Jagiellonian University
Department of Theoretical Computer Science

Master's Thesis

Deep-Hedge MCCFR

An algorithm for solving imperfect information games

Author:
Marcin Ziemiński

Supervisor:
dr hab. Bartosz Walczak

September 2020

Contents

1 Introduction

There have been much progress in the area of training agents to tackle difficult sequential decision-making problems. The dominating solutions from the realm of deep reinforcement learning (Deep RL) [Sutton and Barto, 2018] proved to be extremely successful surpassing the top human players at Atari games [Mnih et al., 2015] or Go [Silver et al., 2016]. These algorithms adapt their behaviours directly from their experience and received rewards.

But on the downside, the Deep RL methods are not well suited to multi-agent environments, which are non-stationary by nature, as the probability distribution over possible outcomes may change over time. The agents within the same environment learn their strategies in parallel and modify their behaviour. What is more, they may have access to private information, concealed from the others. This makes the optimization problem more difficult, but at the same time amenable to game-theoretic analysis.

Game theory is at the centre of the prevailing approaches for solving imperfect information competitive games. The Counterfactual Regret Minimization (CFR) algorithm [Zinkevich et al., 2008] was the base for the poker AI called Pluribus [Brown and Sandholm, 2019], which defeated the best human professionals in six-player no-limit Texas hold'em. CFR is an adaptive tabular algorithm, which means that it aggregates information per each state of the game individually and uses it to iteratively adapt the strategy. However, in order to be feasible for large games, it requires a significant portion of domain-specific knowledge and a look-ahead search during the execution.

In this work we try to combine the two aforementioned paradigms. We propose an algorithm founded on Counterfactual Regret Minimization, which utilizes the advancements of deep learning [Goodfellow et al., 2016]. Our method, called Deep-Hedge MCCFR, uses neural networks to model the agents' strategies. The strategies are improved through the experience gathered during self-play. We show that the algorithm does not require domain expertise and is applicable to various scenarios in unmodified form.

2 Background

Game theory being at the core of this work is undeniably a vast area of science. Therefore we lay the groundwork for further discussion by bringing up the fundamental concepts and results. We hope to help the readers previously unexposed to it obtain basic familiarity.

2.1 Normal-Form Games

For the sake of clarity and seamless transition to the topics of higher complexity, we first focus on games in their most general form. Following [Shoham and Leyton-Brown, 2008] we start by introducing an n-person *normal-form game*.

Definition 2.1 (normal-form game)**.** A finite n-person **normal-form game** is defined as a tuple $(\mathcal{N}, \mathcal{A}, u)$, where:

- $\mathcal{N} = \{1, \ldots, n\}$ is a finite set of players,
- $\mathcal{A} = \mathcal{A}_1 \times \cdots \times \mathcal{A}_n$, where $\mathcal{A}_i$ is a finite set of **actions** available to player i. A tuple $a = (a_1, \ldots, a_n)$ is called an **action profile**.
- $u = (u_1, \ldots, u_n)$, where $u_i : \mathcal{A} \to \mathbb{R}$ is a **utility** (or **payoff**) **function** for player i.

There are various kinds of games, some of which exhibit somewhat distinctive nature, especially from the computational point of view. Let us recall the definition of *zero-sum* games, on which we will focus attention in the later parts of our work.

Definition 2.2 (zero-sum game)**.** A normal-form game $(\mathcal{N}, \mathcal{A}, u)$ is **zero-sum** if for every action profile the sum of the payoffs of all players is equal to 0, that is: $\forall_{a \in \mathcal{A}} \sum_{i \in \mathcal{N}} u_i(a) = 0$.

Having introduced the notion of a game itself, we need to define a *strategy* of a player, which is a central part of our analysis.

Definition 2.3 ((mixed) strategy)**.** A (**mixed**) **strategy** of a player i in a normal-form game $(\mathcal{N}, \mathcal{A}, u)$ is a probability distribution σ_i over the set of actions $\mathcal{A}_i$. Let Σ_i denote the set of all strategies (probability distributions) of player i.

A **pure strategy** is a strategy which is expressed as a deterministic probability distribution.

A vector $\sigma = (\sigma_1, \ldots, \sigma_n)$ is called a **strategy profile**, with

$$\sigma_{-i} = (\sigma_1, \ldots, \sigma_{i-1}, \sigma_{i+1}, \ldots, \sigma_n)$$

referring to all strategies except for σ_i. Thus, we can write $\sigma = (\sigma_i, \sigma_{-i})$.

Acknowledging the fact that players may choose their actions randomly, there is a need for expressing their payoffs in terms of expected values.

Definition 2.4 (expected utility)**.** Let $(\mathcal{N}, \mathcal{A}, u)$ be a normal-form game and σ be a strategy profile. The expected utility u_i for a player i is given as:

$$u_i(\sigma) = \mathbb{E}_{a \sim \sigma} u_i(a) = \sum_{a \in \mathcal{A}} u_i(a) \prod_{j \in \mathcal{N}} \sigma_j(a_j).$$

Assuming a fixed strategy profile σ_{-i}, player i has an incentive to maximize their payoffs by playing a *best response* strategy:

Definition 2.5 (best response)**.** A strategy σ_i^* of a player i is a **best response** strategy to a strategy profile σ_{-i} if:

$$\forall_{\sigma_i \in \Sigma_i} \quad u_i(\sigma_i^*, \sigma_{-i}) \geq u_i(\sigma_i, \sigma_{-i}).$$

We need to consider a "stable" strategy profile in which no player could get an advantage by deviating from their strategy. Such a profile, called *Nash equilibrium*, is a core concept of non-cooperative game theory.

Definition 2.6 (Nash equilibrium)**.** A strategy profile σ of a normal-form game $(\mathcal{N}, \mathcal{A}, u)$ is a (**mixed strategy**) **Nash equilibrium** if:

$$\forall_{i \in \mathcal{N}} \forall_{\sigma_i' \in \Sigma_i} \quad u_i(\sigma_i, \sigma_{-i}) \geq u_i(\sigma_i', \sigma_{-i}).$$

An equilibrium σ which consists only of pure strategies is called a **pure strategy Nash equilibrium**.

An approximation of a Nash equilibrium called **ϵ-Nash equilibrium** is a strategy profile such that:

$$\forall_{i \in \mathcal{N}} \forall_{\sigma_i' \in \Sigma_i} \quad u_i(\sigma_i, \sigma_{-i}) + \epsilon \geq u_i(\sigma_i', \sigma_i).$$

In other words, a Nash equilibrium is a strategy profile σ such that every player i plays their best response strategy to σ_{-i}.

The existence of Nash equilibria is guaranteed by the famous Nash theorem:

Theorem 2.7 ([Nash, 1950])**.** *Any finite normal-form game has a mixed strategy Nash equilibrium.*

We should emphasize here that Nash theorem does not entail the existence of pure strategy equilibria. On the contrary, there are games like the very familiar game of *Rock-Paper-Scissors*, which do not possess equilibria consisting of pure strategies. Furthermore, as stated in [Nisan et al., 2007] only two-player normal-form games always admit equilibria with rational probabilities, which is not true for games with three or more players. One should also note that a solution to finding such a strategy profile is not guaranteed to be unique, as there might exist multiple Nash equilibria.

It is a well-known fact that in the case of two-person zero-sum games having a precomputed Nash equilibrium σ, player i cannot lose in expectation by acting according to σ_i, no matter what the actual strategy σ'_{-i} chosen by the opponent is. In other words a Nash equilibrium strategy is unbeatable in games satisfying such criteria.

Unfortunately this property does not hold for other kinds of games. This is especially apparent in three or more person games, in which every player could independently compute and play an equilibrium strategy. In these circumstances the resulting strategy profile is not necessarily a Nash equilibrium and players might have an incentive to deviate.

So far, we have considered situations where the players choose their strategies independently. However there is also another concept of a game in which the players are provided with a public signal suggesting the actions to take. The players are free to make their own decisions by deviating from the observed values. There is also a notion of equilibrium involved, which in a sense is more general than Nash equilibrium defined in 2.6. It is defined formally as follows:

Definition 2.8 (correlated equilibrium)**.** Let $(\mathcal{N}, \mathcal{A}, u)$ be a normal-form game. A **correlated equilibrium** is a joint distribution q over $\mathcal{A} = \times_i \mathcal{A}_i$, such that for every player i and every actions $a_i, a'_i \in \mathcal{A}_i$:

$$\sum_{a_{-i} \in \mathcal{A}_{-i}} q(a_i, a_{-i}) u_i(a_i, a_{-i}) \geq \sum_{a_{-i} \in \mathcal{A}_{-i}} q(a_i, a_{-i}) u_i(a'_i, a_{-i}).$$

This definition is equivalent to saying that no player can increase their expected payoff by switching to an action different from the proposed one. Note that a Nash equilibrium is a special case of a correlated equilibrium, where the joint distribution q over the action space $\mathcal{A}$ is the product of independent distributions for each player.

2.2 Extensive-Form Games

A normal-form game as defined in 2.1, however general, is a static representation of a game, while in fact many multi-agent interactions are inherently

dynamic. An *extensive-form game* is a different model aimed at grasping the more sequential nature of various games.

A game which is said to be *extensive* is canonically represented as a directed tree. Non-terminal nodes of the tree correspond to *states*, each of which has an associated acting player. They can be divided into *chance nodes* where a *chance player* viewed as an inherent non-determinism of the game determines the state transitions, and *decisions nodes* where regular players choose their actions. In the leaves of the tree otherwise called *terminal nodes*, the payoff for each of the players is defined. A path starting at the root and ending at a leaf encodes a full play.

At the core of extensive-form games lies a notion of *information sets* grouping together game states which are indistinguishable from the acting player's perspective. Information sets amount to potentially limited knowledge of a player on the full state of the environment or other players' private information. Now we provide a formal definition of extensive-form games coherent with [Zinkevich et al., 2008]:

Definition 2.9 (extensive-form game)**.** A (finite) **extensive-form game** is a tuple $G = (\mathcal{N}, \mathcal{A}, \mathcal{H}, \mathcal{Z}, \tau, u, \sigma_c, \mathcal{I})$, where:

- $\mathcal{N} = \{1, \ldots, n\}$ is a finite set of **players**;
- $\mathcal{H}$ is a finite set of possible **histories** of actions, such that:
  - the empty sequence ε is in $\mathcal{H}$,
  - every prefix $h' \sqsubseteq h$ of a sequence $h \in \mathcal{H}$ is also in $\mathcal{H}$,
  - $\mathcal{Z} \subset \mathcal{H}$ is a set of **terminal histories** (not being prefixes of any other history in $\mathcal{H}$),
  - $\mathcal{A}(h) = \{a : ha \in \mathcal{H}\}$ is a set of **actions** available after a non-terminal history h;
- $\tau : \mathcal{H} \setminus Z \to N \cup \{c\}$ is a **player function** assigning to each non-terminal history an acting player, where c denotes the **chance player**;
- $u = (u_1, \ldots, u_n)$ and $u_i : Z \to \mathbb{R}$ is a **utility function** of player i. By $\Delta_{u,i} = \max_{z \in Z} u_i(z) - \min_{z \in Z} u_i(z)$ we denote the range of utilities of player i;
- σ_c is a function that associates with every history h for which $\tau(h) = c$ a probability measure on $\mathcal{A}(h)$. The notation $\sigma_c(h, a)$ corresponds to the probability of an action a occurring given h;

- $\mathcal{I} = (\mathcal{I}_1, \ldots, \mathcal{I}_n)$ and $\mathcal{I}_i$ for $i \in \mathcal{N}$ is an **information partition** of $\{h \in \mathcal{H} : \tau(h) = i\}$ with the property that $\mathcal{A}(h) = \mathcal{A}(h')$ whenever h and h' are in the same member of the partition. For each **information set** $I \in \mathcal{I}_i$ and any $h \in I$ we denote $\mathcal{A}(I) = \mathcal{A}(h)$ and $\tau(I) = \tau(h)$. For notational convenience, we refer to $I(h)$ as the information state that contains h.

If all information sets of every player's partition $\mathcal{I}_i$ are singletons, the game has **perfect information**. Any game without perfect information has **imperfect information**.

We should note that an arbitrary partitioning as described above may lead to situations where a player forgets their past decisions. This would happen when a player after taking some actions would end up in the same information state. Further in this work we will consider only games with all players having *perfect recall*:

Definition 2.10 (perfect recall)**.** Player i has **perfect recall** in a game $(\mathcal{N}, \mathcal{A}, \mathcal{H}, \mathcal{Z}, \tau, u, \sigma_c, \mathcal{I})$ if for any two members h, h' of the same partition set $I \in \mathcal{I}_i$:

- $|h| = |h'|$, where $|h|$ refers to the length of the history h,
- for any two prefix histories $la \sqsubseteq h$ and $l'a' \sqsubseteq h'$ such that $|l| = |l'|$ and $\tau(l) = \tau(l') = i$, their corresponding information sets and actions are equal, that is $I(l) = I(l')$ and $a = a'$.

The game is said to be of perfect recall if every player has perfect recall in it.

Perfect recall means that a player can always distinguish between game states where they previously took a different action or previously visited a different information set. However, this also implies that a player cannot get additional information about their position in an information set by remembering their earlier moves.

Various terms related to normal-form games have their counterparts in the realm of extensive-form games. By straightforward analogy to definition 2.2 we say that an extensive-form game is zero-sum when the sum of the players' payoffs is equal to zero for all terminal states.

Similarly to normal-form games, but in a slightly more involved fashion due to the tree-like structure of extensive-form games as well as their potential imperfect information, we can define strategies of the players.

Definition 2.11 ((behavioural) strategy)**.** A (**behavioural**) **strategy** of a player $i \in \mathcal{N}$ in an extensive-form game $(\mathcal{N}, \mathcal{A}, \mathcal{H}, \mathcal{Z}, \tau, u, \sigma_c, \mathcal{I})$ is a function $\sigma_i : \mathcal{I}_i \to \Delta(\mathcal{A}(\mathcal{I}_i))$ such that $\sigma_i(I) \in \Delta(\mathcal{A}(I))$ for every $I \in \mathcal{I}_i$, where $\mathcal{A}(\mathcal{I}_i) = \cup_{I \in I_i} \mathcal{A}(I)$ and $\Delta(X)$ denotes the set of probability distributions over X. We say that a strategy σ_i is **pure** when for every $I \in \mathcal{I}_i$ player i chooses actions deterministically. Additionally:

- Σ_i is the set of all strategies for player i,
- $\sigma = (\sigma_1, \ldots, \sigma_n)$ is a **strategy profile**.

Given the above, by writing $\sigma_i(I, a)$ for $I \in \mathcal{I}_i$ we refer to the probability of player i choosing an action $a \in \mathcal{A}(I)$. Furthermore, we use the shorthand $\sigma_i(h, a)$ for $\sigma_i(I(h), a)$.

