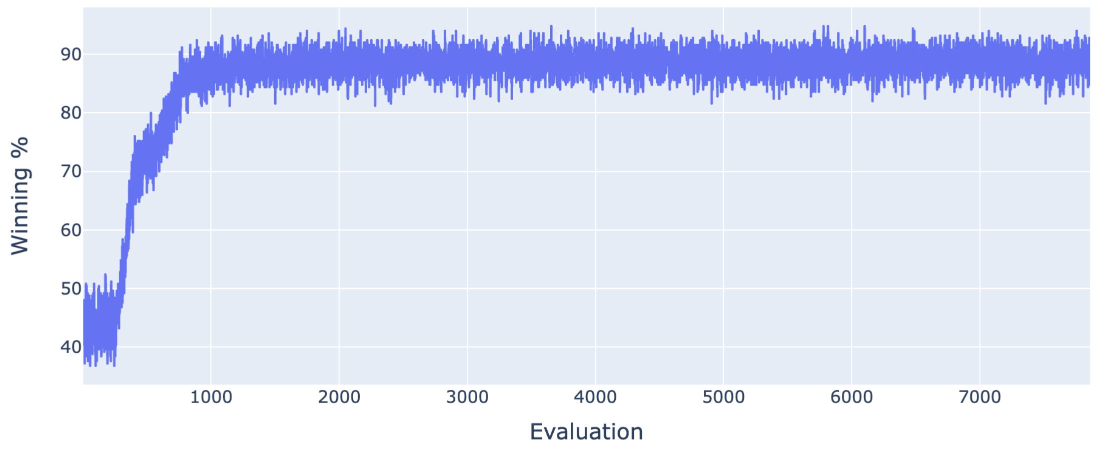


Figure 5.2: Winning % out of 250 games played by the agent against a random player every 100 batches.

Turning our attention to unfeasible actions (Figure 5.3), learning is apparently much faster: after 200 evaluations we are already approaching the minimum value. Unfortunately, even though the average % is low, it is constantly fluctuating and 0 is reached at most for a few evaluations in a row. This is most probably due to two main reasons: either because the agent is seeing too few states during training and they are not sufficient to make it generalize to all the unseen states during evaluation, or because the embedding dimension is too small, preventing the model from discerning between two close states representation.

