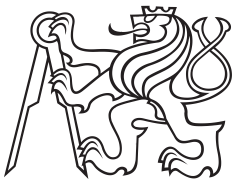
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Combining verification methods and adversarial sample generation with game-theoretic frameworks

Ondřej Skoumal

Supervisor: Mgr. Branislav Bošanský, Ph.D.
Field of study: Open Informatics
Subfield: Artificial Intelligence
May 2021

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Skoumal** Jméno: **Ondřej** Osobní číslo: **368933**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Umělá inteligence**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Využití algoritmů verifikace a generování adversariálních vstupů v herně-teoretických algoritmech

Název diplomové práce anglicky:

Combining verification methods and adversarial sample generation with game-theoretic frameworks

Pokyny pro vypracování:

Seznam doporučené literatury:

[1] Dvijotham K, Stanforth R, Gowal S, Mann TA, Kohli P. A Dual Approach to Scalable Verification of Deep Networks. InUAI 2018 Mar (pp. 550-559).
[2] Ruan W, Huang X, Kwiatkowska M. Reachability analysis of deep neural networks with provable guarantees. InProceedings of the 27th International Joint Conference on Artificial Intelligence 2018 Jul 13 (pp. 2651-2659). AAAI Press.
[3] Šilhavý P. "Využití algoritmu inkrementálního generování strategií pro klasifikaci akcí útočníka." (2019). Diplomová práce. České vysoké učení technické v Praze

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Mgr. Branislav Bošanský, Ph.D., centrum umělé inteligence FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **19.02.2020** Termín odevzdání diplomové práce: **21.05.2021**

Platnost zadání diplomové práce: **19.02.2022**

doc. Mgr. Branislav Bošanský, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. | Datum převzetí zadání

Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

I would like to thank my supervisor, Doc. Mgr. Branislav Bošanský, Ph.D., for his very patient guidance, insightful helpful advises, and constructive criticism.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university thesis.

Prague, May 21, 2021

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze 21. května 2021

Abstract

This thesis examines the possible use of the domain-specific knowledge of NNs to speed up the double-oracle framework, which solves the Nash equlibria of the adversarial classification game. We examine various adversarial attack methods and methods for verification of NNs and analyze their possible compatibility with the DO framework. We create a compatibility framework for entire families of methods and examine three of them in the experiments. It shows that PGD attack with compatibility framework can rapidly accelerate the DO algorithm, although new bottleneck shows up.

Keywords: Double Oracle, adversarial machine learning, infinite games

Supervisor: Mgr. Branislav Bošanský, Ph.D.

Abstract

Diplomová práce se zabývá možným použitím doménově-specifických znalostí z oblasti neuronových sítí v double-oracle frameworku, který se zabývá najitím Nashova equilibria adversariálního klasifikačního problému. Procházíme různé druhy metod adversariálních útoků a metod na verifikaci neuronových sítí, a analyzujeme jejich možnou kompatibilitu s frameworkem double-oracle. vytvořili jsme framework zajišťující kompatibilitu s celými "rodinami" metod a tři z nich experimentálně analyzujeme. Ukazalo se že PGD útok společně s frameworkem zajišťujícím kompatibilitu dokáže rapidně zrychlit DO algoritmus, i přesto že se našel další zdroj zpomalení algoritmu.

Keywords: algoritmus inkrementálního generování strategií, adversariální klasifikace, nekonečné hry

Title translation: Využití algoritmů verifikace a generování adversariálních vstupů v herně-teoretických algoritmech

Contents

Figures

Tables

Chapter 1

Introduction

In today world, artificial intelligence (AI) becomes less and less the enigmatic field inducing the ideas of huge computers that one day mysteriously become self-aware and tries to do some malicious intention against humans, as is common for example in science-fiction movies.

Instead, subfield of AI, the machine learning, has incredibly advanced over the last 10 years, and its many state of the art successes such as optical character recognition [9], face recognition [10], speech recognition [11], to playing various games, e.g. Starcraft 2 [12] or Atari games [13] and huge practical application potential exposes it's true paradigms, algorithms and possibilities to a wider public. Yet another example of possible machine learning deployment can be seen in recent trend in automotive and other industries called Industry 4.0 [14].

In the machine learning, the deep neural networks surpasses other machine learning technologies and are behind successes mentioned earlier. [15]. Yet recently, it was shown that exactly these neural networks are highly susceptible to small perturbations performed to the input (these minimally perturbed inputs are called *adversarial examples*, because they are generated by the adversary), which cause NNs to misclassify with high probability [16, 17, 5].