Having introduced a strategy we can now define *reach probability.*

Definition 2.12 (reach probability)**.** Let $(\mathcal{N}, \mathcal{A}, \mathcal{H}, \mathcal{Z}, \tau, u, \sigma_c, \mathcal{I})$ be an extensive-form game and σ be a strategy profile. We define the **reach probability** of a history $h \in \mathcal{H}$ as

$$\pi^\sigma(h) = \prod_{h' a \sqsubseteq h} \sigma_{\tau(h')}(h', a),$$

which is the product of the probabilities of all actions leading up to h. Equivalently $\pi^\sigma(h) = \pi_i^\sigma(h)\pi_{-i}^\sigma(h)$, where the reach probability is decomposed into the contribution of player i and the contributions of the opponents of player i including the chance player.

For $I \in \cup_{i \in \mathcal{N}} \mathcal{I}_i$ the value of $\pi^\sigma(I) = \sum_{h \in I} \pi^\sigma(h)$ refers to the probability of reaching a particular information set under a strategy profile σ.

We also define a shorthand $\pi^\sigma(h, z)$ as $\prod_{h' a \sqsubseteq z : h \sqsubseteq h'} \sigma_{\tau(h')}(h', a)$ when $h \sqsubseteq z$ and 0 otherwise.

Given a strategy profile σ of an extensive-from game G the *expected utility* is defined as follows:

Definition 2.13 (expected utility)**.** The **expected utility** for player i is defined as:

$$u_i(\sigma) = \sum_{h \in Z} u_i(h)\pi^\sigma(h).$$

For the sake of brevity we do not provide the definition for a **Nash equilibrium** in an extensive-form game, because its contents would be identical to 2.6 except for the game definition itself.

There are more parallels between normal-form and extensive-form games. As a matter of fact, we could imagine a program transcribing the latter to the former by tabulating all pure strategies of every player. As a result an action of player i in the new setting would map to a sequence of actions in the original game, each of which corresponds to some unique information set I for every $I \in \mathcal{I}_i$. That is, a single action in the normal-form game would determine decisions to take at every information set of the player. We should note that this procedure would result in an exponential blowup in the size of the game representation.

We can see that while in the original game a player's behavioural strategy may be viewed as a vector of distributions, a mixed strategy in its normal-form counterpart is essentially a distribution over vectors. Although in general settings the expressive power of related behavioural and mixed strategies might be incomparable, there is a special case relevant to this work in which they coincide. Quoting [Shoham and Leyton-Brown, 2008]:

Theorem 2.14 ([Kuhn, 1953])**.** *In a game of perfect recall, any mixed strategy of a given agent can be replaced by an equivalent behavioural strategy, and any behavioural strategy can be replaced by an equivalent mixed strategy. Here two strategies are equivalent in the sense that they induce the same probabilities on outcomes, for any fixed strategy profile (mixed or behavioural) of the remaining agents.*

Recalling the Nash theorem 2.7 we immediately arrive at the following corollary:

Corollary 2.15. *Any finite extensive-form game has a Nash equilibrium of behavioural strategies.*

2.3 Complexity of Finding Equilibria

We could see in the first section that the knowledge of Nash equilibria might be desired in various competitive scenarios. Unfortunately even though the Nash theorem 2.7 asserts their existence, there is likely no polynomial-time algorithm for finding such strategy profiles in general case. In this section we outline the computational complexity of this particular problem.

The difficulty of determining Nash equilibria of games stems from the fact that this is a complete problem in the **PPAD** (Polynomial Parity Argument Directed) complexity class [Nisan et al., 2007]. PPAD is the class of all problems whose solving is akin to finding a sink or a non-standard source of a directed graph with an exponentially large set of vertices. Each vertex of the input graph has in-degree and out-degree of at most one and the

information on the incoming or outgoing edge is given by a polynomial-time computable function. The search starts at a known source called the "standard source". Despite the fact that it is always guaranteed that a solution exists, the problems of this class are believed to be hard, since no polynomial-time solution has been found. The PPAD class is in fact a subset of a more general class called TFNP, which contains total function problems solvable in non-deterministic polynomial-time.

We need to make a very important distinction for the unique case of two-player zero-sum games. Nash equilibria for such games can be calculated efficiently using linear programming.

It is worth mentioning that the problem of computing correlated equilibria (see definition 2.8) is significantly easier. Correlated equilibria form a convex set and as such can be found in polynomial time. That is under the condition that games are defined compactly via their payoff matrices.

3 Solving Extensive-Form Imperfect Information Games

In following chapter we focus on a family of recently developed algorithms for tackling imperfect information games. All of the described approaches are based on a common core, which is the **Counterfactual Regret Minimization (CFR)** algorithm [Zinkevich et al., 2008]. This procedure proved to be very successful at finding approximate Nash equilibria (see definition 2.6) in very large two-player zero-sum extensive games (with more than 10^{12} states). Despite the shortcomings of Nash equilibria outside the realm of two-player zero-sum games and the lack of different theoretical models for solving games with more than two players, empirical studies show the effectiveness of CFR for six-player poker games. The algorithm when used as a workhorse behind the AI [Brown and Sandholm, 2019] consistently wins against elite human players.

In this chapter we will give a thorough overview of the CFR algorithm and its derivatives. We will start by introducing the concept of regret minimization and its relation to competitive environments. Then we will focus our attention on the vanilla version of the CFR algorithm. The subsequent sections will be devoted to addressing some of the problems of this algorithm by using Monte-Carlo methods and function approximation.

The structure of this chapter is aimed at presenting the train of thought which led to the main contribution of this work.

3.1 Regret Minimization

Before we dig into the internals of the Counterfactual Regret Minimization we need to consider the problem of *regret* itself and what it means to have a *no-regret* learning algorithm.

Let us assume an adversarial online model, where an agent repeatedly makes decisions under uncertain environment. An agent has at its disposal a set A of n available **actions** and at each time step t uses an algorithm H to select a probability distribution p^t over A. Then the environment responds by revealing the outcome in the form of a bounded **reward vector**, so that, without loss of generality, $u^t : A \to [-1, 1]$. Finally, an action a^t is drawn from p^t and the algorithm receives a reward $u^t(a^t)$.

The end goal of the agent is to devise an algorithm H which minimizes the expected regret for not choosing a single best action at all time steps. Formally:

Definition 3.1 (regret, no-regret algorithm)**.** Assume the online decision-making problem as described above. The (**average**) **regret** of an algorithm H after T time steps with respect to an action a is defined as:

$$R_H^T(a) = \frac{1}{T}\sum_{t=1}^{T}\left(u^t(a) - u^t(a^t)\right).$$

We say that an online algorithm H is **no-regret** (or has no regret), when for every reward sequence $u^1, \ldots, u^T$ and for every $a \in A$, the **expected regret** with respect to a converges to 0 as T goes to infinity, that is:

$$\lim_{T\to\infty} \mathbb{E}\left[R_H^T(a)\right] = \lim_{T\to\infty} \frac{1}{T}\sum_{t=1}^{T}\left(u^t(a) - \mathbb{E}_{a^t\sim p^t}\left[u^t(a^t)\right]\right) = 0.$$

In other words, algorithm H learns without regret if in the limit no single action in A receives higher rewards than H.

This model can be applied to a broader context of game theory. Let us consider a scenario in which n players are repeatedly playing a game as defined in 2.1. At each step every player chooses their mixed strategy for action selection according to some online algorithm. Then after all actions a^t are sampled and played according to a strategy profile σ^t, each player i receives a reward $u_i(a_i^t, a_{-i}^t)$. A player i knowing the decisions of the opponents a_{-i}^t, can calculate the payoffs that would have been achieved for actions that have not been played.

So far we have been comparing the performance of an adaptive algorithm with the best single action in hindsight. However, in the context of repeatedly played games it would be more natural to use the best possible mixed strategy to view the overall regret. For this purpose we define an *average overall regret* [Zinkevich et al., 2008]:

Definition 3.2 (average overall regret)**.** Consider a repeatedly played game G (in normal or extensive form). Let σ_i^t be a strategy used by player i at time t. The **average overall regret** of player i after T rounds is:

$$R_i^T = \frac{1}{T}\max_{\sigma_i^*\in\Sigma_i}\sum_{t=1}^{T}\left(u_i(\sigma_i^*, \sigma_{-i}^t) - u_i(\sigma^t)\right).$$

In fact, the notion of a no-regret algorithm in the sense of average overall regret is equivalent to the one defined in 3.1. This is because the maximum in the formula above is always realized by a pure strategy of player i, thanks to the linearity of the $u_i(\cdot, \sigma_{-i}^t)$ functions.

We see that each player by incorporating an algorithm for reducing their regret could iteratively adapt their strategy. It turns out that applying no-regret adaptive algorithms in two-player zero-sum games leads to Nash equilibria.

Theorem 3.3. *Let $G = (\mathcal{N}, \mathcal{A}, u)$ be a two-player zero-sum game and assume that for T time steps both players follow an algorithm with expected regret at most ϵ. Then their **average strategies** $\bar{\sigma}$, where $\bar{\sigma}_i = \frac{1}{T}\sum_{t=1}^{T}\sigma_i^t$, form a 2ϵ-Nash equilibrium.*

Proof. Knowing the fact that player 1's regret is bounded by ϵ we can adapt definition 3.1 to express their expected regret with respect to an arbitrary action $a_1 \in \mathcal{A}_1$:

$$\frac{1}{T}\sum_{t=1}^{T}\left(\mathbb{E}_{a_2^t\sim\sigma_2^t}\left[u_1(a_1, a_2^t)\right] - \mathbb{E}_{(a_1^t,a_2^t)\sim(\sigma_1^t,\sigma_2^t)}\left[u_1(a_1^t, a_2^t)\right]\right) \leq \epsilon.$$

Assume an arbitrary mixed strategy $\sigma_1^* \in \Sigma_1$. Multiplying the inequalities for every action a_1 in A_1 by the corresponding $\sigma_1^*(a_1)$ and summing them together, we arrive at:

$$\frac{1}{T}\sum_{t=1}^{T}\left(\sum_{a_1\in\mathcal{A}_1}\sigma_1^*(a_1)\mathbb{E}_{a_2^t\sim\sigma_2^t}\left[u_1(a_1, a_2^t)\right] - \mathbb{E}_{(a_1^t,a_2^t)\sim(\sigma_1^t,\sigma_2^t)}\left[u_1(a_1^t, a_2^t)\right]\right) \leq \epsilon,$$

which using the definition 2.4 of expected utility can be rewritten as

$$\frac{1}{T}\sum_{t=1}^{T}\left(u_1(\sigma_1^*, \sigma_2^t) - u_1(\sigma_1^t, \sigma_2^t)\right) \leq \epsilon.$$

Similarly for player 2 we get that for an arbitrary $\sigma_2^* \in \Sigma_2$ the following inequality holds:

$$\frac{1}{T}\sum_{t=1}^{T}\left(u_2(\sigma_1^t, \sigma_2^*) - u_2(\sigma_1^t, \sigma_2^t)\right) \leq \epsilon.$$

Summing the inequalities for both players and applying the fact that in the case of a zero-sum game $u_1(\sigma) = -u_2(\sigma)$ for every $\sigma \in \Sigma$, we can write:

$$\frac{1}{T}\sum_{t=1}^{T}\left(u_1(\sigma_1^*, \sigma_2^t) + u_2(\sigma_1^t, \sigma_2^*)\right) \leq 2\epsilon.$$

Let us focus first on player 1. Remembering that $u_2(\sigma_1^t, \sigma_2^*) = -u_1(\sigma_1^t, \sigma_2^*)$ and applying basic transformations we see that:

$$\frac{1}{T}\sum_{t=1}^{T} u_1(\sigma_1^t, \sigma_2^*) + 2\epsilon \geq \frac{1}{T}\sum_{t=1}^{T} u_1(\sigma_1^*, \sigma_2^t).$$

Since σ_2^* is arbitrary we choose $\sigma_2^* := \bar{\sigma}_2$. Then we substitute the definitions of $\bar{\sigma}_1$ and $\bar{\sigma}_2$ on the left and the right sides of the inequality accordingly. Finally we get that for player 1 and for every $\sigma_1^* \in \Sigma_1$ the following is true:

$$u_1(\bar{\sigma}_1, \bar{\sigma}_2) + 2\epsilon \geq u_1(\sigma_1^*, \bar{\sigma}_2).$$

The very same reasoning can be followed for player 2. Thus, for every $\sigma_2^* \in \Sigma_2$:

$$u_2(\bar{\sigma}_1, \bar{\sigma}_2) + 2\epsilon \geq u_2(\bar{\sigma}_1, \sigma_2^*),$$

which completes the proof. □

Unfortunately it is not feasible to extrapolate this property to the general case, since similar no-regret dynamics ensure only the convergence towards correlated equilibria (2.8) [Nisan et al., 2007]. On the other hand, as shown by [Brown and Sandholm, 2019] and to some extent by this work, adaptive learning based on self-play appears to be very successful in empirical studies.

3.2 Regret Matching

The *regret matching* algorithm [Hart and Mas-Colell, 2016] is an instance of a simple but convenient no-regret procedure. Regret matching requires one to store and constantly update the values of expected regrets (see definition 3.1) for every action $a \in \mathcal{A}$. As far as the strategy update is concerned, actions at every time step are assigned probabilities in proportion to the positive regrets. In case when no action is assigned a positive regret, the probability distribution is uniform. Formally:

Definition 3.4 (regret matching)**.** Let $x^+ = \max(x, 0)$ for $x \in \mathbb{R}$. The **regret matching** algorithm H defines the probability of choosing an action $a \in \mathcal{A}$ at time $T + 1$ as:

$$p^{T+1}(a) = \begin{cases} \frac{R^{T,+}(a)}{\sum_{a' \in \mathcal{A}} R^{T,+}(a')} & \text{if } \sum_{a' \in \mathcal{A}} R^{T,+}(a') > 0, \\ \frac{1}{|\mathcal{A}|} & \text{otherwise,} \end{cases}$$

where $R^T(a) = \mathbb{E}\left[R_H^T(a)\right]$.

Regret matching has the following convergence properties [Zinkevich et al., 2008].

Theorem 3.5. *At time step T of the regret matching algorithm H:*

$$\max_{a \in \mathcal{A}} \mathbb{E}\left[R_H^T(a)\right] \leq \frac{\Delta_u \sqrt{|\mathcal{A}|}}{\sqrt{T}},$$

where Δ_u is the difference between the maximum and the minimum possible rewards.

3.3 Counterfactual Regret Minimization

Having laid out the fundamentals of game theory and regret minimization we are ready now to focus on the key algorithm for solving imperfect information games. **Counterfactual Regret Minimization** (**CFR**), which is the subject of this section, was introduced in the paper [Zinkevich et al., 2008]. The idea behind this procedure is the application of a no-regret procedure, namely the regret matching algorithm (see definition 3.4), to minimize a value called *immediate counterfactual regret* at every information set of the game, which in aggregate is the upper bound on the average overall regret (see definition 3.2). Thus, following the no-regret procedure the immediate counterfactual regret must converge to zero, which leads to a Nash equilibrium for a two-player zero-sum game (see theorem 3.3).