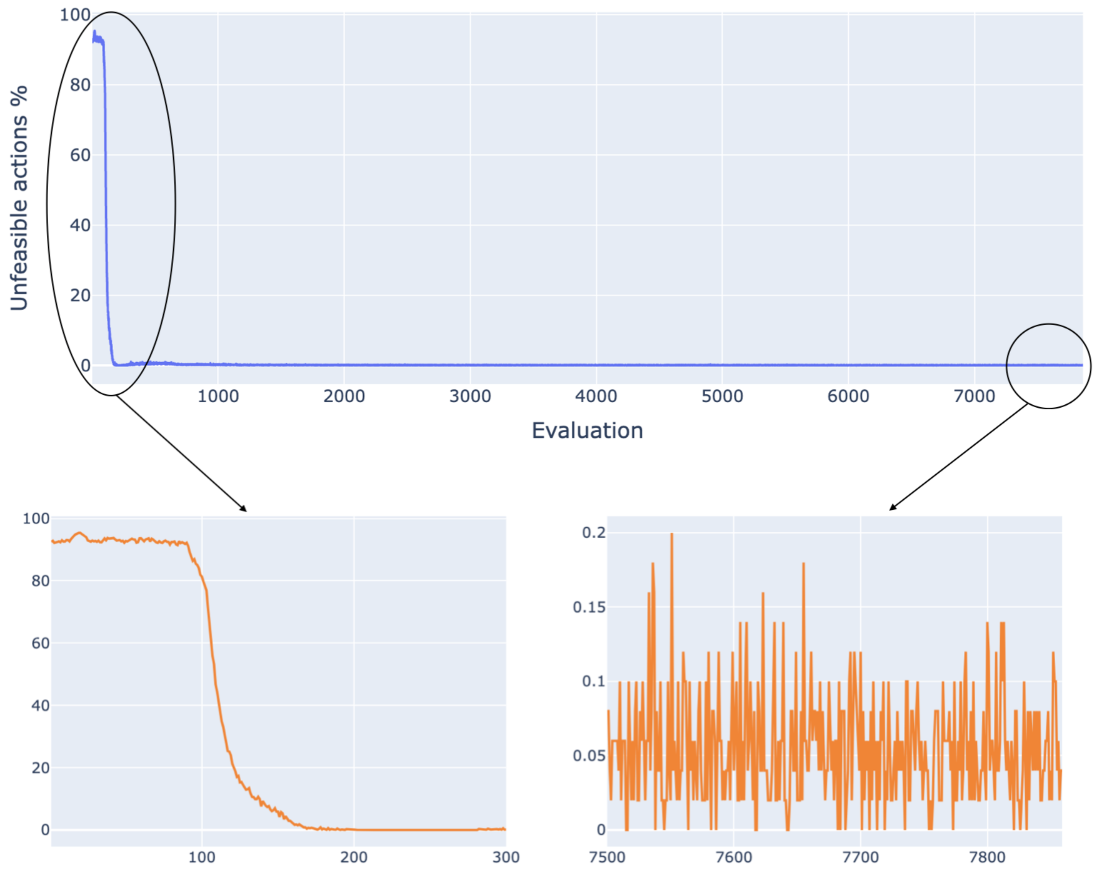


Figure 5.3: Unfeasible actions % out of 250 games played by the agent against a random player every 100 batches, including a zoom in on beginning and ending to see the pattern more clearly.

In order to understand if there was a pattern in the unfeasible actions predicted by the model, we inspected several games:

Agent's cards	39, 37, 2	10, 3, 2	3, 10, 2	6, 12, 13	30, 10, 5	9, 13, 7
Unfeasible action	10	9	12	11	32	16
Agent's cards	7, 13, 0	0, 40, 20	36, 17, 26	32, 5, 26	15, 6, 21	15, 25, 33
Unfeasible action	16	14	20	4	3	24

Table 5.8: Several unfeasible actions with the corresponding agent's card.

Although there seems not to be a clear correlation, in one third of cases the wrong selection was a card right next to one of the agent's cards, corroborating the idea of a too small embedding dimension which prevents the model to distinguish between very similar states.

5.5 Trained model inspection

During training we saved the model with the highest winning % against the random player so as to test it against BriscolaBot. As we mentioned earlier, the average value of winning points in the dataset is around 71, so it would make sense to choose this value as the target returns-to-go. Anyway, we decided to test the agent for returns-to-go ranging from 61 to 120 to check whether this parameter could actually make a difference.