To demonstrate simplicity and effectiveness of adversarial attacks, we will create an adversarial example for the winner of ILSVRC 2015 classification task, the 50-layer deep convolutional neural network ResNet50 [18]. We take an image of a pig from ImageNet images [1] (if we would classify it by our (pre-trained) model we would get approximately 0.996 probability of a pig)

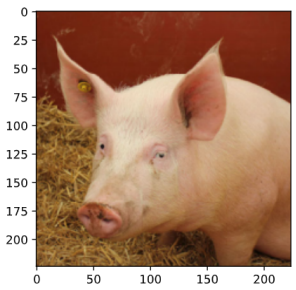


Figure 1.1: An image of "pig" from ImageNet database [1].

We create adversarial example from it (we will describe process of creation of adversarial examples in much greater detail later in chapter) by using e.g. PyTorch framework [19]. It can be done by computing gradient with respect to this original image, so it tells us how small change to image will affect the loss function. So we will maximize the loss with respect to our new adversarial image and original image will be changed to other image with highly decreased logit value of the original class. If we in the same time minimize loss of a target class, we will get targeted adversarial example. With PyToch, in a few lines of code we change the visuals of a pig (unnoticeably to human viewers) and trick model into deciding that this new image is an airliner, with a convicting approximately 0.968 probability:

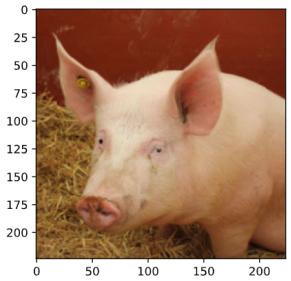


Figure 1.2: An image of "pig airliner" adversarial example.

And all it took was adding this "random" noise to the original image:

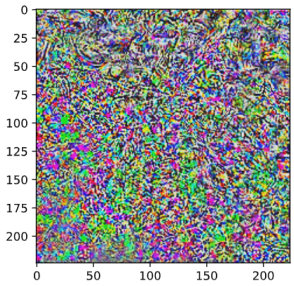


Figure 1.3: A random noise computed by gradient descent to create "pig airliner" adversarial example.

As this example can be seen rather funny than dangerous, it is not hard to conceive much more dangerous situations, such as autonomous car not stopping on the Stop sign because of a little mud on this sign, or much more sensitive areas where machine learning can be used, more exposed to adversarial behavior such as security - e.g. malware detection.

This lead us to the adversarial settings - the game theory studies various adversarial games [20] and one such game can be detecting an input to be malign or benign, in dependency on some negative data. We can use game-theory to model such situations, and even use its tools to find the optimal or near optimal solutions [21].

So why we could not use the game-theory to solve the problem of finding the optimal or nearby optimal strategy how to resist the adversarial attacks -

game theoretic approaches are employed by many works that aims to make adversarially robust classifiers [22].

In this thesis, we examine these approaches and methods for generating adversarial examples and verification of neural networks to analyse their possible use and compatibility in game-theoretical model. We use an existing Double Oracle framework [4], which models an adversarial classification problem as an almost zero-sum game, in which the strategy of the defender is to setting up hyper-parameters of a classifier and the strategy of an attacker is to choose such an input that causes misclassification. The framework is capable to incrementally build a discrete version of this infinite game and converge to it's Nash equilibrium on various predefined domains, thus find a robust classifier for the defender [4]. The framework experiments with three classifier types - decision trees, support vector machines and neural networks, and although it converges with all these classifiers in it's experiments, the computation time grows very quickly with number of dimensions.

1.1 Related work

As mentioned earlier, there is now a very large volume of work on the verification of neural networks and generating adversarial examples. Many of these works proposes various forms of *adversarial training*, a method of training in which the defender augments each minibatch of training data with adversarial examples to improve robustness of a model [23]. We discuss these works in detail in chapter ID, but to date, it remains unknown *what exactly* makes neural network models so vulnerable to adversarial attacks or if exists any universal defense to protect from them [5].

An intriguing existence of the adversarial examples was first discovered by Szegedy et al. [16]. In [17], authors suggests that adversarial examples can be explained as a property of high-dimensional dot products, and that they are caused by the fact that a models are being too linear.