In order to build a more formal ground to what has been said regarding CFR, we start by providing key definitions, which will stay relevant through the rest of the work. From now on when referring to a game we assume an extensive-form game $G = (\mathcal{N}, \mathcal{A}, \mathcal{H}, \mathcal{Z}, \tau, u, \sigma_c, \mathcal{I})$ of perfect recall (see definition 2.10).

The first building block is a *value function*:

Definition 3.6. The **value function** is the expected utility of a player $i \in \mathcal{N}$ at a node $h \in H$ when all players act according to a strategy profile σ:

$$u_i(\sigma, h) = \sum_{z \in \mathcal{Z}} \pi^\sigma(h, z) u_i(z).$$

The above formula for the value function may also be expressed in a recursive manner which is more convenient for implementations:

$$u_i(\sigma, h) = \begin{cases} u_i(h) & \text{when } h \in \mathcal{Z}, \\ \sum\limits_{a \in \mathcal{A}(h)} \sigma_{\tau(h)}(h, a) u_i(\sigma, ha) & \text{otherwise.} \end{cases} \tag{3.1}$$

Then we define a measure of utility which expresses the limited information the players have.

Definition 3.7. Let $i \in \mathcal{N}$, $I \in \mathcal{I}_i$, $h \in I$ and σ be a strategy profile. The **counterfactual value function** is given as follows:

$$v_i(\sigma, h) = \pi^{\sigma}_{-i}(h) u_i(\sigma, h),$$
$$v_i(\sigma, I) = \sum_{h \in I} v_i(\sigma, h).$$

In other words $v_i(\sigma, I)$ represents the payoffs player i can expect if they play to reach the information set I while the opponents act according to their strategies σ_{-i}. Due to the assumption of perfect recall (see definition 2.10) the sequence of actions that player i takes in this setting on the path to I is the same for every $h \in I$.

Finally we are ready to define the *immediate counterfactual regret*.

Definition 3.8. Consider a player $i \in \mathcal{N}$, an information set $I \in \mathcal{I}_i$ and a history $h \in I$. For all $a \in \mathcal{A}(I)$ let $\sigma|_{I \to a}$ be the strategy profile identical to σ except that player i always chooses action a when in information set I. The **instantaneous regret** associated with an action $a \in \mathcal{A}(I)$ at a time step t is defined as:

$$r^t_i(h, a) = v_i(\sigma^t|_{I \to a}, h) - v_i(\sigma^t, h)$$
$$r^t_i(I, a) = \sum_{h \in I} r^t_i(h, a) = v_i(\sigma^t|_{I \to a}, I) - v_i(\sigma^t, I).$$

We define the **accumulated regret** as:

$$R^T_{i,\text{acc}}(I, a) = \sum_{t=1}^{T} r^t_i(I, a).$$

The **immediate counterfactual regret** after T rounds is:

$$R^T_{i,\text{imm}}(I) = \frac{1}{T} \max_{a \in \mathcal{A}(I)} R^T_{i,\text{acc}}(I, a).$$

In essence this entity models the player's regret in their decisions at the information set I in terms of values of actions at every history $h \in I$. The regret at each h is weighted by the probability that it would have been visited if the player had always played to reach it. This statement is immediately

clear after expanding the formula:

$$\begin{aligned} R^T_{i,\text{imm}}(I) &= \frac{1}{T} \max_{a \in \mathcal{A}(I)} \sum_{t=1}^{T} r_i^t(I,a) \\ &= \frac{1}{T} \max_{a \in \mathcal{A}(I)} \sum_{t=1}^{T} \left(v_i(\sigma^t|_{I \to a}, I) - v_i(\sigma^t, I) \right) \\ &= \frac{1}{T} \max_{a \in \mathcal{A}(I)} \sum_{t=1}^{T} \sum_{h \in I} \pi^{\sigma^t}_{-i}(h) \left(u_i(\sigma^t, ha) - u_i(\sigma^t, h) \right). \end{aligned} \tag{3.2}$$

We can now provide the statement of the key theorem [Zinkevich et al., 2008] on the average overall regret bound (see definition 3.2).

Theorem 3.9.

$$R_i^T \leq \sum_{I \in \mathcal{I}_i} R^{T,+}_{i,\text{imm}}(I).$$

This theorem implies that the average overall regret can be minimized by deploying an adaptive learning procedure independently at each information set. For this purpose, CFR utilizes the regret matching algorithm (see definition 3.4) to determine the behavioural strategy of every player $i \in \mathcal{N}$ at each of their information sets $I \in \mathcal{I}_i$:

$$\sigma_i^{T+1}(I,a) = \begin{cases} \frac{R^{T,+}_{i,\text{acc}}(I,a)}{\sum_{a' \in \mathcal{A}(I)} R^{T,+}_{i,\text{acc}}(I,a')} & \text{if } \sum_{a' \in \mathcal{A}(I)} R^{T,+}_{i,\text{acc}}(I,a') > 0, \\ \frac{1}{|\mathcal{A}(I)|} & \text{otherwise.} \end{cases} \tag{3.3}$$

Taking theorem 3.5 into consideration, we see that regret matching in the context of extensive-form games has the following convergence property [Zinkevich et al., 2008].

Corollary 3.10. *If player i selects actions according to equation* (3.3) *then* $R^T_{i,\text{imm}}(I) \leq \frac{\Delta_{u,i}\sqrt{m_i}}{\sqrt{T}}$ *and consequently* $R_i^T \leq \frac{\Delta_{u,i}|\mathcal{I}_i|\sqrt{m_i}}{\sqrt{T}}$, *where* $m_i = \max_{h:\tau(h)=i} |\mathcal{A}(h)|$.

Thanks to the relative simplicity and the lack of parameters in the regret matching algorithm, CFR [Zinkevich et al., 2008] can be implemented in a straightforward way, as follows.

We need to maintain in the memory the values of accumulated regrets $R^t_{i,\text{acc}}(I,a)$ for every player $i \in \mathcal{N}$, every information set $I \in \mathcal{I}_i$ and every action $a \in \mathcal{A}(I)$. These values are initialized to zero, that is $R^0_{i,\text{acc}}(I,a) = 0$.

The algorithm is iterative and based on self-play. For each round t starting at $t = 1$ it unfolds the entire game tree in a recursive manner and updates the regrets for all players. Before every iteration the regrets are initialized with the previous values, that is $R^t_{i,\text{acc}}(\cdot,\cdot) = R^{t-1}_{i,\text{acc}}(\cdot,\cdot)$ for every $i \in \mathcal{N}$. The steps performed at each node $h \in \mathcal{H}$ can be briefly described as follows:

- if $h \in \mathcal{Z}$ then the players' payoffs $u_i(h)$ are calculated and propagated back;
- if $\tau(h) = c$, that is h is a *chance node*, every child ha for $a \in A(h)$ is explored and the outcomes are weighted with the respective probabilities $\sigma_c(h,a)$ before finally returning $u_i(\sigma^t, h)$ (see equation 3.1);
- otherwise h is a *decision node* of player $i = \tau(h)$:
  1. the strategy $\sigma^t_i(I(h),\cdot)$ is determined using the accumulated regret $R^{t-1}_{i,\text{acc}}(I(h),\cdot)$ (see equation 3.3),
  2. each child node ha for $a \in \mathcal{A}(h)$ is recursively visited to determine the values $u_i(\sigma^t, ha)$ and consequently to calculate $u_i(\sigma^t, h)$ (see equation 3.1),
  3. the accumulated regret for information set $I(h)$ is updated, for all $a \in A(h)$ (see equation 3.2):

$$R^t_{i,\text{acc}}(I(h),a) = R^t_{i,\text{acc}}(I(h),a) + \underbrace{\pi^{\sigma^t}_{-i}(h)\left(u_i(\sigma^t, ha) - u_i(\sigma^t, h)\right)}_{r^t_i(h,a)},$$

  4. the value $u_i(\sigma^t, h)$ of the history h is passed back.

The players' strategies for every information set may be significantly altered with consecutive iterations of CFR. Even though their variability is likely to be large, it is the average strategy that leads to Nash equilibria (3.3) in two-player zero-sum games and is also empirically more robust in more general scenarios. Following [Zinkevich et al., 2008], the formula for calculating the **average strategy** for each player $i \in \mathcal{N}$ from time 1 to T with respect to an information set $I \in \mathcal{I}_i$ and an action $a \in \mathcal{A}(I)$ is given by:

$$\bar{\sigma}_i(I,a) = \frac{\sum_{t=1}^{T} \pi^{\sigma^t}_i(I)\sigma^t_i(I,a)}{\sum_{t=1}^{T} \pi^{\sigma^t}_i(I)}. \tag{3.4}$$

A more in-depth explanation of the intricacies behind CFR implementation is provided in [Neller and Lanctot, 2013].

3.4 Monte Carlo Methods for CFR

Despite no-regret guarantees and empirical effectiveness, the CFR algorithm has a few shortcomings. One of them is a very high computational cost per iteration, as the entire game tree has to be explored in order to adapt the strategies. This indeed might be prohibitive when tackling large games, but is often mitigated by various forms of action abstraction [Brown and Sandholm, 2019] to reduce the game complexity. Unfortunately those require advanced knowledge of the domain and have to be tailor-made to every game. What is more, CFR is not suitable for online environments in which the learning player has no access to the strategies of their opponents.

Monte Carlo CFR (**MCCFR**) [Lanctot et al., 2009] is a family of domain-independent CFR sample-based algorithms created to address the issues mentioned above. It is based on the idea that not all histories have to be traversed during each iteration for the immediate counterfactual regrets (see definition 3.8) to stay unchanged *in expectation*. The authors of the algorithm show empirically that even though more iterations are required than in vanilla CFR, MCCFR converges dramatically faster in various games with respect to time or the number of visited nodes.

The key to understanding MCCFR is its sampling scheme. Let $\mathcal{Q} = \{Q_1, \dots, Q_r\} \subseteq P(\mathcal{Z})$ be a set of subsets of $\mathcal{Z}$ spanning $\mathcal{Z}$. Every element of $\mathcal{Q}$ is called a **block**. Note that blocks are not necessarily disjoint. The algorithm starts each round by sampling a block according to a sampling distribution q, such that $q_j > 0$ is the probability of selecting Q_j and $\sum_{j=1}^{r} q_j = 1$. Then it considers only the histories from the chosen block to adapt the strategies. This is done in a similar fashion to CFR, but instead of calculating the counterfactual values (defined in 3.7), their sampled counterparts are used.

Definition 3.11. Let $i \in \mathcal{N}$, $I \in \mathcal{I}_i$, $h \in I$ and σ be a strategy profile. The **sampled counterfactual value** conditioned on choosing a block $Q_j \in \mathcal{Q}$ is:

$$\tilde{v}_i(\sigma, h|j) = \sum_{z \in Q_j} \frac{1}{q(z)} \pi^{\sigma}_{-i}(h) \pi^{\sigma}(h, z) u_i(z),$$

$$\tilde{v}_i(\sigma, I|j) = \sum_{h \in I} \tilde{v}_i(\sigma, h|j),$$

where $q(z) = \sum_{j: z \in Q_j} q_j$ is the probability of choosing a block with a terminal history $z \in \mathcal{Z}$.

It is readily visible that if we choose $\mathcal{Q} = \{\mathcal{Z}\}$, the sampled counterfactual values are equal to the regular counterfactual values and the algorithm reduces to CFR.

As mentioned earlier, the fundamental property of MCCFR is that the sampled counterfactual values match the counterfactual values in expectation.

Lemma 3.12. *Let $I \in \mathcal{I}_i$ be an information set of player $i \in \mathcal{N}$, σ be a strategy profile and q be a sampling distribution over $\mathcal{Q}$. Then for any $h \in I$:*

$$\mathbb{E}_{j\sim q}\left[\tilde{v}_i(\sigma, h|j)\right] = v_i(\sigma, h),$$

and consequently

$$\mathbb{E}_{j\sim q}\left[\tilde{v}_i(\sigma, I|j)\right] = v_i(\sigma, I).$$

Proof. By expanding the formula for the expectation of sampled counterfactual value (see definition 3.11), we have:

$$\begin{aligned}\mathbb{E}_{j\sim q}\left[\tilde{v}_i(\sigma, h|j)\right] &= \sum_{Q_j\in\mathcal{Q}} q_j \sum_{z\in Q_j} \frac{1}{q(z)}\pi^{\sigma}_{-i}(h)\pi^{\sigma}(h,z)u_i(z) \\ &= \sum_{z\in\mathcal{Z}} \frac{1}{q(z)}\pi^{\sigma}_{-i}(h)\pi^{\sigma}(h,z)u_i(z) \sum_{j:z\in Q_j} q_j.\end{aligned}$$

Thus, by recalling that $q(z) = \sum_{j:z\in Q_j} q_j$ we get:

$$\mathbb{E}_{j\sim q}\left[\tilde{v}_i(\sigma, h|j)\right] = \sum_{z\in\mathcal{Z}} \pi^{\sigma}_{-i}(h)\pi^{\sigma}(h,z)u_i(z) = v_i(\sigma, h).$$

Finally, thanks to the linearity of expectation we arrive at:

$$\mathbb{E}_{j\sim q}\left[\tilde{v}_i(\sigma, I|j)\right] = \sum_{h\in I} \mathbb{E}_{j\sim q}\left[\tilde{v}_i(\sigma, h|j)\right] = \sum_{h\in I} v_i(\sigma, h) = v_i(\sigma, I). \quad \square$$

This lemma implies that **sampled instantaneous regrets**

$$\tilde{r}^t_i(I, a|j) = \tilde{v}_i(\sigma^t|_{I\to a}, I|j) - \tilde{v}_i(\sigma^t, I|j)$$

can be accumulated at each iteration and used by the regret-matching algorithm to alter the strategies. Thus, MCCFR should behave in expectation like the vanilla CFR.

Various families of MCCFR algorithms are considered depending on the structure of the set $\mathcal{Q}$ and the sampling distribution q. The authors introduce two kinds of sampling schemes, namely *external sampling* and *outcome sampling*.

3.4.1 External-Sampling MCCFR

We briefly describe the external-sampling procedure, as it is not used in the algorithm proposed by this work. Nevertheless, it is the core method behind the *Deep CFR* algorithm [Brown et al., 2018], which was the starting point to this work. We will elaborate on Deep CFR later.