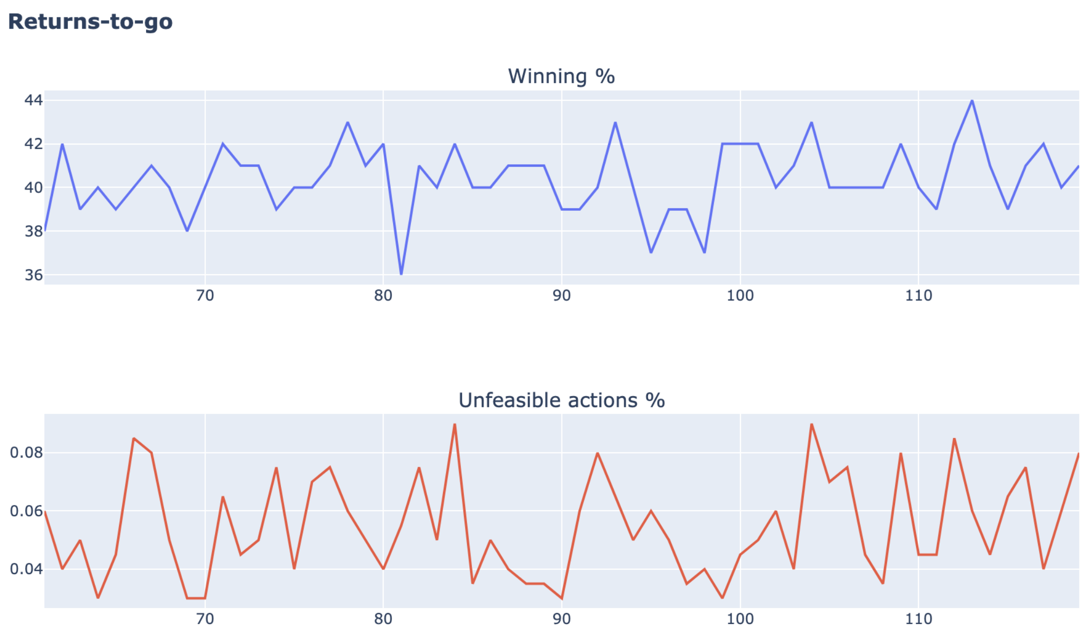


Figure 5.4: Winning % and unfeasible actions % out of 1000 games against BriscolaBot for every value of the returns-to-go target, ranging from 61 to 120.

Figure 5.4 shows clearly that there is not a best value, neither with respect to winning %, nor with respect to unfeasible actions %. Indeed, in both graphs there is a random pattern and even repeating the test, keeping in mind that there always is a stochastic component, did not show any outstanding values.

Anyway, we noticed that asking for a returns-to-go smaller than 60, meaning that the agent should try to lose the game, the agent still wins the same % of games, i.e. the agent's decisions are totally independent of the target returns-to-go. Even though predicting returns-to-go did not increase the agent's performance, as we mentioned in section 5.4.1, we decided to do so in order to inspect if this could allow to condition the agent in its choices, which is the correct functioning of the decision transformer. We conducted a new training, decreasing the embedding dimension to 128 and the number of epochs to 4, in order to reduce the computational time but not losing too much performance.

Figure 5.5 shows what we managed to achieve: the winning % is definitely smaller for a returns-to-go less than 20, then it increases for returns-to-go ranging from 20 to 50 and finally becomes "stable" around 40% for values greater than 50. However, the ability of losing on purpose actually comes with a cost. Indeed, for returns-to-go less than 20 the % of unfeasible actions is much bigger. This could mean that the agent did not learn counterproductive actions and hence, when forced to lose, ends up making mistakes.

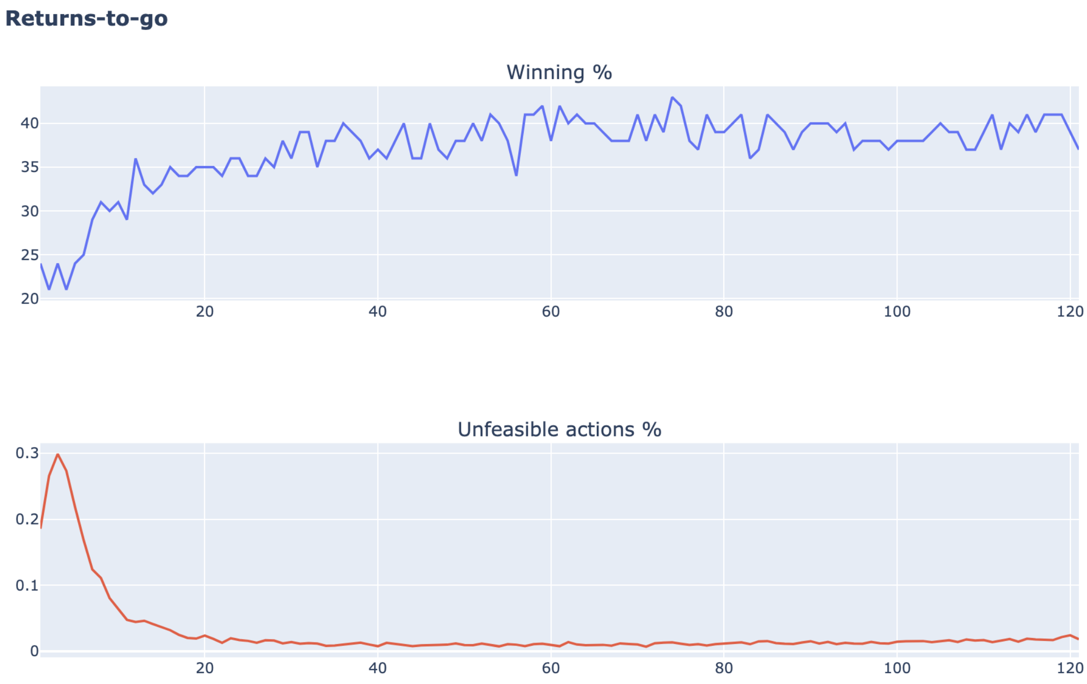


Figure 5.5: Winning % and unfeasible actions % out of 1000 games against BriscolaBot for every value of the returns-to-go target, ranging from 0 to 120.