Therefore, in this work we study the various adversarial attacks and verification methods for the deep neural network domain and we hope to find a methods compatible with the reward functions and strategic constraints of the adversarial game setting, which we would leverage to improve the scalability of the DO framework.

1.2 Outline

We first define necessary game theoretic concepts, on which is Double Oracle framework build on, in Chapter 2.

Later in Chapter 3 we use these concepts and define adversarial classification problem as a game as is implemented in Double Oracle framework we use, and introduce this framework in detail.

Next in Chapter 4 we study the adversarial attacks and verification methods to possibly use the to accelerate the DO framework.

In Chapter 5, we propose a compatibility framework that is possible to provide compatible ground for many various methods.

We study and experiment with three of them and provide the experimental analysis in 6.

Finally, we conclude the efforts we made and discuss the results, as well as possible future work.

Chapter 2

Game Theory Concepts

In this chapter, we provide basic overview of necessary game theory definitions and concepts which are in the next chapter used to define adversarial classification problem as a *game*, and to find it's solution.

We define basic concepts on the simplest form of games first, which we later generalize to the game-theoretical models which are able to capture critical features of our adversarial classification problem, and which we are able to solve.

2.1 Introduction

Game Theory is a computational framework used in various fields such as economics, political science, psychology, biology and computer science. It is using mathematical formalization of a strategic interaction between self-interested agents - and works with mathematical models called *games*. The agents interacting in the games are called players, and in game theory they are expected to behave rationally [21, 24].

Example 2.1. Popular Rock, Paper, Scissors is an example of a two-player game.

The objective of the game theory is to completely analyze games, which means to predict the result or at least find conditions which the result must uphold [24].

Games can be divided into *non-cooperative games* and *cooperative games*. Non-cooperative games models interactions between individual players with their interests, and in cooperative games there are modelled interactions between coalition of players [21].

The individual player's interests are modelled using *utility theory*. An *utility function* maps states of the game to real numbers and thus shows player's *preferences*. The interests of individual players doesn't need to be necessarily in conflict with other players, but often is [21].

We focus only on the two-player games in this thesis, as our adversarial classification problem setting has only two players - the attacker and the defender.

2.2 Normal-Form Games

2.2.1 Definition

The most fundamental formalization of strategic interaction between players is the normal form game. In this form, every player's utility is stated for all players' combined actions. The game is single-step, which means that after all players performs simultaneously single action, the game ends.

Definition 2.2 (Normal-form game [21]). A (finite, n-person) *normal-form game* (NFG) is a tuple (N, A, u), where:

- N is a finite set of n players, indexed by i;
- $A = A_1 \times \cdots \times A_n$, where A_i action is a finite set of actions available to player i. Each vector $a = (a_1, \ldots, a_n) \in A$ is called an action profile;
- $u = (u_1, \ldots, u_n)$, where $u_i : A \to \mathbb{R}$ is a real-valued utility (or payoff) function for player i

Small normal-form games can be conveniently represented by matrices, called the payoff matrices:

Example 2.3. We show the payoff matrix of an previous example 2.1 of hand game Rock, Paper, Scissors.

		Player 2		
		Rock	Paper	Scissors
Player 1	Rock	0, 0	-1, 1	1, -1
	Paper	1, -1	0, 0	-1, 1
	Scissors	-1, 1	1, -1	0, 0

Figure 2.1: The payoff matrix of Rock, Paper, Scissors game.

Each of the two players can choose either rock, paper, or scissors [1]. Each player action wins against one, loses against other and draws against itself. E.g. rock wins over scissors, loses against paper and draws against another rock.

2.2.2 Strategies in Normal-Form Games

If the player choose single action and play it, we term it as a *pure strategy*. So the rows of the matrix in example 2.3 represent pure strategies for player 1, columns represent pure strategies of player 2.

Concept of pure strategies is not enough for solution of the normal-form games, as there exists games in which the solution must contain randomization over available actions. Such a game is indeed the Rock, Paper and Scissors game, because the pick of an arbitrary pure strategy leads to negative utility,

[1] there exists many variants, such as a full-body variation in lieu of the hand signs called *Bear, Hunter, Ninja*

e.g. if player 1 strategy would be Rock, then player 2 will always choose Paper, so player 1 would always lose.

Definition 2.4 (Mixed strategy [21]). Let (N, A, u) be a normal-form game, and for any set X let $\Pi(X)$ be the set of all probability distributions over X. Then the set of *mixed strategies* for player i is $\Sigma_i = \Pi(A_i)$.