In external sampling the blocks are associated with pure the strategies of the opponents $-i$ (including the chance player c) for some player $i \in \mathcal{N}$. In other words a block $Q_{\alpha_i} \in \mathcal{Q}$ is defined by a deterministic mapping α_i from $h \in \mathcal{H}_c = \{h \in \mathcal{H} : \tau(h) = c\}$ to $\mathcal{A}(h)$ and from $I \in \mathcal{I}_{-i} = \cup_{j \in \mathcal{N} \setminus \{i\}} \mathcal{I}_j$ to $\mathcal{A}(I)$. The corresponding sampling probabilities are decided based on σ_{-i}, that is:

$$q_{\alpha_i} = \left(\Pi_{h \in \mathcal{H}_c} \sigma_c(h, \alpha_i(h))\right) \left(\Pi_{I \in \mathcal{I}_{-i}} \sigma_{\tau(I)}(I, \alpha_i(I))\right).$$

In this setting the block Q_{α_i} contains all terminal histories consistent with the mapping α_i, which means that for every $z \in Q_{\alpha_i}$ if $ha \sqsubseteq z$ and $\tau(h) \neq i$ then $\alpha_i(I(h)) = a$ if $\tau(I) \neq c$ and $\alpha_i(h) = a$ otherwise.

A practical implementation of the external-sampling MCCFR algorithm implicitly constructs the mapping α_i in a randomized way. In fact during each round it iterates over players $i \in \mathcal{N}$ and for each of them it traverses the game tree individually. When player i is the *traversing* player, the actions are sampled at histories $h \in \mathcal{H}$ such that $\tau(h) \neq i$. The actions chosen at the histories h from the opponents' information sets $\mathcal{I}_{-i}$ are memorized, so they could be used in the same iteration during the visits to $h' \in \mathcal{H}$ where $I(h') = I(h)$. Whenever player i's decision node is entered, the algorithm performs a post-order traversal over all available actions.

3.4.2 Outcome-Sampling MCCFR

Outcome sampling is a more natural approach to constructing an MCCFR algorithm, because each $Q \in \mathcal{Q}$ amounts to a single terminal history, which can be viewed as an *episode* of the game, so $\mathcal{Q} = \{\{z\} : z \in \mathcal{Z}\}$. In this setup q defines a probability distribution over the terminal histories. The distribution q is given by a *sampling profile* σ' , so that $q(z) = \pi^{\sigma'}(z)$. The sampling profile is treated as an alternative strategy profile used to decide subsequent nodes during game tree traversal (the actions in chance nodes are always sampled from σ_c).

Furthermore, to ensure that the formula for sampled counterfactual value defined in 3.11 is well-founded, we require that $\sigma'_i(I, a) > \epsilon$ for all $i \in \mathcal{N}$, $I \in \mathcal{I}_i$, $a \in \mathcal{A}(I)$. This guarantees that $q(z) > \delta$ for some $\delta > 0$.

In order to make the difference between σ and σ' more clear, we will make use of the fact that q is fully defined by a profile σ'. We overload the symbol

q by treating it as a sampling strategy profile σ', so when writing $q(h, a)$ we will actually refer to $\sigma'_{\tau(h)}(h, a)$. This notation will stay relevant through the rest of this dissertation.

Now we derive the formula for sampled instantaneous regrets in Outcome-Sampling MCCFR. Since there is a one-to-one mapping between the blocks $Q_j \in \mathcal{Q}$ and terminal the histories $z \in \mathcal{Z}$, we will be conditioning on z in place of index j in notation like $\tilde{v}_i(\sigma, h|z)$ etc. Adjusting the definition 3.11 to Outcome-Sampling we get:

$$\tilde{v}_i(\sigma, h|z) = \frac{1}{\pi^q(z)} \pi^\sigma_{-i}(h)\pi^\sigma(h, z)u_i(z). \tag{3.5}$$

It is clear that when h is not a prefix of z the value above is equal to 0. What is more, the formula $\tilde{v}_i(\sigma, I|z) = \sum_{h \in I} \tilde{v}_i(\sigma, h|z)$ for $I \in \mathcal{I}_i$ has at most one non-zero summand when I contains a prefix of z. Thus, when $h \sqsubseteq z$ we can simply write $\tilde{v}_i(\sigma, I(h)|z) = \tilde{v}_i(\sigma, h|z)$.

In what follows we assume that $h \sqsubseteq z$, as in the opposite case all formulas are trivially equal to 0.

Let us now consider the sampled counterfactual value of a modified strategy. For $a \in \mathcal{A}(h)$ we get:

$$\tilde{v}_i(\sigma_{|I \to a}, h|z) = \frac{1}{\pi^q(z)} \pi^\sigma_{-i}(h)\pi^\sigma(ha, z)u_i(z) \tag{3.6}$$

which is equal to 0 when ha is not a prefix of z.

Let a' be the next action after history h on the path z. We recall that $\pi^\sigma(h, z) = \sigma_i(h, a')\pi^\sigma(ha', z)$. Finally, by combining the equations (3.5) and (3.6) we arrive at the following formula for sampled instantaneous regret:

$$\tilde{r}_i(I(h), a|z) = \tilde{r}_i(h, a|z) = (\mathbf{1}[a = a'] - \sigma_i(h, a')) \frac{\pi^\sigma_{-i}(h)\pi^\sigma(ha', z)u_i(z)}{\pi^q(z)}. \tag{3.7}$$

An iteration of Outcome-Sampling MCCFR algorithm can be broken into several steps:

1. A single terminal history $z \in \mathcal{Z}$ is chosen using the sampling profile q and the corresponding payoffs $u(z)$ are calculated.

2. The episode z is traversed forward computing the reach probabilities $\pi^\sigma_i(h)$ of each player for every prefix $h \sqsubseteq z$, where σ is determined by the regret-matching formula.

3. The history z is traversed backwards to calculate the reach probabilities of the remainders, that is $\pi^\sigma_i(h, z)$ for $h \sqsubseteq z$. While visiting h the

sampled counterfactual regrets $\tilde{r}_{\tau(h)}\left(I\left(h\right),\cdot|z\right)$ are computed using the formula (3.7) and added to accumulated regrets.

A very desirable property of Outcome-Sampling MCCFR is that it is suitable for *online learning*, where player i does not have access to σ_{-i}. In such cases episodes are sampled according the to the opponents' strategies, so that $q_{-i} = \sigma_{-i}$. Then the learning player has only control over the randomization strategy q_i. Moreover, q_i is ideally as close to σ_i as possible, but at the same time it should guarantee some degree of exploration, so that $q(z) \geq \delta > 0$ for every terminal history z for which $\pi^{\sigma}_{-i}(z) \neq 0$. Then the regret formula (3.7) in online learning reduces to:

$$\tilde{r}_i(h, a|z) = (\mathbf{1}[a = a'] - \sigma_i(h, a'))\,\frac{\pi^{\sigma}_i(ha', z)u_i(z)}{\pi^{q}_i(z)},$$

and in order to be computed it requires only information that player i has access to.

As far as the convergence property of this sampling scheme are concerned, the authors [Lanctot et al., 2009] prove that with the probability of $p \in [0, 1)$ the upper bound on the average overall regret is greater by a factor of $(1 + \frac{\sqrt{2}}{\sqrt{1-p}})\frac{1}{\delta}$ than for the vanilla CFR (see corollary 3.10).

3.4.3 Variance Reduction for Outcome-Sampling MCCFR

MCCFR and other Monte-Carlo based methods utilize the property that some easily calculable values are equal in expectation to the reference variables, therefore some degree of accuracy may be attained through sufficient sampling. Unfortunately MCCFR may still suffer from slow convergence rates in various games due to high variance in the estimates of counterfactual values. The issues of Outcome-Sampling MCCFR are mitigated by the algorithm **VR-MCCFR** (VR stands for Variance Reduced) [Schmid et al., 2018], [Davis et al., 2019] which introduces *baseline-enhanced* estimates of the counterfactual values.

The essence of VR-MCCFR is the utilization of a **control variate** to lower the variance of the estimates of the counterfactual values in outcome sampling. A control variate can be briefly described as follows. Consider a random variable X which is an unbiased estimator of some target value x, that is $\mathbb{E}[X] = x$. A control variate is a random variable Y with a known expectation $\mu_Y = \mathbb{E}[Y]$ that is coupled with X. It is used to create another random variable

$$Z = X - c(Y - \mu_Y). \tag{3.8}$$

Given that $\mathbb{E}[Z] = \mathbb{E}[X]$, the variable Z can be used instead of X. The variance of Z which is $\mathbb{V}\text{ar}[Z] = \mathbb{V}\text{ar}[X] + c^2\mathbb{V}\text{ar}[Y] - 2c\mathbb{C}\text{ov}[X, Y]$, is lower than that of X when $c > 0$ and $\mathbb{C}\text{ov}[X, Y] > \frac{c}{2}\mathbb{V}\text{ar}[Y]$. This makes Z a better estimator of x in cases when Y is sufficiently correlated with X.

Now we present a general overview on how VR-MCCFR estimates the counterfactual values. The core of the improved Outcome-Sampling MCCFR algorithm is the extension to the sampled counterfactual value $\tilde{v}_i(I, h|z)$ with a *baseline* function $b_i(I, a)$ and its estimator called a *sampled baseline* function $\hat{b}_i(I, a)$, such that $\mathbb{E}[\hat{b}_i(I, a)] = b_i(I, a)$. The sampled baseline function plays the role of a control variate in the estimation of $v_i(I, a)$. The resulting *baseline-enhanced sampled counterfactual value* $\tilde{v}_i^b(I, a|z)$ exhibits lower variance than its non-enhanced counterpart, while still being equal in expectation to the counterfactual value.

Another strength of the VR-MCCFR algorithm is a method called *recursive bootstraping*. The intuition behind it is the utilization of baseline-enhanced estimates during the backward pass of an iteration. That is, the estimates of counterfactual values at a history h are not only improved by the baseline at the corresponding level, but also by the *baseline-enhanced sampled values* returned from the recursive call to $ha \sqsubseteq z$, where $z \in \mathcal{Z}$.

Before we move to formal definitions we introduce a very useful extension to information sets called **augmented information sets**. In contrast to the information sets defined in 2.9 their augmented counterparts also group the indistinguishable histories h for player i where $\tau(h) \neq i$. For a history h we denote an augmented information set of player i as $I_i(h)$. We define $\hat{\mathcal{I}}_i$ to be the set of all augmented information sets of player i. We note that if $\tau(h) = i$ then $I_i(h) = I(h)$ and $I(h) = I_{\tau(h)}(h)$. The structure of the augmented information sets should be viewed as an inherent part of the game definition itself that is not derivable from the regular information sets.

We are now ready to provide the key definitions, which bring out the recursive nature of VR-MCCFR. We assume a sampling strategy profile $q \in \Sigma$ used to collect a terminal history $z \in \mathcal{Z}$, so that $z \sim \pi^q$.

Definition 3.13. The **sampled value** for a player $i \in \mathcal{N}$, a strategy profile σ and a history $h \in \mathcal{H}$ is defined recursively:

$$\hat{u}_i(\sigma, h, a|z) = \frac{\mathbf{1}[ha \sqsubseteq z]}{q(h, a)} \hat{u}_i(\sigma, ha|z)$$

and

$$\hat{u}_i(\sigma, h|z) = \begin{cases} u_i(z) & \text{if } h = z, \\ \sum_{a \in \mathcal{A}(h)} \sigma_{\tau(h)}(h, a)\hat{u}_i(\sigma, h, a|z) & \text{if } h \sqsubset z, \\ 0 & \text{otherwise.} \end{cases}$$

The sampled counterfactual value given by the equations (3.5) and (3.6) can now be reformulated as:

$$\tilde{v}_i(\sigma, h|z) = \frac{\pi^\sigma_{-i}(h)}{\pi^q(h)}\hat{u}_i(\sigma, h|z),$$
$$\tilde{v}_i(\sigma|_{I(h)\to a}, h|z) = \frac{\pi^\sigma_{-i}(h)}{\pi^q(h)}\hat{u}_i(\sigma, h, a|z).$$

Nonetheless, we are interested in improving upon $\tilde{v}_i$, by using the baselines.

Definition 3.14. Assume the existence of a **baseline** function $b_i(I, a)$ which is meant to approximate $u_i(\sigma, ha)$ for $I \in \hat{\mathcal{I}}_i$ and $h \in I_i$. Then the **baseline-enhanced sampled value** is defined as:

$$\hat{u}^b_i(\sigma, h, a|z) = \frac{\mathbf{1}[ha \sqsubseteq z]}{q(h, a)}\left(\hat{u}^b_i(\sigma, ha|z) - b_i(I_i(h), a)\right) + b_i(I_i(h), a) \tag{3.9}$$

when $h \sqsubset z$, and

$$\hat{u}^b_i(\sigma, h|z) = \begin{cases} u_i(z) & \text{if } h = z, \\ \sum\limits_{a\in\mathcal{A}(h)} \sigma_{\tau(h)}(h, a)\hat{u}^b_i(\sigma, h, a|z) & \text{if } h \sqsubset z, \\ 0 & \text{otherwise.} \end{cases}$$

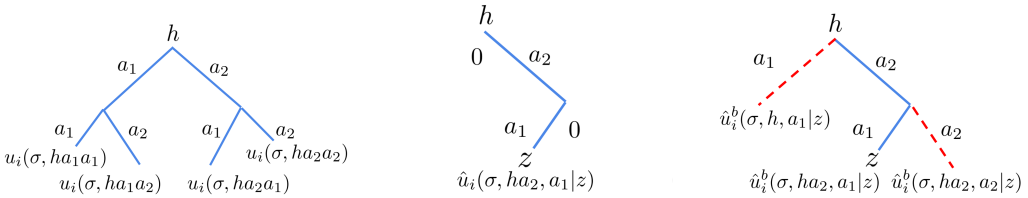
In the equation (3.9) above one could discern the application of a control variate with $c = 1$, where:

$$X = \frac{\mathbf{1}[ha \sqsubseteq z]}{q(h, a)}\hat{u}^b_i(\sigma, ha|z),$$
$$Y = \frac{\mathbf{1}[ha \sqsubseteq z]}{q(h, a)}b_i(I_i(h), a),$$
$$\mathbb{E}[Y] = b_i(I_i(h), a).$$

Finally, we can provide the concrete definition of $\tilde{v}^b_i$ which is the key to the refinement of regret estimates.