The long trained agent won 40.4% of 6×10^4 games with 0.055% unfeasible actions against BriscolaBot, meaning that there was a mistake every 90 games, more or less. Therefore, we could not achieve a fully correct model. In addition, our decision transformer has the advantage, with respect to BriscolaBot, of exploiting past tokens, which should yield wiser choices. Nevertheless, our model did not learn enough game strategy to surpass the agent that generated the training data.

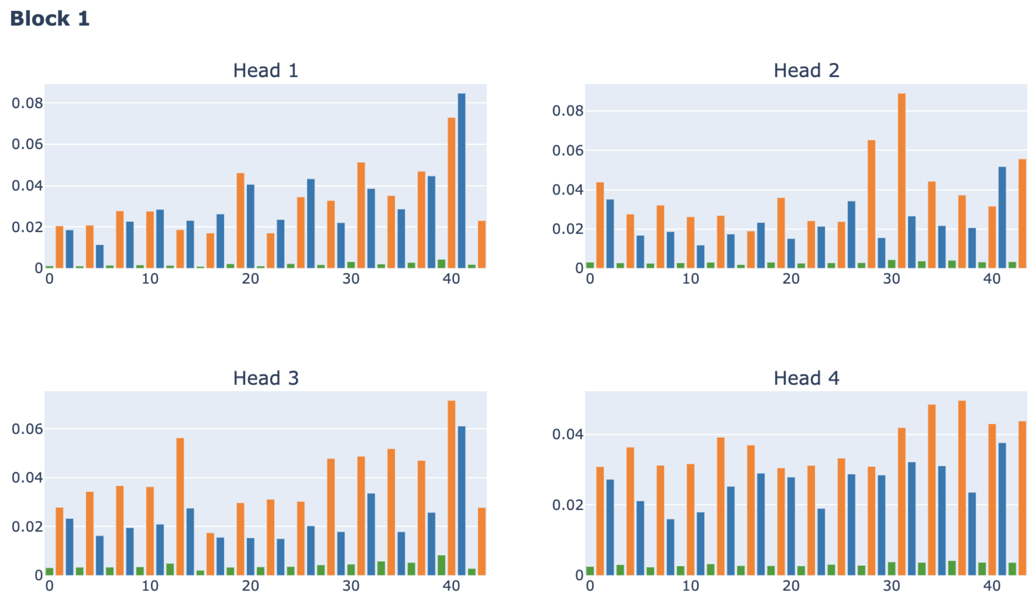


Figure 5.6: The attention given by the heads of block 1 to the past trajectory in order to predict the action at the 15th trick of a game.
Returns-to-go = green, states = orange, actions = blue.