So the mixed strategy can be seen as generalization of the pure strategy concept. The pure strategies which are given non zero probabilities by mixed strategy s are called *support* of s. If the size of support of s equals number of actions of s, we will call it the *full support* [21].

We can extend the utility function to calculate the *expected utility* value as:

Definition 2.5 (Expected utility [21]). Given a normal-form game (N, A, u), the *expected utility* u_i for player i of the mixed-strategy profile $\sigma = (\sigma_1, \ldots, \sigma_n)$ is defined as

$$u_i(\sigma) = \sum_{a \in A} u_i(a) \prod_{j=1}^{n} \sigma_j(a_j) \tag{2.1}$$

Now that we can compute expected utility of individual players according to their respective strategies, we can ask how to maximize their individual utility, because as utility represents preferences of players, each player's goal is naturally to maximize it [3].

Example 2.6. We stay with our example of Rock, Paper and Scissors game, for which we have already defined the payoff matrix (example 2.3), and we compute expected utility for player 1 with the strategy profile $\sigma = ((\frac{1}{4}, \frac{1}{2}, \frac{1}{4}), (1, 0, 0))$, i.e. player 1 plays strategy with full support, Rock and Scissors have probabilities of $\frac{1}{4}$, Paper have probability of $\frac{1}{2}$. Player 2 plays strategy with only one single action in support - Rock. The expected utility can be computed as (we leave actions for which player 2 has zero probability of playing out of computation):

$$u_1(\sigma) = 0 \cdot \frac{1}{4} \cdot 1 + 1 \cdot \frac{1}{2} \cdot 1 - 1 \cdot \frac{1}{4} \cdot 1 = \frac{1}{4} \tag{2.2}$$

Because player 1 plays with highest probability Paper, which "counters" pure Rock strategy of player 2, he gets positive expected utility of $\frac{1}{4}$. From this observation we can notice that if player 1 knew pure strategy of the opponent, he can simply choose the *best response* and have maximum possible utility - his best response to pure Rock strategy of the opponent would be to play pure strategy Paper with expected utility 1. This observation is important and will take us further to the most influential concept in game theory, the *Nash equilibrium* [21].

2.2.3 Definition of Nash Equilibrium

Definition 2.7 (Best response [25]). Let $\sigma = (\sigma_1, \ldots, \sigma_i, \ldots, \sigma_n)$ be a strategy profile. Player i's *best response* to the strategy profile $\sigma_{-i} = (\sigma_1, \ldots, \sigma_{i-1},$

$\sigma_{i+1}, \ldots, \sigma_n)$ is a mixed strategy $\sigma_i^* \in \Sigma_i$ such that $u_i(\sigma_i^*, \sigma_{-i}) \geq u_i(\sigma_i, \sigma_{-i})$ for all strategies $\sigma_i \in \Sigma_i$.

So the best response is such a strategy of the player that there doesn't exists any other strategy that would give better utility for this player against some given strategy of the opponent.

Definition 2.8 (Nash equilibrium [25]). A mixed strategy profile σ is a *Nash equilibrium* if $u_i(\sigma_i', \sigma_{-i}) \leq u_i(\sigma)$ for all i and all $\sigma_i' \in \Sigma_i$.

If the players are playing Nash equilibrium (NE), neither of them has anything to gain by changing only their own strategy, so Nash equilibrium is a *stable* strategy profile. It is important to state that Nash equilibrium don't anticipate any form of communication between players - they cannot arrange ahead.

Example 2.9. We show Nash equilibrium in interesting example that is less classical (but very similar to the Prisoner's dilemma) - the Stag Hunt [2]. We have two hunters going to the hunt, each one from a distant village and they cannot coordinate ahead which game to hunt. They can go hunting for a stag which has the most meat but risks that the other hunter will go hunting for a hare, and if they both go for a hare they share two small hares living in the range. If one goes for a hare and other for a stag, the one going for a hare finds two of them and other cannot alone catch a stag, so he returns with nothing. The payoff matrix is:

		Player 2	
		Stag	Hare
Player 1	Stag	3, 3	0, 2
	Hare	2, 0	1, 1

Figure 2.2: A Stag Hunt game [2].

We can simply find the pure-strategy Nash equilibria in this game by iterating through all possible states and analyse if any of players has a incentive to change his or her strategy.