Definition 3.15. The **baseline-enhanced sampled counterfactual value** for a player $i \in \mathcal{N}$, a strategy profile σ, an information set $I \in \mathcal{I}_i$ and a history $h \in \mathcal{H}$ is defined as:

$$\tilde{v}^b_i(\sigma, h|z) = \frac{\pi^\sigma_{-i}(h)}{\pi^q(h)}\hat{u}^b_i(\sigma, h|z),$$
$$\tilde{v}^b_i(\sigma, I|z) = \sum_{h\in I}\tilde{v}^b_i(\sigma, h|z).$$



(a) CFR (b) MCCFR (c) VR-MCCFR

Figure 1: Differences in the methods for calculating values. CFR calculates the exact values, while Outcome-Samplng MCCFR and VR-MCCFR move along only a single episode to compute estimates. VR-MCCFR utilizes baseline-enhanced values for the off-trajectory actions. The figure is borrowed from [Schmid et al., 2018]

The authors [Schmid et al., 2018] show that $\tilde{v}_i^b$ is indeed an unbiased estimator of v_i:

Lemma 3.16. *For any player $i \in \mathcal{N}$, σ_i, $I \in \mathcal{I}_i$, $h \in I$, $a \in \mathcal{A}(I)$, it holds that $\mathbb{E}_{z\sim q}[\tilde{v}_i^b(\sigma, h|z)] = v_i(\sigma, h)$ and $\mathbb{E}_{z\sim q}[\tilde{v}_i^b(\sigma, I|z)] = v_i(\sigma, I)$.*

This property is crucial to the construction of the **baseline-enhanced sampled instantaneous regrets** which are ultimately used to adapt strategies of the players:

$$\begin{aligned}\tilde{r}_i^b(h, a|z) &= \tilde{v}_i^b(\sigma|_{I(h)\to a}, h|z) - \tilde{v}_i^b(\sigma, h|z) \\ &= \frac{\pi_{-i}^{\sigma}(h)}{\pi^q(h)} \left(\hat{u}_i^b(\sigma, h, a|z) - \hat{u}_i^b(\sigma, h|z)\right). \end{aligned} \tag{3.10}$$

Even though the VR-MCCFR algorithm stills samples a single history z, it can also utilize the information returned by the baseline $b_i(I_i(h), a')$ at the off-trajectory paths ha', that is when $ha \sqsubseteq z$ and $a' \neq a$. This stands in contrast to Outcome-Sampling MCCFR, for which $\tilde{v}_i(\sigma|_{I(h)\to a'}, h|z) = 0$. The intuition behind the key differences between CFR, Outcome-Sampling MCCFR and VR-MCCFR is depicted in figure 1.

The authors of the method [Schmid et al., 2018] use the exponentially-decaying averages of the baseline-enhanced values $\hat{u}_i^b$ as their baseline function. These averages are tracked in memory for every augmented information set I and updated on each visit to $h \in I$. Thus, the baseline assigns more weight to the more recent samples. This construction of the baseline proved to be very effective in practice. The authors report an overall exponential improvement over the Outcome-Sampling MCCFR in terms of convergence rates.

3.5 Function Approximation Methods

CFR and its sampling-based derivatives have several weak spots which are evident in larger games.

Firstly they exhibit very expensive memory requirements, as they need to store accumulated regrets at each information set of the game with respect to every available action. Unfortunately this might be infeasible in games like Poker, in case of which researches resort to devising complex abstractions [Brown and Sandholm, 2019]. The abstractions are based on deep domain expertise and are very time-consuming to invent. What is worse, they do not generalize between games.

Secondly the algorithms do not utilize the experience gathered at "similar" information states visited in the past. This means that the algorithm always has to explore more to build up reliable statistics for every node.

The approaches based on function approximation are aimed at addressing these problems. Instead of calculating and storing information explicitly in memory, we rely on parameterized functions (usually differentiable) rather cheap to compute. Such functions, when given an encoded representation of the state, return approximations of some desired values. We can perceive them as "smooth" versions of abstraction, since they essentially perform some sort of internal problem compression. Such encoding is implicit and does not require almost any domain knowledge. In addition to that, they tend to generalize well to previously unseen inputs.

Neural Networks [Theodoridis, 2015a], [Goodfellow et al., 2016] have achieved tremendous success when used as function approximators in the context of Reinforcement Learning [Sutton and Barto, 2018]. They are the key components of AI programs which defeated the best performing human players in the games of Go [Silver et al., 2016] and StarCraft [Vinyals et al., 2019]. Recently they have been increasingly more often used in more game-theoretic settings. One of the most prominent examples crucial to this work is the **Deep CFR** algorithm.

3.5.1 Deep CFR

The Deep CFR algorithm [Brown et al., 2018] is a direct attempt at approximating the Counterfactual Regret Minimization algorithm. We omit the details behind it for brevity and instead we focus on the big picture, which is helpful to understand the motivation of our work.

Similarly to the vanilla (tabular) CFR, the algorithm iteratively improves the strategies. However, it performs a constant number of K traversals of the game using external sampling performed individually for every player i,

which is called a *traversing* player.

The central piece of Deep CFR is a neural network $V : I \to \mathbb{R}^{\mathcal{A}(I)}$ which is parameterized by a vector θ_i^{t-1}. We denote the outputs of the network by $V(I, a; \theta_i^{t-1})$. The goal of the algorithm is for $V(I, a; \theta_i^{t-1})$ to be "proportional" to the sampled accumulated regret $\tilde{R}_{i,\text{acc}}^{t-1}(I, a|\alpha_i^{t-1})$ (see definition 3.8) where α_i^{t-1} is the mapping chosen by the external sampling method. Thus, at each information set $I \in \mathcal{I}_i$ encountered, the strategy $\sigma_i^t(I, \cdot)$ is determined by regret matching (see equation 3.3) applied to the output of the neural network $V(I, \cdot; \theta_i^{t-1})$.

During a visit to a node h, Deep CFR handles the following cases:

- when a terminal node $h = z \in \mathcal{Z}$ is reached, the utility $u_i(z)$ is passed back up as $\tilde{v}_i(\sigma^t, z|\alpha_i^t)$;

- if $\tau(h) \neq i$, then an action a is sampled from $\sigma_{\tau(h)}^t(h, \cdot)$. When h is not a chance node, this decision is memorized as $\alpha_i^t(I(h))$ in order to be reused when visiting different histories $h' \in I(h)$ during the same iteration. The sampled counterfactual value $\tilde{v}_i(\sigma^t, ha|\alpha_i^t)$ returned from the recursive call is passed back as $\tilde{v}_i(\sigma^t, h|\alpha_i^t)$;

- otherwise, that is if $\tau(h) = i$, the child nodes are recursively visited to calculate $\tilde{v}_i(\sigma^t, ha|\alpha_i^t)$ for all $a \in \mathcal{A}(h)$. Then the sampled counterfactual value for the history h is expressed as:

$$\tilde{v}_i(\sigma^t, h|\alpha_i^t) = \sum_{a \in \mathcal{A}(h)} \sigma_i^t(h, a)\tilde{v}_i(\sigma^t, ha|\alpha_i^t).$$

  These results are combined to calculate the sampled instantaneous counterfactual regrets $\tilde{r}_i^t(h, a|\alpha_i^t) = \tilde{v}_i(\sigma^t, ha|\alpha_i^t) - \tilde{v}_i(\sigma^t, h|\alpha_i^t)$ for all $a \in \mathcal{A}(h)$, which are then added to the *advantage memory* $\mathcal{M}_{v,i}$. If the capacity is exceeded a *reservoir sampling* procedure is applied to it (to prune the memory). Finally, the value of $\tilde{v}_i(\sigma^t, h|\alpha_i^t)$ is returned.

Once K traversals are completed, a new network is trained from scratch to determine the parameters θ_i^t:

- samples of instantaneous regrets $\tilde{r}_i^{t'}(I, a|\alpha_i^{t'})$ for all prior iterations $t' \leq t$ are drawn from the memory $\mathcal{M}_{v,i}$;

- the *Mean Squared Error* (MSE) [Theodoridis, 2015b], [Theodoridis, 2015a] between the samples $\tilde{r}_i^{t'}(I, a|\alpha_i^{t'})$ and the predicted $V(I, a; \theta_i^t)$ values is minimized. The network trained in such a manner approximates the average regrets, which are proportional to the sampled accumulated regret $\tilde{R}_i^t(I, a)$.

In addition to the advantage memory $\mathcal{M}_{v,i}$, the algorithm also stores the memory $\mathcal{M}_{\Pi,i}$ of the played strategy vectors $\sigma_i^t(I,\cdot)$. The final step of Deep CFR is the training of a dedicated neural network $\Pi : I \to \mathbb{R}^{\mathcal{A}(I)}$ on the samples from this memory buffer. The target for the network Π is to approximate the average strategy (see theorem 3.3).

Despite mitigating some of the issues of its predecessors, the Deep CFR algorithm has its own shortcomings. It employs the external sampling scheme, which is problematic in practice for games with large branching factors. More importantly, it maintains heavy memory buffers (ideally infinite) in order to approximate the average regrets. Although the memory and computational requirements of an already trained network are rather low, the cost of the training itself is significant. This makes the algorithm hardly applicable to more complex environments.

4 Deep-Hedge MCCFR Algorithm

In this chapter we finally describe the proposed algorithm for solving extensive-form games. Our solution called **Deep-Hedge MCCFR** is a direct response to the shortcomings of the Deep CFR algorithm. It is fundamentally a synthesis of the Outcome-Sampling methods family with the approaches based on function approximation.

Our algorithm, in contrast to Deep CFR, uses neural networks to model the players' strategies explicitly rather than implicitly through the regrets approximation. Deep-Hedge MCCFR does not require maintaining large memory buffers to grasp the average regrets. The instantaneous regrets (see definition 3.8) sampled from an episode of play serve as a sufficient signal to direct the changes of the *strategy network's* parameters. This is possible thanks to the modification of the CFR algorithm, namely the replacement of the regret matching procedure with a multiplicative weights update method called *Hedge* [Freund and Schapire, 1997]. Furthermore, following the VR-MCCFR algorithm, the high variance of estimates is alleviated by utilizing another neural network in the role of a baseline function.

We start by giving an overview of the multiplicative weights methods for regret minimization with a particular focus on the Hedge algorithm. This method turned out to be amenable to function approximation techniques, which we cover in details. Then we follow by describing our baseline modelling scheme. Once we establish solid fundamentals, we will thoroughly recount the structure, the implementation and the evaluation results of the Deep-Hedge MCCFR algorithm.

4.1 Multiplicative Weights Update and Hedge Methods

The **multiplicative weights update method** is similarly to regret matching an iterative and adaptive technique for regret minimization. The decision-making agent assigns weights to each of the available actions. The probability of choosing a particular action at a given time is determined by its relative weight. The weights themselves are updated in a multiplicative way according to the reward the agent receives at the end of the round. In case the action performs well, its weight is increased relatively to other actions, otherwise it is decreased.

Let us express the above in more formal terms. By A we denote a finite set of actions and let $w^t(a)$ be the weight the algorithm allocates to action $a \in A$ at time t. Initially the weights of all actions are equal. The probability of choosing an action a is then given by:

$$p^t(a) = \frac{w^t(a)}{\sum_{a' \in A} w^t(a')}. \tag{4.1}$$

After announcing p^t a reward vector $u^t : A \to [-1, 1]$ is observed.

The multiplicative update rule of **Hedge** [Freund and Schapire, 1997] for action $a \in A$ is defined by the following formula:

$$w^{t+1}(a) = w^t(a) \cdot e^{\eta u^t(a)}, \tag{4.2}$$

where $\eta > 0$ is a fixed *learning rate* parameter.

Provided that η is chosen carefully, based on the number of trials T known ahead of time, the Hedge method guarantees the upper bound of $O\left(\sqrt{(\ln |A|)/T}\right)$ on the expected regret [Freund and Schapire, 1997].

We should note that the parameter η is the key factor which differentiates the Hedge algorithm from regret matching (see definition 3.4). The latter is a parameterless procedure, which is independent of the number of steps T.

4.2 Approximating Hedge with Neural Networks

The Deep-Hedge MCCFR algorithm approximates the behaviour of the Hedge method that is deployed individually for each information set of the game. It relies on deep neural networks to model the players' strategies. A network receives a representation of the information state as an input and outputs a vector of real numbers used directly to obtain the relevant mixed strategy. The core idea of our method is to alter the weights (or parameters) of the network at each round t in such a way that its outputs would be modified similarly to how Hedge alters the weights assigned to available actions.

Definition 4.1 (strategy network)**.** We define a **strategy network** of player $i \in \mathcal{N}$ as a function $W : \mathcal{I}_i \to \mathbb{R}^{\mathcal{A}(\mathcal{I}_i)}$ parameterized by a vector $\theta_i \in \mathbb{R}^d$, where d is the number of the weights (parameters) of the network. We assume that W is differentiable with respect to θ_i.

By $W(I, a; \theta_i)$ we denote the output value of the strategy network at an information state $I \in \mathcal{I}_i$ associated with an action $a \in \mathcal{A}(I)$.

Having defined a strategy network we can now provide the recipe for calculating the mixed strategy. For that purpose we use a *softmax* function [Goodfellow et al., 2016], which is frequently employed both in literature and in practice to model categorical distributions. The softmax function takes as input a vector of k arbitrary real numbers and normalizes it into a probability distribution consisting of k probabilities. The probabilities are proportional to the exponentials of the corresponding inputs.

Assume an information set $I \in \mathcal{I}_i$ and a time step t. The strategy of player i for action $a \in \mathcal{A}(I)$, is given by the application of the softmax function to the vector returned by the network W:

$$\sigma_i^t(I,a) = \frac{e^{W(I,a;\theta_i^t)}}{\sum_{a' \in \mathcal{A}(I)} e^{W(I,a';\theta_i^t)}}. \tag{4.3}$$

In this context the outputs $W(I;\theta_i^t)$ are called **logits**.

A close resemblance between the above and the formula (4.1) may be discerned, that is the value $e^{W(I,a;\theta^t)}$ can be seen as the weight $w^t(a)$ assigned to action a. Assuming the observed reward vector u^t, according to the Hedge update rule (4.2) the *target weights* vector $\hat{w}^{t+1}$ for the next round is:

$$\hat{w}^{t+1}(a) = e^{W(I,a;\theta_i^t)+\eta u^t(a)}. \tag{4.4}$$

We aim to modify the parameters θ_i^t so as to make $e^{W(I,a;\theta^{t+1})}$ as close to $\hat{w}^{t+1}(a)$ as possible for every action. Equivalently, we want to decrease some distance measure d between the vectors $W(I;\theta_i^t)$ and $\ln \hat{w}^{t+1}$, where the natural logarithm is applied piece-wise. We achieve this by using the gradient descent method:

$$\theta_i^{t+1} = \theta_i^t - \alpha \nabla_{\theta_i} d(W(I;\theta_i^t), \ln \hat{w}^{t+1}), \tag{4.5}$$

where $\alpha > 0$ is an adequately chosen **learning rate** parameter.

We use the squared l_2 norm as in the *least squares* [Theodoridis, 2015b] method for regression analysis. Thus, we can write:

$$\begin{aligned} d(W(I;\theta_i^t), \ln \hat{w}^{t+1}) &= ||W(I;\theta_i^t), \ln \hat{w}^{t+1}||_2^2 \\ &= \sum_{a \in \mathcal{A}(I)} \left(W(I,a;\theta^t) - \ln \hat{w}^{t+1}(a)\right)^2. \end{aligned}$$

Plugging the above into the equation (4.5) and substituting $\hat{w}^{t+1}(a)$ with (4.4) we get the update formula for the network parameters:

$$\begin{aligned} \theta_i^{t+1} &= \theta_i^t - \alpha \nabla_{\theta_i} \sum_{a \in \mathcal{A}(I)} \left(W(I,a;\theta_i^t) - \ln \hat{w}^{t+1}(a)\right)^2 \\ &= \theta_i^t - \alpha \sum_{a \in A(I)} 2\left(W(I,a;\theta_i^t) - \ln \hat{w}^{t+1}(a)\right) \nabla_{\theta_i} W(I,a;\theta_i^t) \\ &= \theta_i^t + 2\alpha\eta \sum_{a \in A(I)} u^t(a) \nabla_{\theta_i} W(I,a;\theta_i^t) \\ &= \theta_i^t + 2\alpha\eta u^t \cdot \nabla_{\theta_i} W(I;\theta_i^t), \end{aligned}$$

where $\cdot$ stands for the dot-product between the reward and the partial derivatives vectors. Since $2\alpha\eta$ is a constant coefficient we simplify the equation further, by using a single *learning rate* parameter $\alpha_w > 0$:

$$\theta_i^{t+1} = \theta_i^t + \alpha_w u^t \cdot \nabla_{\theta_i} W(I; \theta_i^t). \tag{4.6}$$

The formula we arrived at exhibits a very clear interpretation. It changes the logits the strategy network returns according to the direction of the corresponding rewards u^t. More concretely, the parameters θ_i^t are updated in such a way that the value the network W returns for an information set I and an action a is relatively increased (decreased) when the reward $u^t(a)$ is higher (lower).