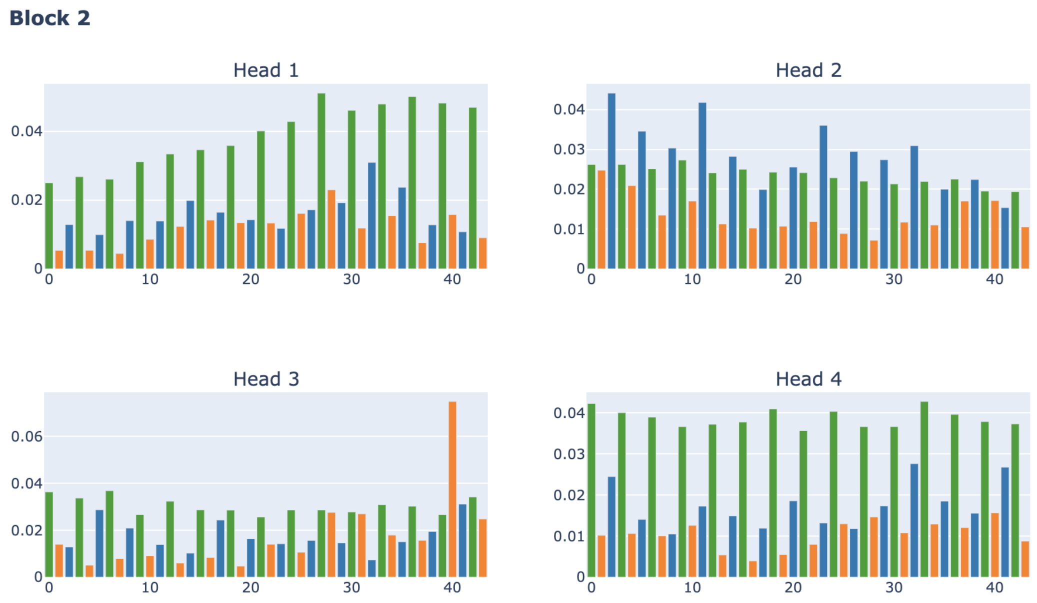


Figure 5.7: The attention given by the heads of block 2 to the past trajectory in order to predict the action at the 15th trick of a game.
Returns-to-go = green, states = orange, actions = blue.

Next, we decided to inspect the hidden procedures of this trained decision transformer.

Figures 5.6 and 5.7 show how the attention mechanism actually works. In our architecture there are 2 blocks and 4 heads per each block, and every one of them is concentrating on a different part of the trajectory. The heads of block 1 are overall focusing on the states and somewhat on the actions, while neglecting the returns-to go. On the other hand, half of the heads in block 2 concentrate on the returns-to go and another head seems to average out the three kind of tokens. Instead, the center of attention of the last head is the actions and it is actually weighing more the oldest ones, meaning that the model is perfectly able to exploit past items of the trajectory when making predictions.

Another way of visualizing the attention weights of a head is by means of a heatmap (Figure 5.8) (J. Li et al. 2016). It allows to observe in the columns the weights of the past trajectory as the actions follow one another along the x-axis. As we move towards the end of the game, the length of the past trajectory increases and the weights spread across it, but the most weighted tokens are still noticeable.

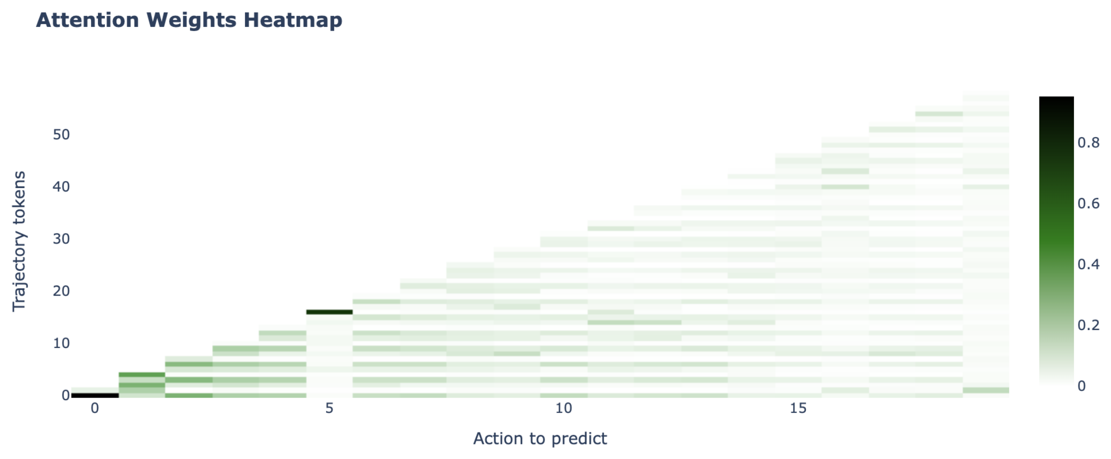


Figure 5.8: Heatmap of the attention weights of one head throughout a game.