The one Nash equilibrium is the $\{Stag, Stag\}$ strategy profile, but there exists another one, the $\{Hare, Hare\}$.

The example 2.2 demonstrates important aspect - in general a game can have multiple NE that have different expected utilities [3].

2.2.4 Computing Nash Equilibria in Zero-Sum NFG

If we want to compute NE effectively in NFG, we must define additional types of games - *general-sum games* with arbitrary player's utility values, and *zero-sum games*, where for each strategy profile holds that it's utilities sum to zero (they are also called strictly competitive, because one player's gain is to the expense of the other). Additionally, for the zero-sum games holds

$u_i = -u_{-i}$ [21].

So even in normal-form games, we cannot compute NE effectively for any game variations. The great advantage of the normal-form zero-sum games is that their NE are much easier to find than in the case of general-sum games, and they all have the same value. This equilibrium strategies expected utility value is called *value of the game* [3].

As our adversarial classification problem has only two players, the *linear programming* (LP) can be conveniently adopted for the computation of a solution - it is because the problem of solving the two player, zero-sum games can be expressed as a *linear program*, which allows us to avoid the other more complicated game solutions [3, 21]. The linear program used is from [21, 4]:

$$\begin{aligned}
&\text{minimize} && U_1^* && && (2.3)\\
&\text{subject to} && \sum_{k \in A_2} u_1(a_1^j, a_2^k) \cdot s_2^k \leq U_1^* && \forall j \in A_1 && (2.4)\\
& && \sum_{k \in A_2} s_2^k = 1 && && (2.5)\\
& && s_2^k \geq 0 && \forall k \in A_2 && (2.6)
\end{aligned}$$

The linear program is formed using the minmax theorem [26]. It minimizes utility of player 1, and it's Nash equilibrium gives us mixed strategy of player 2. By solving the dual:

$$\begin{aligned}
&\text{maximize} && U_2^* && && (2.7)\\
&\text{subject to} && \sum_{j \in A_1} u_2(a_1^j, a_2^k) \cdot s_1^j \geq U_2^* && \forall k \in A_2 && (2.8)\\
& && \sum_{j \in A_1} s_1^j = 1 && && (2.9)\\
& && s_1^j \geq 0 && \forall j \in A_1 && (2.10)
\end{aligned}$$

we get mixed strategy of player 1, so we now have a Nash equilibrium.

Example 2.10. We again return to our Rock, Paper and Scissors game (example 2.6). First, from the payoff matrix can be seen that this game is zero-sum, because all utilities always sum to zero. Moreover, if we compute the expected utility value again for player 2, we would obtain $u_2(\sigma) = -\frac{1}{4}$, which is correct value according to the definition of the zero-sum games (expected utility of player 2 can be in zero-sum games computed as $u_2(\sigma) = -u_1(\sigma) = -\frac{1}{4}$).

As the game is zero-sum, we can just use the described linear program for finding of NE in zero-sum games or we deduce solution by reasoning, we discover that Nash equilibrium for the game is $(\sigma_1, \sigma_2) = ((\frac{1}{3}, \frac{1}{3}, \frac{1}{3}), (\frac{1}{3}, \frac{1}{3}, \frac{1}{3}))$. If we think for a while about the game, we will discover that the game has only one Nash equilibrium - the one we found. The pure-strategy NE doesn't exists - to any pure strategy can be easily found adequate best response, as we discussed in example 2.6, so this example shows another observation -

even in zero-sum two-player games, Nash equilibrium doesn't need to exists in pure strategies.

In the context of these findings, We state another related fundamental game-theoretical theorem, known as the *Nash's theorem* - it states that in mixed strategies, Nash equilibrium exists for all *finite games* [**?**].

2.2.5 Computing Nash Equilibria in General-Sum NFG

Example 2.11. If we get back to the example 2.2, we can see from the payoff matrix of the Stag Hunt game that this game is not zero-sum, but general-sum. So we cannot simply use our defined LP for finding of NE in zero-sum games in this game.

Because in two-player general-sum games utilities are not diametrically opposite, these games cannot be conveniently formalized as the optimization problem or form it's linear program [21].

But as mentioned before in example 2.2, in this concrete game we can simply find NE by simple enumeration and check for which strategy profiles are the required best response conditions fulfilled. And as mentioned before, another difficulty of finding NE in general-sum games is that their expected utility values doesn't need to match, so NE in this case doesn't provide any guarantees of expected outcome [3].