4.3 Baseline Network

In order to mitigate the high variance of the utility estimates, the Deep-Hedge MCCFR algorithm harnesses the baseline functions (3.14) in a similar fashion to VR-MCCFR. We introduce another neural network playing the role of a baseline function. We define it analogously to the strategy network (4.1) except that it operates on augmented information sets.

Definition 4.2 (baseline network)**.** The **baseline network** of a player $i \in \mathcal{N}$ is a function $U : \hat{\mathcal{I}}_i \to \mathbb{R}^{\mathcal{A}(\hat{\mathcal{I}}_i)}$ parameterized by a vector $\phi_i \in \mathbb{R}^d$.

Now consider a player $i \in \mathcal{N}$, a terminal history $z \in \mathcal{Z}$ and its arbitrary prefix $h \sqsubset z$. When calculating the baseline-enhanced sampled values $\hat{u}_i^b(\sigma^t, h, a|z)$ (3.14) at time t, we will use $b_i(I_i(h), a) = U(I_i(h), a; \phi_i^t)$, for all $a \in \mathcal{A}(h)$.

Since the goal of the baseline is to approximate or at least to be sufficiently correlated with $u_i(\sigma^t, h \cdot a)$, we update the network's parameters after each round t making the output closer to the already calculated baseline-enhanced sampled values. However, this is done only for the chosen action a' where $h \cdot a' \sqsubseteq z$. Thus, similarly to the update of the strategy network we resort to the gradient descent method as a tool minimizing the distance between $U(I_i(h), a'; \theta_i^t)$ and $\hat{u}_i^b(\sigma^t, ha'|z)$:

$$\phi_i^{t+1} = \phi_i^t - \alpha_u \nabla_{\phi_i} \left(U(I_i(h), a'; \phi_i^t)) - \hat{u}_i^b(\sigma^t, ha'|z) \right)^2, \tag{4.7}$$

where $\alpha_u > 0$ is the learning rate parameter associated with the baseline.

We can draw a parallel between the baseline function proposed by us and the one used by the authors of VR-MCCFR [Schmid et al., 2018]. The update that we perform at each time step also "skews" the network more towards the recently estimated values rather than their historical averages.

4.4 Algorithm Description

We have now fully established the ground for the Deep-Hedge MCCFR algorithm. The procedure essentially follows an Outcome-Sampling scheme to minimize the sampled immediate counterfactual regrets (see definition 3.8) at each information state of the game.

Each player $i \in \mathcal{N}$ has two neural networks associated with them, namely the strategy network W (see definition 4.1) and the baseline network U (see definition 4.2), parameterized by θ_i and ϕ_i accordingly. We assume that the networks are exactly the same for each player in terms of their architecture, they only differ in the values of their weights. At the beginning, that is for $t = 1$, the parameters θ_i^1, ϕ_i^1 are initialized randomly.

Deep-Hedge MCCFR, following Deep CFR, is a self-play algorithm. During every round t the algorithm iterates through the players designating a *traversing* player i, for which the parameters θ_i^t, ϕ_i^t are to be adapted, while the networks of their opponents $-i$ are kept fixed. Then the algorithm performs a fixed number of K traversals using a sampling profile q. Each time it collects the baseline-enhanced sampled instantaneous regrets $\hat{r}_i^{b,t}(I(h), a|z)$ (see equation 3.10) along a single terminal history of the game $z \in \mathcal{Z}$ for all $a \in \mathcal{A}(h)$, where $h \sqsubset z$ and $\tau(h) = i$. The regrets vector $\hat{r}_i^{b,t}(I(h), \cdot|z)$, together with the corresponding information set $I(h)$ are memorized in the *strategy memory buffer* $\mathcal{M}_W$. The contents of $\mathcal{M}_W$ are used to update the strategy network according to the formula (4.6). By doing so we increase the output of the network at actions with positive regrets (better than average) and decrease at ones with negative regrets (worse than average). Thanks to using the regrets rather than the sampled values alone, the reward signal is always centered around 0, which guarantees significantly more stable training of the network.

Upon updating the strategy network, L series of H traversals are performed, each of which fills the *value memory buffer* $\mathcal{M}_U$ with the baseline-enhanced sampled values $\hat{u}_i^b(\sigma^t, ha)$. After each series the algorithm adapts the baseline network as prescribed by the formula (4.7). The reason for these repeated updates is that they improve the convergence and the quality of the baseline itself, which actually is trained partially on its own estimates.

We need to emphasize that the networks are not updated immediately when visiting a particular history h, but only after collecting more experience in the form of $\mathcal{M}_W$ or $\mathcal{M}_U$. Instead of minimizing the distance between a single target value and the relevant output of the network, we minimize the mean distance between numerous pairs of such target-output pairs. Not only this makes the algorithm faster in practice as it utilizes larger batches of values, but also this results in gradients showing less variance.

The sampling profile q is controlled by a parameter ϵ, which is the probability of choosing an action according to the uniform distribution instead of drawing it from the traversing player's mixed strategy. In other words, for a history $h \in \mathcal{H} \setminus Z$, such that $\tau(h) = i$ and a strategy profile σ, the probability of choosing an action $a \in \mathcal{A}(h)$ is $q(h, a) = \frac{\epsilon}{|\mathcal{A}(h)|} + (1 - \epsilon)\sigma_i(h, a)$. Whenever $\tau(h) \neq i$, q is identical to $\sigma_{\tau(h)}$.

Listing 1: Deep-Hedge MCCFR: Main training loop

```
1  function Deep_HedgeMCCFR(α_w, α_u, ϵ, T, K, L, H)
2     for player i ∈ 𝒩 do
3        Initialize the player's strategy network W with random weights θ_i
4        Initialize the player's baseline network U with random weights φ_i
5
6     for iteration step t = 1, ..., T do
7        for player i ∈ 𝒩 do
8           Initialize the strategy memory buffer
9           𝓜_W ← ∅
10          Initialize the value memory buffer
11          𝓜_U ← ∅
12
13       for traversal k = 1, ..., K do
14          traverse(i, ε, 1, 1, ϵ)
15
16       Update the strategy network's parameters
17       θ_i ← θ_i + (α_w / |𝓜_W|) Σ_{(I,r)∈𝓜_W} r · ∇_θ W(I; θ_i)
18
19       for series l = 1, ..., L do
20          for traversal h = 1, ..., H do
21             traverse(i, ε, 1, 1, ϵ)
22
23          Update the baseline network's parameters
24          φ_i ← φ_i − (α_u / |𝓜_U|) Σ_{(I_i,a,u)∈𝓜_U} ∇_φ (U(I_i, a; φ_i) − u)^2
25          Clear the value memory buffer
26          𝓜_U ← ∅
```

The main loop of the algorithm is enclosed in the `Deep_HedgeMCCFR` function presented on listing 1. We omit the superscripts denoting the time step t for clarity. The function takes two learning rate parameters, α_w for the strategy network and α_u for the baseline network. It also requires the parameter ϵ to drive the degree of sampling, the parameters T, K, L, H where

T controls the number of main training steps and K, L, H determine the amount of sampling done for the strategy and baseline networks.

Each iteration of the loop triggers execution of a recursive procedure `traverse` (see listing 2), which is responsible for sampling a single episode of play and filling the strategy and the value memory buffers $\mathcal{M}_W$ and $\mathcal{M}_U$. This function has to be provided with the traversing player i, the history h, the opponents' reach probability $\pi^{\sigma}_{-i}(h)$, the sampling reach probability $\pi^{q}(h)$ and the sampling degree ϵ. Both functions treat the network parameters θ, ϕ and the memory buffers as global and mutable variables.

Listing 2: Deep-Hedge MCCFR: Recursive traversal procedure

```
1  function traverse (i, h, π^σ_{-i}(h), π^q(h), ε)
2      if h ∈ Z then
3          return u_i(h)
4      else if τ(h) = c then
5          Sample the chance player's outcome
6          a ~ σ_c(h, ·)
7          return traverse(i, ha, σ_c(h,a)π^σ_{-i}(h), σ_c(h,a)π^q(h), ε)
8
9      Calculate the current player's strategy according to the equation (4.3)
10     σ_{τ(h)}(h, ·) ← softmax(W(I(h); θ_{τ(h)}))
11
12     Calculate the sampling strategy
13     if τ(h) = i then
14         q(h, ·) ← ε 1/|A(h)| + (1 − ε)σ_i(h, ·)
15     else
16         q(h, ·) ← σ_{τ(h)}(h, ·)
17
18     Sample the next action
19     a ~ q(h, ·)
20
21     Recursive call for the decision a
22     if τ(h) = i then
23         û^b_i(σ, ha|z) ← traverse(i, ha, π^σ_{-i}(h), q(h,a)π^q(h), ε)
24     else
25         û^b_i(σ, ha|z) ← traverse(i, ha, σ_{τ(h)}(h,a)π^σ_{-i}(h), q(h,a)π^q(h), ε)
26
27     Update the value memory buffer
28     M_U ← M_U ∪ {(I_i(h), a, û^b_i(σ, ha|z))}
29
```

30 *Calculate the baseline values*
31 $b_i(I_i(h), \cdot) \leftarrow U(I_i(h); \phi_i)$
32 *Calculate the baseline-enhanced sampled values*
33 $\hat{u}_i^b(\sigma, h, a|z) \leftarrow \frac{1}{q(h,a)} \left(\hat{u}_i^b(\sigma, ha|z) - b_i(I_i(h), a)\right) + b_i(I_i(h), a)$
34 $\hat{u}_i^b(\sigma, h, a'|z) \leftarrow b_i(I_i(h), a') \quad \forall_{a' \neq a}$
35
36 $\hat{u}_i^b(\sigma, h|z) \leftarrow \sum_{a \in \mathcal{A}(h)} \sigma_{\tau(h)}(h, a) \hat{u}_i^b(\sigma, h, a|z)$
37
38 **if** $\tau(h) = i$ **then**
39 *Calculate the baseline-enhanced sampled instantaneous regret*
40 $\hat{r}_i^b(I(h), \cdot|z) = \frac{\pi_{-i}^{\sigma}(h)}{\pi^q(h)} \left(\hat{u}_i^b(\sigma, h, \cdot|z) - \hat{u}_i^b(\sigma, h|z)\right)$
41 *Update the strategy memory buffer*
42 $\mathcal{M}_W \leftarrow \{(I(h), \hat{r}_i^b(I(h), \cdot|z))\}$
43
44 **return** $\hat{u}_i^b(\sigma, h|z)$

4.4.1 Notes on the Average Strategy

The Deep-Hedge MCCFR algorithm approximates the no-regret procedure scaled to extensive-form games and as such does not guarantee the convergence to Nash equilibria of the final strategies determined by the network parameters θ_i^T, for all $i \in \mathcal{N}$. We recall that this is also the case of the vanilla CFR algorithm. It is the time-averaged strategy (see equation (3.4)) that is likely to be the most resilient and desirable state. This work however, is focused solely on the no-regret procedure itself and does not tackle the problem of efficient and accurate estimation of the average strategy with neural networks. For the purpose of evaluation we calculate $\bar{\sigma}$ in a tabular way, exactly like the CFR algorithms family does. Nonetheless, we outline our ideas for addressing this problem, which could be explored in the future.

The average strategy could be obtained during the evaluation in a rather crude fashion by taking the average of the outputs of the networks $\theta_i^1, \ldots, \theta_i^T$. Unfortunately this requires storing the snapshots from all time steps and can be very costly in terms of the execution time. One could also imagine an additional, final step of the `Deep_HedgeMCCFR` procedure in which a new network with parameters $\bar{\theta}_i$ would be trained. The objective would be to minimize the expected difference in its outputs and the outputs of some randomly chosen network θ_i^t on a series of randomly sampled information states.

The idea described above could be further refined by introducing a constant-

size memory of k past networks from various points of time. This memory would be kept and updated throughout the entire life-cycle of the algorithm. At each time step t, when updating θ_i^t, an additional loss would be imposed on the network for diverging from the outputs of the networks in the memory. This approach, while appearing rather unsound, could potentially make the final network closer to the average.

4.5 Implementation

This section is dedicated to an overview of our implementation of the proposed algorithm. We omit fine-grained details to avoid obscuring the big picture of the idea itself. We briefly touch on the project structure and the tools we rely on. Then we direct our attention to various practical solutions that bring more context to the very abstract description of the algorithm from the previous section.

4.5.1 Source Code and Used Tools

We provide an implementation of the Deep-Hedge MCCFR algorithm and the script to reproduce our results. The code is written in the C++ language of the revision C++17. The project is set up using CMake (`https://cmake.org`), which manages the entire build process. The instructions for building and using the algorithm are written in the enclosed `README.md` file.

The project has several external dependencies, which are mostly included as `git` (`https://git-scm.com/`) submodules under the `extern` directory. Two key libraries that we use are `OpenSpiel` [Lanctot et al., 2019] and `LibTorch` [Paszke et al., 2019].

`OpenSpiel` is a general framework aiding the research on algorithms for playing numerous types of games. It provides various families of games, including competitive and cooperative multiplayer, perfect and imperfect-information, one-shot and sequential decision making games. The library exposes a convenient interface for viewing and manipulating the state of the game, which serves as an environment model during the algorithm execution. In addition to that, it provides multiple implementations of the most prominent algorithms from the literature such as CFR, Outcome-Sampling MCCFR, Deep CFR or different kinds of reinforcement learning [Sutton and Barto, 2018] based solutions. We use it as an interface to communicate with selected game environments.

`LibTorch` is a C++ front-end for the popular deep learning library `PyTorch`. It is built on an engine for transforming large multidimensional arrays of numbers to which we often refer as *tensors*. Essentially, it is aimed at building

complex neural networks and optimizing their parameters towards a given objective. This functionality is provided thanks to automatic and seamless differentiation. Sequences of tensor transformations are implicitly tracked in computational graphs to enable propagation of gradients during the optimization phase. The library embraces an imperative programming model and supports accelerated execution on GPUs.