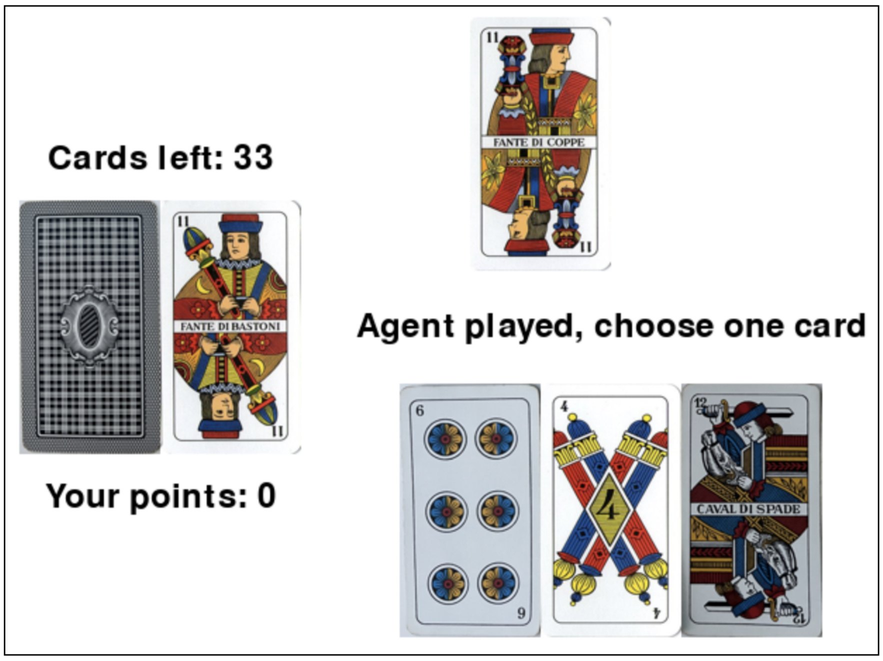
5.6 Human evaluation

We evaluated the performance of the trained agent by having human players play against it using a very easy Pygame interface (Figure 5.9). All players that took part in the experiment can be classified as intermediate level. The results of the evaluation are presented in Table 5.9.

Total games	Wins	Draws	Defeats	Winning rate
65	39	1	25	60%

Table 5.9: Human evaluation results.

Despite everything, a 60% winning rate is a satisfactory result. Clearly, a bigger sample would be needed to find the real performance of the agent against human players.


Cards left: 33

Agent played, choose one card

Your points: 0

Figure 5.9: The Pygame interface for human evaluation.

During these evaluation games, we observed the strategy of the agent:

- Especially in the first half of the game, it throws the ace and the three of briscola very easily, while it may be wiser to keep them for better later use. In addition, it tends to use stronger briscola cards to win over weaker ones, even if the latter have 0 points.
- Generally, it is able to use correctly the cards of briscola, choosing them in order to win the points of the opponent's card.
- It usually throws a card of the suit of the ace or three that has just been played, which is done in order to possibly prevent the opponent from using aces or threes of other suits still in the game.
- Finally, in the last tricks of the game, when the deck is finished, it plays very wisely. This shows that it is able to exploit the information of the rest of the game and make decisions based on the cards that the opponent must have in its hand.

Chapter 6

Discussion

In this work we trained a decision transformer with the goal of learning the game of Briscola and compete against BriscolaBot (Lorenzo 2023). Due to resource limitations, we had to restrict both dataset and architecture sizes. We still managed to surpass 40% of winning rate with an agent that does not choose unfeasible actions in most of the games. However, in order to improve the performance, the embedding dimension should definitely be increased, as well as the number of blocks that needs be raised to 4, at least, assuming that training can be conducted on a GPU. Indeed, the trasformer architecture is easily parallelizable and training on a GPU would allow to exploit this quality and speed up the process.

Training was stopped because of a too long computational time, hence we never really reached the ”end” of it. As a matter of fact, even a small decrease in loss, like in our case (Figure 5.1), may indicate learning, especially if the problem is complex or the dataset is large. Hence, a longer training should be conducted to examine how much the model can still learn.

The plateau we reached corresponds to a period during which activation patterns, i.e. the number of data points that activate a given neuron, remain constant. An explicit adjustment of the activation pattern at every step could be enable to induce a quick exit from the plateau region (Ainsworth and Shin 2020). Another option, while the model is still training, is to generate new games with the current policy. This new data would be sub-optimal, but it may help modify the activation patterns, therefore escaping from the plateau, and at the same time it would provide more data for better generalization. By combining the decision transformer with an offline actor-critic method (Springenberg et al. 2024) we could feed back self-generated data to the offline RL optimization and allow for a much larger parameters number, outperforming the current trained model.

This could be taken even further by considering counterfactual data augmentation (Pitis et al. 2022), which consists in not just creating new training data, but specific data that was left unseen during training, enhancing even further the generalization capabilities of the model.