The source files of the project can be split into several categories. The library code is located in the `include` and `src` directories. It provides:

- the `DeepHedgeMCCFRSolver` class implementing a single iteration of our algorithm,
- a core neural network architecture provided by the `StackedLinearNet` class,
- various utilities including the calculation of the tabular time-averaged strategy and loading the game environment from the configuration.

Then the `app` directory contains the following executables:

- `inspect_game`, which is a tool that prints a detailed information on a particular game and runs a full exemplary episode,
- `run_experiment` is the entry point to our algorithm, which when provided with a configuration file, runs a predefined number of training iterations and reports the results.

The configurations for all experiments are kept in the `configs` directory. Each experiment is fully defined by a `JSON` file, which describes the game, the settings for the strategy and baseline networks, as well as the algorithm hyperparameters.

The experiments themselves are recorded under the `experiments` directory. The directory keeps track of every execution by storing:

- the associated configuration,
- a file with the metrics of the algorithm reported every fixed number of steps (usually 250),
- checkpoints with a serialized algorithm state, which can be used to resume long-running experiments.

We also provide a Jupyter notebook (`https://jupyter.org/`) script located at `notebooks`. The notebook contains a brief Python code used to render the plots with execution results.

4.5.2 Architecture of Neural Networks

The core architecture of the neural networks [Theodoridis, 2015a], [Goodfellow et al., 2016] we use in our implementation is driven by the principle of simplicity and universality. The structure is common for both the strategy (4.1) and the baseline (4.2) networks. It is also shared by all the players of the given game. Moreover, the networks are game agnostic meaning that their architecture does not vary across the evaluated games, and more importantly, is not geared towards improving the performance at one of them at the expense of the others. The only differentiating factor between the games or between the strategy and baseline networks, may be the number of parameters or layers (the depth of the network).

At the basis of our implementation is a feed-forward neural network with a pure sequential arrangement. The network receives a vector x_I which is a real-valued encoding of an (augmented) information state $I \in \mathcal{I}_i$, for $i \in \mathcal{N}$. We make use of the encodings provided by the `OpenSpiel` framework and we do not subject them to any preliminary transformations. Every stage, except the last one, in the sequence is an application of a fully-connected layer followed by the ReLU activation (Rectified Linear Unit) [Goodfellow et al., 2016]. We also use skip-connections [He et al., 2015] in the hidden layers to improve the performance of learning and its stability.

If we assume a network consisting of K layers, then the kth stage, where $k < K$, can be defined as a function

$$l^k(x) = \max(H^k x,\, \mathbf{0}) + x',$$

where $x \in \mathbb{R}^{n\times 1}$ is an input to the layer, $H^k \in \mathbb{R}^{m\times n}$ is the matrix of parameters (weights) of the layer and $x' \in \mathbb{R}^{m\times 1}$ is the vector x adjusted to the dimension m either by padding it with $m - n$ zeros when $m > n$, or by removing the last $n - m$ values otherwise. The dimension m denotes the number of **units** of the layer.

The final layer of the network is a pure linear layer with no activation on top, that is $l^K(x) = H^K x$. The matrix H_i^K for player i has exactly $\max_{h\in\mathcal{H}:\tau(h)=i} \mathcal{A}(h)$ rows. Each action $a \in \mathcal{A}(h)$ is mapped to a dedicated index within the vector returned by the last stage. The indices with no actions assigned to them are ignored in further calculations.

To conclude the above with an example, let us assume a player $i \in \mathcal{N}$ and (augmented) information set $I \in \mathcal{I}_i$. The output of the K-layer strategy network W parameterized by $\theta_i = (H_i^1; \ldots; H_i^K)$ is given by the formula

$$W(I; \theta_i) = l_i^K \circ \cdots \circ l_i^1(x_I).$$

4.5.3 Algorithm Hyper-parameters and Implementation Tricks

The task of training a neural network, especially the one modelling the strategy of an agent, turns out to be difficult in practice. Gradient based optimization methods have a tendency to push the outputs of the network towards sharp distributions representing pure strategies. The trained weights take large values causing the network to get stuck in local optima of the training objective. Therefore, we draw the knowledge from the rich literature on deep learning and reinforcement learning and use several widely applied techniques.

In order to smooth the vectors returned from the final layer of the strategy network (4.1) to properly express mixed strategies, we introduce entropy to the optimization objective of the strategy network (see equation 4.6). This improvement is adopted from the asynchronous policy gradient methods [Mnih et al., 2016]. The updated formula for adapting the parameters is given as:

$$\theta_i^{t+1} = \theta_i^t + \alpha_w r_i^t \cdot \nabla_{\theta_i} W(I; \theta_i^t) + \beta \nabla_{\theta_i} H(\sigma_i^t(I)), \tag{4.8}$$

where $\sigma_i^t(I)$ is defined as in the equation (4.3), H is the entropy function, so that $H(\sigma_i^t(I)) = -\sum_{a \in \mathcal{A}(I)} \sigma_i^t(I,a) \log(\sigma_i^t(I,a))$, and $\beta > 0$ is a hyper-parameter controlling the relative influence of the entropy. We also set $r_i^t = u^t$ in the equation (4.6), where r_i^t stands for the vector of baseline-enhanced sampled instantaneous regrets.

In addition to the entropy regularization we also mitigate the problem described above by performing the updates selectively with respect to certain actions. That is, we assign zeros to actions in the r_i^t vector in case they would contribute to excessive polarization of the strategy. More precisely, we do not intend to increase (decrease) the probability of taking a particular action, for which the probability is already very high (low) relatively to other actions. Technically, we start by subtracting the mean from the logits vector element-wise

$$y^t = W(I, \theta_i^t) - \frac{\sum_{a \in \mathcal{A}(I)} W(I, a; \theta_i^t)}{|\mathcal{A}(I)|}.$$

Note that the resulting vector y^t induces exactly the same strategy $\sigma_i^t(I)$ as $W(I; \theta_i^t)$, since constant shifts do not affect the softmax function. We then mask the actions $a \in \mathcal{A}(I)$ in the regrets vector, for which $r_i^t(a) > 0$ and $y^t(a) > \gamma$ or $r_i^t(a) < 0$ and $y^t(a) < -\gamma$. In this context γ is a configurable parameter defining the acceptance threshold for the updates.

We further regularize the network parameters (both for baseline and strategy networks) by incurring an L2 cost on them [Theodoridis, 2015b], which

is also achieved by adding a term to the objective. This penalizes the network for having large weights and often helps avoid the detrimental effects of *over-fitting*. As a result another hyper-parameter κ is introduced to control the degree of this regularization.

Additionally we prevent the training from the problem of *exploding gradients* by using *gradient clipping* [Goodfellow et al., 2016]. Thus, whenever the norm of computed gradients exceeds a certain value it is scaled down to fit within the predefined range.

With time, as the network gets closer to desired strategies, there is a need to take more refined steps when performing the parameter updates. For this purpose we exponentially decrease the learning rate parameters α_w and α_u (see listing 1). That is, if we assume an initial learning rate α_0, then its value at time t is equal to

$$\alpha_t = \max(\alpha_0 d^{t/s}, \alpha_{\min}), \tag{4.9}$$

where $d < 1$ is the *decay rate*, s is the number of *decay steps* and $\alpha_{\min}$ is the minimum accepted learning rate.

We break down all the hyper-parameters of the implemented Deep-Hedge MCCFR algorithm in table 1.

4.6 Empirical Evaluation

In this section, which finalizes the chapter, we present the results of the implemented Deep-Hedge MCCFR algorithm. We describe our evaluation framework, namely the games and the metrics we use to analyze the performance. We also mention some research papers on related approaches and compare our metrics with the results reported therein.

4.6.1 Exploitability

We measure the performance of trained strategy profiles by quantifying the incentives that the players have to deviate from their strategies. This metric is called *exploitability* and is equal to the average difference in the payoffs with respect to the best response strategy (see definition 2.5).

Definition 4.3. Assume a set of $\mathcal{N}$ players and a strategy profile σ. Let σ_i^* denote the best response strategy for player i. The **exploitability** is given as:

$$\text{Exploitability}(\sigma) = \frac{1}{|\mathcal{N}|} \sum_{i \in \mathcal{N}} \left(u_i(\sigma_i^*, \sigma_{-i}) - u_i(\sigma_i, \sigma_{-i}) \right).$$

Table 1: List of the algorithm hyperparameters with names used by the implementation.

name	**meaning**
`player_units`	number of units of each hidden layer of strategy network
`baseline_units`	number of units of each hidden layer of baseline network
`max_steps`	argument T of `Deep_HedgeMCCFR` function
`player_traversals`	argument K of `Deep_HedgeMCCFR` function
`baseline_traversals`	argument H of `Deep_HedgeMCCFR` function
`player_update_freq`	argument L of `Deep_HedgeMCCFR` function
`player_lr_init`	initial learning rate α_w of strategy network
`baseline_lr_init`	initial learning rate α_u of baseline network
`decay_steps`	parameter s in equation (4.9)
`decay_rate`	parameter d in equation (4.9)
`player_lr_end`	parameter $\alpha_{w,\min}$ in equation (4.9)
`baseline_lr_end`	parameter $\alpha_{u,\min}$ in equation (4.9)
`gradient_clipping_value`	maximum gradient norm used by gradient clipping
`logits_threshold`	threshold γ for the strategy network updates
`weight_decay`	coefficient κ for L2 regularization cost
`entropy_cost`	coefficient β for entropy regularization cost
`epsilon`	exploration degree, the argument ϵ of `Deep_HedgeMCCFR` function

We can say that the exploitability metric determines the rate of convergence to equilibria of algorithms in practice.

4.6.2 Used Games

We evaluate the Deep-Hedge MCCFR algorithm on three zero-sum n-player games: Kuhn poker, Leduc poker and Goofspiel. The choice of the games is dictated by their relatively small sizes amenable to efficient analysis. What is more, these imperfect information games are commonly used as benchmarks across the literature.

Kuhn poker is a toy version of poker. Each player starts with 2 chips, antes 1 chip to play, and is dealt one card face down from a deck of size $n + 1$. The last card is put aside unseen. The players proceed by betting (raise/call) by adding their remaining chip to the pot, or passing (check/fold) until all players are either in (contributed as all other players to the pot) or out (folded, passed after a raise). The player with the highest-ranked card that has not folded wins the pot. We use the 2 and 3 player versions of the game.

Leduc poker [Southey et al., 2005] is another, less significant simplification of poker. The players have a limitless number of chips, and the deck has size $2(n + 1)$, divided into two suits of identically-ranked cards. There are two rounds of betting and after the first round a single private card is dealt to each player. Each player antes 1 chip to play, and the bets are limited to two per round. The raise amounts are limited to 2 and 4 and in the first and second round, respectively.

Goofspiel is an example of a multi-stage simultaneous-move game. In this game each player is dealt with the same suit of cards. One additional and shuffled suit is singled-out as prizes with the top card always revealed. The players proceed by making sealed bids for the top prize card. The selected cards are revealed simultaneously and the player with the highest bid wins the competition card. The cards used for bidding are discarded and the game continues until there are no remaining cards. In our evaluations we use the version of the game with suits consisting of 4 cards each.

The properties of the used games are presented in table 2. Each row shows the information on the numbers of players and information sets of the game, the maximum length of the episode over the players' decisions, the maximum number of distinct actions a player can take, and the range of utilities for each player.

Table 2: Properties of $n-$ player games used for evaluation.

name	**n**	**#infosets**	**length**	**#actions**	**utility range**
Kuhn	2	12	3	2	$[-2, 2]$
Kuhn	3	48	5	2	$[-2, 4]$
Leduc	2	936	8	3	$[-13, 13]$
Goofspiel	2	6056	8	4	$[-1, 1]$

4.6.3 Results

The Deep-Hedge MCCFR algorithm was evaluated against the four aforementioned game scenarios. We performed all experiments on a single machine with Intel Core i7-6700K processor, 32GB of random access memory, and Nvidia GeForce 1080 GTX GPU. In each case we ran the algorithm for 500000 iterations, which resulted in a wide range of execution times depending on the game and the algorithm settings. The evaluation of 2-player Kuhn poker took about 40 minutes to finish, while the 3-player version required one more hour. Then both Leduc and Goofspiel where significantly more expensive in terms of evaluation time, taking about 2 days each. This is not only caused by relatively large depth and width of the games, but also by the increased number of traversals per iteration. We played 8, 128 and 64 games per iteration and player for Kuhn, Leduc and Goofspiel, respectively. The sizes of the used neural networks were another key factor contributing to longer execution times.

When selecting the algorithm parameters we aimed at sharing most of them between the games, in order for the configuration to be as universal as possible. The settings differ mostly when it comes to the number of units of the networks and the number of traversals performed in each iteration. However, we also made an exception for Goofspiel, which consistently achieved higher performance with no entropy regularization applied at all. In other cases we used the factor of 0.1. Table 3 juxtaposes the algorithm parameters for every evaluation scenario.

The training dynamics for each game is presented in figures 2, 3, 4 and 5. We plot the exploitability metric of the current strategy σ^t and the average strategy $\bar{\sigma}^t$ at each iteration step. Note that the vertical axis has a logarithmic scale. All plots share the same pattern with the time-averaged strategy reaching values close to a Nash equilibrium, and the current strategy exhibiting great variability. This is anticipated by the theory covered in this

Table 3: Selected algorithm parameters.

	2,3p Kuhn	**Leduc**	**Goofspiel**
`max_steps`	500000	500000	500000
`player_units`	[48, 48, 32]	[196, 196, 48]	[196, 196, 48]
`baseline_units`	[48, 48, 32]	[128, 128, 32]	[128, 128, 32]
`player_traversals`	4	64	32
`baseline_traversals`	4	16	16
`player_update_freq`	1	4	2
`player_lr_init`	0.01	0.1	0.01
`baseline_lr_init`	0.01	0.01	0.01
`decay_steps`	1×10^5	1×10^5	1×10^5
`decay_rate`	0.95	0.95	1×10^5
`player_lr_end`	1×10^{-6}	1×10^{-6}	1×10^{-6}
`baseline_lr_end`	1×10^{-6}	1×10^{-6}	1×10^{-6}
`gradient_clipping_value`	1×10^4	1×10^4	1×10^4
`logits_threshold`	2.0	2.0	2.0
`weight_decay`	1×10^{-4}	1×10^{-4}	1×10^{-4}
`entropy_cost`	0.1	0.1	0.0
`epsilon`	0.1	0.1	0.1

work, but still emphasizes the need for methods avoiding the calculation of tabular average strategies.

Consider the case of 2-player Kuhn poker depicted in figure 2. The average strategy profile computed by the algorithm reached the regions empirically close to a Nash equilibrium, as the exploitability stabilized oscillating around 0.009. Further improvements might be difficult to achieve due to the lack of numerical accuracy or still too large learning rates at this stage. The latter could be addressed by more careful selection of the learning rate adjustment schedule.

Let us now move to the 3-player version of Kuhn poker (see figure 3). Even though the setting is more complex with four times the number of information states, the algorithm quickly reaches the exploitability of 0.02 in around 100000 iterations. After that the progress appears to slow down not going beyond 0.01 in the predefined number of steps.

Goofspiel and Leduc presented in figures 4 and 5, despite being disparate games, appear to be of similar difficulty to the Deep-Hedge MCCFR algorithm. The point of exploitability equal to 0.2 is crossed after about 200000 iterations. The subsequent 300000 steps of training bring down the metrics to around 0.17 for both games. The plots suggest that the metrics would

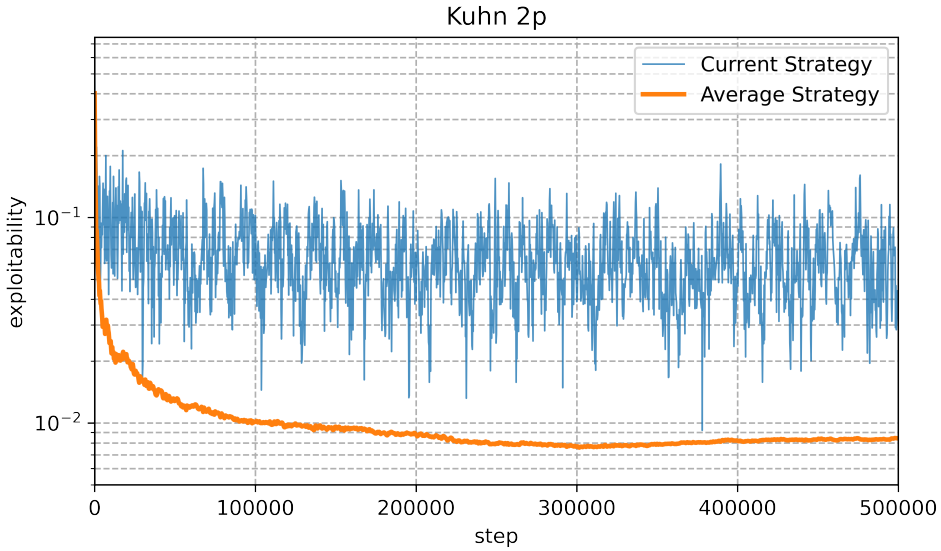


Figure 2: Exploitability of 2-player Kuhn poker.

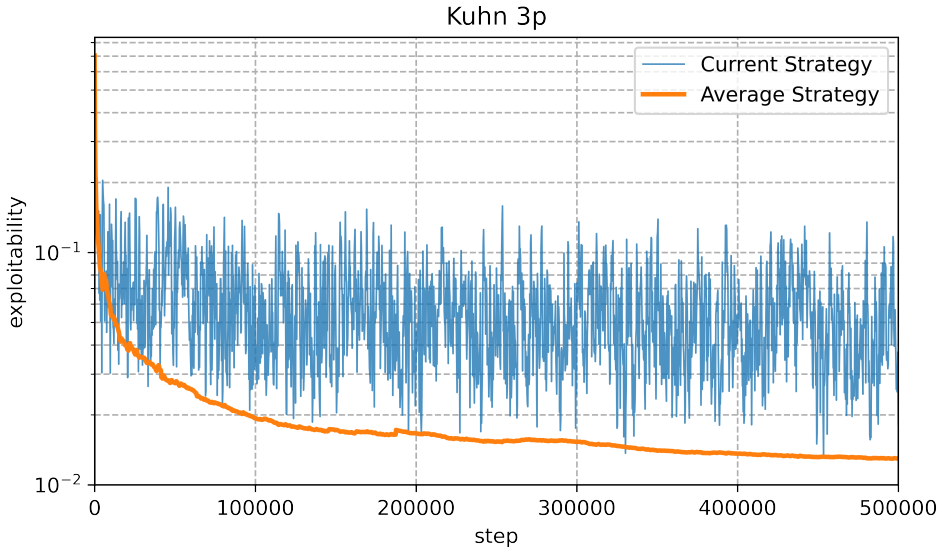


Figure 3: Exploitability of 3-player Kuhn poker.

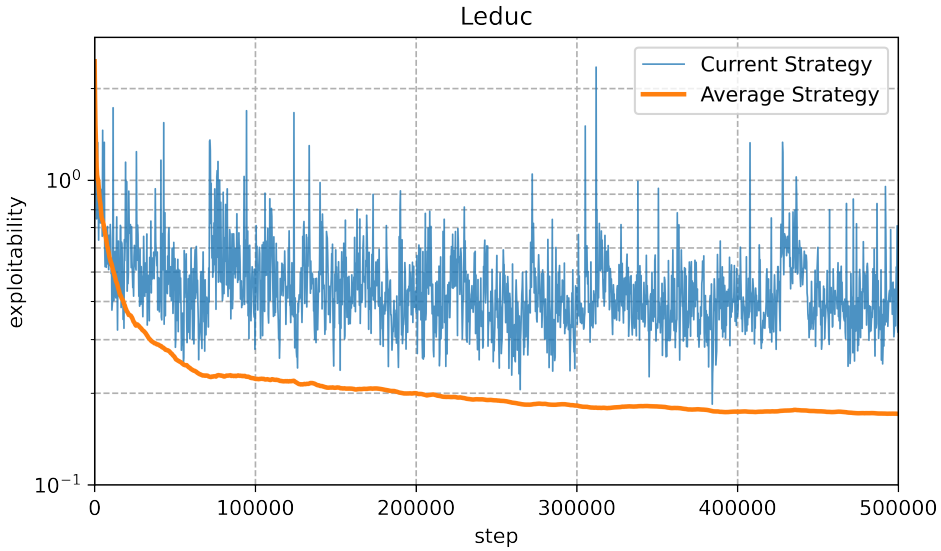


Figure 4: Exploitability of Leduc poker.

continue to improve. Unfortunately because of the observed exponential slowdown, reaching values below 0.1 would potentially require a few millions of additional steps.

We proved empirically that the proposed solution converges towards Nash equilibria in the benchmark games. Larger imperfect information games like Texas Holdem, as less amenable to exploitability analysis, would involve a direct comparison of the agents trained with various algorithms. We leave this for future explorations, especially that the training process being so expensive poses an independent problem to solve.

We acknowledge that better results could be achieved by a thorough hyper-parameter search performed independently for each game. Given the number of algorithm parameters and the time requirements of evaluation itself, to make such an effort feasible a costly distributed optimization algorithm would have to be devised and executed. This, however, we treat as a topic for future research.

4.6.4 Comparison with Related Approaches

There are numerous algorithms for solving imperfect information games. However, for the comparison to be sound, we chose algorithms based on function approximation, which similarly to Deep-Hedge MCCFR sample single episodes during every traversal. We briefly describe the solutions and the

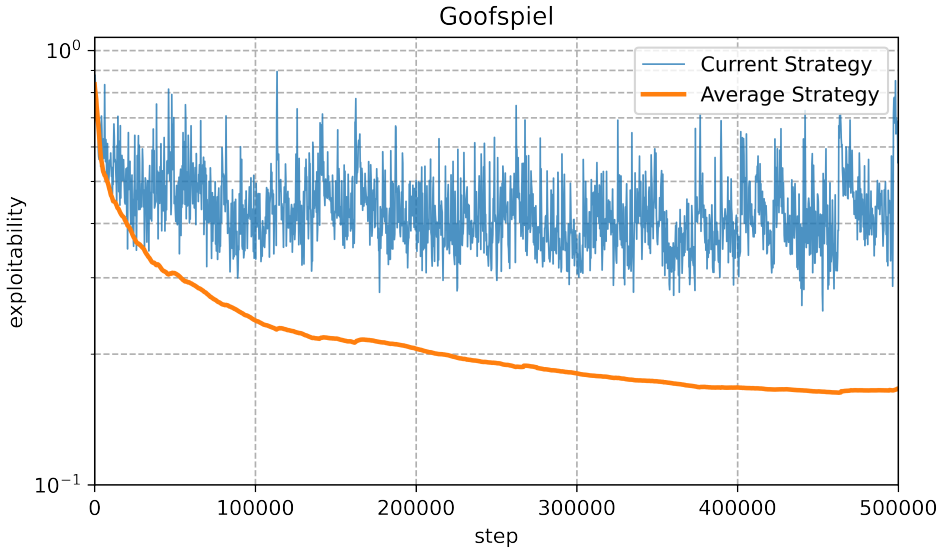


Figure 5: Exploitability of Goofspiel.

results reported in the literature.

State-of-the-art policy gradient based algorithms [Sutton and Barto, 2018] were adapted to domain of game theory in the paper [Srinivasan et al., 2018]. The authors propose several modifications to the update formula for the policy network, to which they give the names of q-based Policy Gradient (QPG), Regret Policy Gradient (RPG) and Regret Matching Policy Gradient (RPMG). The paper contains a comparison of these algorithms with the baseline solution, namely a Neural Fictitious Self-Play (NSFP) algorithm [Heinrich and Silver, 2016], which is a combination of deep Q-learning [Sutton and Barto, 2018] with fictitious self-play. The evaluation results are presented for 2-player and 3-player Kuhn poker as well as 2-player Leduc poker. The evaluation metric is plotted with respect to the number of episodes rather than iterations steps. After accounting for this fact, the Deep-Hedge MCCFR algorithm seems to be converging faster in all scenarios. Unfortunately we cannot confirm this with exact numbers due to the poor readability of the reported plots.

An alternative technique is the Neural Replicator Dynamics (NeuRD) algorithm [Hennes et al., 2019]. The solution, which is inspired by methods of evolutionary game theory, uses neural networks to model the agents' strategies. The authors propose an update rule which resembles the formula used by Deep-Hedge MCCFR, except that it does not utilize baseline-enhanced values (see definition 3.15). What is more, the formula used for computing

the values at certain states is not provided, which makes it more difficult to assess the differences. Each iteration of the NeuRD algorithm samples a batch of 256 episodes independent of the game, which is significantly more than in our case. The evaluation plots generated for Kuhn poker, Leduc poker and Goofspiel are of insufficiently high resolution for concrete values to be inferred. Nonetheless, the metrics seem to follow similar paths to the dynamics reported for Deep-Hedge MCCFR.

References

[Brown et al., 2018] Brown, N., Lerer, A., Gross, S., and Sandholm, T. (2018). Deep counterfactual regret minimization. *CoRR*.

[Brown and Sandholm, 2019] Brown, N. and Sandholm, T. (2019). Superhuman ai for multiplayer poker. *Science*, 365(6456):885–890.

[Davis et al., 2019] Davis, T., Schmid, M., and Bowling, M. (2019). Low-variance and zero-variance baselines for extensive-form games. *CoRR*.

[Freund and Schapire, 1997] Freund, Y. and Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[Hart and Mas-Colell, 2016] Hart, S. and Mas-Colell, A. (2016). 9. a simple adaptive procedure leading to correlated equilibrium. *General Equilibrium and Game Theory*.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*.

[Heinrich and Silver, 2016] Heinrich, J. and Silver, D. (2016). Deep reinforcement learning from self-play in imperfect-information games. *CoRR*.

[Hennes et al., 2019] Hennes, D., Morrill, D., Omidshafiei, S., Munos, R., Perolat, J., Lanctot, M., Gruslys, A., Lespiau, J.-B., Parmas, P., Duenez-Guzman, E., and Tuyls, K. (2019). Neural replicator dynamics. *CoRR*.

[Kuhn, 1953] Kuhn, H. W. (1953). Extensive games and the problem of information. In Kuhn, H. W. and Tucker, A. W., editors, *Contributions to the Theory of Games (AM-28), Volume II*, pages 193–216. Princeton University Press.

[Lanctot et al., 2019] Lanctot, M., Lockhart, E., Lespiau, J.-B., Zambaldi, V., Upadhyay, S., Pérolat, J., Srinivasan, S., Timbers, F., Tuyls, K., Omidshafiei, S., Hennes, D., Morrill, D., Muller, P., Ewalds, T., Faulkner, R., Kramár, J., Vylder, B. D., Saeta, B., Bradbury, J., Ding, D., Borgeaud, S., Lai, M., Schrittwieser, J., Anthony, T., Hughes, E., Danihelka, I., and Ryan-Davis, J. (2019). Openspiel: a framework for reinforcement learning in games. *CoRR*.

[Lanctot et al., 2009] Lanctot, M., Waugh, K., Zinkevich, M., and Bowling, M. (2009). Monte carlo sampling for regret minimization in extensive games. In Bengio, Y., Schuurmans, D., Lafferty, J. D., Williams, C. K. I., and Culotta, A., editors, *Advances in Neural Information Processing Systems 22*, pages 1078–1086. Curran Associates, Inc.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *CoRR*.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518:529–33.

[Nash, 1950] Nash, J. F. (1950). Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, 36(1):48–49.

[Neller and Lanctot, 2013] Neller, T. W. and Lanctot, M. (2013). An introduction to counterfactual regret minimization. `http://modelai.gettysburg.edu/2013/cfr/cfr.pdf`.

[Nisan et al., 2007] Nisan, N., Roughgarden, T., Tardos, E., and Vazirani, V. V. (2007). *Algorithmic Game Theory*. Cambridge University Press, New York, NY, USA.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). PyTorch: an imperative style, high-performance deep learning library. *CoRR*.

[Schmid et al., 2018] Schmid, M., Burch, N., Lanctot, M., Moravcik, M., Kadlec, R., and Bowling, M. (2018). Variance reduction in monte carlo counterfactual regret minimization (VR-MCCFR) for extensive form games using baselines. *CoRR*.

[Shoham and Leyton-Brown, 2008] Shoham, Y. and Leyton-Brown, K. (2008). *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, USA.

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503.

[Southey et al., 2005] Southey, F., Bowling, M., Larson, B., Piccione, C., Burch, N., Billings, D., and Rayner, C. (2005). Bayes' bluff: Opponent modelling in poker. In *In Proceedings of the 21st Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 550–558.

[Srinivasan et al., 2018] Srinivasan, S., Lanctot, M., Zambaldi, V., Perolat, J., Tuyls, K., Munos, R., and Bowling, M. (2018). Actor-critic policy optimization in partially observable multiagent environments. *CoRR*.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.

[Theodoridis, 2015a] Theodoridis, S. (2015a). Chapter 18: Neural networks and deep learning. In Theodoridis, S., editor, *Machine Learning*, pages 875–936. Academic Press, Oxford.

[Theodoridis, 2015b] Theodoridis, S. (2015b). Chapter 6: The least-squares family. In Theodoridis, S., editor, *Machine Learning*, pages 233–274. Academic Press, Oxford.

[Vinyals et al., 2019] Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., and Silver, D. (2019). Grandmaster level in Starcraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354.

[Zinkevich et al., 2008] Zinkevich, M., Johanson, M., Bowling, M., and Piccione, C. (2008). Regret minimization in games with incomplete information. In Platt, J. C., Koller, D., Singer, Y., and Roweis, S. T., editors, *Advances in Neural Information Processing Systems 20*, pages 1729–1736. Curran Associates, Inc.