Something else worth investigating would be to remove the 21st state from the trajectory. It was added in order to show to the agent the last card played by the opponent, but being this card the last to be seen from the deck, the agent might still be able to understand what card it is by excluding all the already played cards.
This adjustment could make the training harder but the model should eventually learn regardless. Perhaps, it has already learnt to deduce the last card and the last state was acting merely as a confirmation of the model's understanding of the game.

In addition, firstly we selected the "optimal" architecture by arbitrary choices of parameters and without a proper architecture optimization; secondly, as we mentioned already, our architecture was probably too simple. However, this does not mean that it should be as complicated as possible. Balderas, Lastra, and Benítez 2024 developed a technique based on pruning that could be applied to our decision transformer with an increased number of layers and neurons in order to be optimized to find the best trade-off between complexity and performance.

Bibliography

Agarwala, Atish et al. (2020). *Temperature check: theory and practice for training models with softmax-cross-entropy losses*. arXiv: 2010.07344 [cs.LG].

Ainsworth, Mark and Yeonjong Shin (2020). *Plateau Phenomenon in Gradient Descent Training of ReLU networks: Explanation, Quantification and Avoidance*. arXiv: 2007.07213 [cs.LG].

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton (2016). *Layer Normalization*. arXiv: 1607.06450 [stat.ML].

Balderas, Luis, Miguel Lastra, and José M. Benítez (2024). "Optimizing dense feed-forward neural networks". In: *Neural Networks* 171, pp. 229–241.

Beslin Pajila, P.J. et al. (2023). "A Survey on Natural Language Processing and its Applications". In: *2023 4th International Conference on Electronics and Sustainable Communication Systems (ICESC)*, pp. 996–1001.

Chang, Jonathan D. et al. (2022). *Mitigating Covariate Shift in Imitation Learning via Offline Data Without Great Coverage*. arXiv: 2106.03207 [cs.LG].

Charles A. Holt, Alvin E. Roth (2004). "The Nash equilibrium: A perspective". In: *PNAS* 101.12.

Chen, Lili et al. (2021). *Decision Transformer: Reinforcement Learning via Sequence Modeling*. arXiv: 2106.01345 [cs.LG].

Devlin, Jacob et al. (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. arXiv: 1810.04805 [cs.CL].

Fu, Quchen et al. (2023). *Deep Learning Models on CPUs: A Methodology for Efficient Training*. arXiv: 2206.10034 [cs.LG].

Ghojogh, Benyamin and Ali Ghodsi (2020). *Attention Mechanism, Transformers, BERT, and GPT: Tutorial and Survey*.

Guo, Cheng and Felix Berkhahn (2016). *Entity Embeddings of Categorical Variables*. arXiv: 1604.06737 [cs.LG].

He, Yu-Lin et al. (2018). "Determining the optimal temperature parameter for Softmax function in reinforcement learning". In: *Applied Soft Computing* 70, pp. 80–85.

Hendrycks, Dan and Kevin Gimpel (2023). *Gaussian Error Linear Units (GELUs).* arXiv: 1606.08415 [cs.LG].

Hong, Joey, Aviral Kumar, and Sergey Levine (2023). *Confidence-Conditioned Value Functions for Offline Reinforcement Learning.* arXiv: 2212.04607 [cs.LG].

Hong, Zhang-Wei et al. (2023). *Beyond Uniform Sampling: Offline Reinforcement Learning with Imbalanced Datasets.* arXiv: 2310.04413 [cs.LG].

Hongchang, Zhang et al. (2022). "State Deviation Correction for Offline Reinforcement Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 36.8, pp. 9022–9030.

Kidambi, Rahul et al. (2021). *MOReL : Model-Based Offline Reinforcement Learning.* arXiv: 2005.05951 [cs.LG].

Kukačka, Jan, Vladimir Golkov, and Daniel Cremers (2017). *Regularization for Deep Learning: A Taxonomy.* arXiv: 1710.10686 [cs.LG].

Kumar, Aviral et al. (2020). *Conservative Q-Learning for Offline Reinforcement Learning.* arXiv: 2006.04779 [cs.LG].

Levine, Sergey et al. (2020). *Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems.* arXiv: 2005.01643 [cs.LG].

Li, Jiwei et al. (2016). *Visualizing and Understanding Neural Models in NLP.* arXiv: 1506.01066 [cs.CL].

Li, Wenzhe et al. (2023). *A Survey on Transformers in Reinforcement Learning.* arXiv: 2301.03044 [cs.LG].

Lin, Tianyang et al. (2021). *A Survey of Transformers.* arXiv: 2106.04554 [cs.LG].

Lorenzo, Cavuoti (2023). *BriscolaBot: Mastering Briscola with model-free Deep Reinforcement Learning.* https://github.com/LetteraUnica/BriscolaBot.

Mitchell, Eric et al. (2021). *Offline Meta-Reinforcement Learning with Advantage Weighting.* arXiv: 2008.06043 [cs.LG].

Nie, Allen et al. (2023). *Data-Efficient Pipeline for Offline Reinforcement Learning with Limited Data.* arXiv: 2210.08642 [cs.LG].

Noam Brown, Tuomas Sandholm (2017). *Safe and Nested Subgame Solving for Imperfect-Information Games.* arXiv: 1705.02955 [cs.AI].

O'Shea, Keiron and Ryan Nash (2015). *An Introduction to Convolutional Neural Networks.* arXiv: 1511.08458 [cs.NE].

Pitis, Silviu et al. (2022). *MoCoDA: Model-based Counterfactual Data Augmentation.* arXiv: 2210.11287 [cs.LG].

Polosky, Nicholas et al. (2022). "Constrained Offline Policy Optimization". In: *Proceedings of the 39th International Conference on Machine Learning.* Vol. 162. Proceedings of Machine Learning Research. PMLR, pp. 17801–17810.

Radford, Alec et al. (2018). *Improving language understanding by generative pre-training.*

Richard S. Sutton, Andrew G. Barto (2018). *Reinforcement Learning: An Introduction.* Second. The MIT Press.

Sandholm, Tuomas (2015). “Solving imperfect-information games”. In: *Science* 347, pp. 122–123.

Sazli, Murat (2006). “A brief review of feed-forward neural networks”. In: *Communications Faculty Of Science University of Ankara* 50, pp. 11–17.

Schäfer, Anton Maximilian (2008). “Reinforcement Learning with Recurrent Neural Networks”. PhD thesis. Osnabrück University.

Schmidhuber, Jürgen (2015). “Deep learning in neural networks: An overview”. In: *Neural Networks* 61, pp. 85–117.

Schmidt, Robin M. (2019). *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview.* arXiv: `1912.05911 [cs.LG]`.

Schweighofer, Kajetan et al. (2022). *A Dataset Perspective on Offline Reinforcement Learning.* arXiv: `2111.04714 [cs.LG]`.

Sherstinsky, Alex (2020). “Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network”. In: *Physica D: Nonlinear Phenomena* 404.

Smith, Leslie N. (2018). *A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay.* arXiv: `1803.09820 [cs.LG]`.

Springenberg, Jost Tobias et al. (2024). *Offline Actor-Critic Reinforcement Learning Scales to Large Models.* arXiv: `2402.05546 [cs.LG]`.

Torfi, Amirsina et al. (2021). *Natural Language Processing Advancements By Deep Learning: A Survey.* arXiv: `2003.01200 [cs.CL]`.

Turner, Richard E. (2024). *An Introduction to Transformers.* arXiv: `2304.10557 [cs.LG]`.

Vaswani, Ashish et al. (2017). “Attention is All you Need”. In: *Advances in Neural Information Processing Systems.* Vol. 30. Curran Associates, Inc.

Wang, Wei and Jianxun Gang (2018). “Application of Convolutional Neural Network in Natural Language Processing”. In: *2018 International Conference on Information Systems and Computer Aided Education (ICISCAE)*, pp. 64–70.

Wikipedia (2024). *Briscola.* URL: `https://en.wikipedia.org/wiki/Briscola`.

Wu, Yifan, George Tucker, and Ofir Nachum (2019). *Behavior Regularized Offline Reinforcement Learning.* arXiv: `1911.11361 [cs.LG]`.

Zhang, Kaiqing, Zhuoran Yang, and Tamer Başar (2021). *Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms.* arXiv: `1911.10635 [cs.LG]`.