2.3 Infinite Games

2.3.1 Definition

In the previous section, we occasionally encountered term finite games. Finite games are defined as the games in which each player has a finite set of possible actions. So previous examples of normal-form games - Rock, Paper and Scissors (2.3) and Stag Hunt (2.2) are finite games.

The normal-form finite games are not enough to model our more complex adversarial classification setting, where player's set of pure strategies is infinite and continuous, so we use more generalized model of *infinite games* [4].

Definition 2.12 (Infinite game [4])**.** A two-player *infinite game* is a tuple (N, C, u), where:

- N is a set of two players, indexed by $i \in \{1, 2\}$
- $C = C_1 \times C_2$, where C_i is a compact metric space corresponding to the i^{th} player's set of pure strategies.
- $u = (u_1, u_2)$, where $u_i : C \to \mathbb{R}$ is a real-valued utility (or payoff) function for player i

As every finite space is a compact metric space under a discrete metric, every finite normal-form game is also an infinite game [24].

Next we generalize concept of mixed strategy to infinite games:

Definition 2.13 (Mixed strategy [27]). Let (N, C, u) be an infinite game, and for any set X let $\Delta(X)$ be the set of Borel probability measures over X. Then the set of *mixed strategies* for player i is $\Sigma_i = \Delta(C_i)$.
The set of mixed-strategy profiles is the Cartesian product of the individual mixed-strategy sets, $\Sigma_1 \times \Sigma_2$.

And next we generalize expected utility concept to infinite games:

Definition 2.14 (Expected utility [27]). Given an infinite game (N, C, u), the *expected utility* u_i for player i of the mixed-strategy profile $\sigma = (\sigma_1, \ldots, \sigma_n)$ is defined as

$$u_i(\sigma) = \int_C u_i(c) d\sigma \tag{2.11}$$

Although generalized definitions are more complex, their concept remains same as in finite games [4].

2.3.2 Computing Nash Equilibria in Infinite Games

Computation of equilibria in infinite games is recognized as complex. There arises "natural" problems with the discontinuity of utility functions, which leads to non-existence of Nash equilibria in such games, example e.g. in [24].

We need to introduce another subtypes of infinite games, which in some reasonable way limits the "bad" behavior of infinite games, e.g. we will be able to guarantee that NE can even exist or that NE can have finite support [28].

We therefore introduce *continuous games* with conditions on *compactness* of strategy space and *continuity* of utility functions.

Definition 2.15 (Continuous game [29]). A *continuous game* is an infinite game (N, C, u), where:

- $u = (u_1, \ldots, u_n)$, where $u_i : C \to \mathbb{R}$ is a **continuous** real-valued utility (or payoff) function for player i

The most important property of continuous games is that in every continuous game, Nash equilibrium must exist in mixed strategies [24].

But despite the fact that the Nash equilibrium in mixed strategies must exist in continuous games, it is still very hard to compute or even approximate it - the equilibrium measures can be too complicated [24]. We must limit these games to yield such strategy sets of players which admit "simple" descriptions [24].

Thus, we define separable games:

Definition 2.16 (Separable game [29]). A *separable game* is an infinite game with utility functions $u_i : C \to \mathbb{R}$ taking the form

$$u_i(\sigma) = \sum_{j_1=1}^{m_1} \cdots \sum_{j_n=1}^{m_n} a_i^{j_1 \cdots j_n} f_1^{j_1}(\sigma_1) \cdots f_n^{j_n}(\sigma_n) \tag{2.12}$$

where $a_i^{j_1 \cdots j_n} \in \mathbb{R}$ and the $f_i^j : C_i \to \mathbb{R}$ are continuous.

Every finite set is a compact metric space under the discrete metric and any function from a finite set to R is continuous and can be written in the form 2.12 by replacing f_i^j by Kronecker delta functions, so finite games are also separable games [28].

In this class of continuous games, it is known that Nash equilibrium exists in finitely supported mixed strategies for zero-sum games. In [28], authors show that this holds for nonzero-sum separable games as well.

2.4 Double-Oracle Algorithm for NFG

As the last game-theoretic concept, we describe the Double-Oracle algorithm used in the Double-Oracle framework for actual solving of adversarial classification game.

Adopted from decision theory and introduced by (McMahan et al., 2003), the Double-Oracle algorithm is proven to converge to Nash equilibrium, and demonstrated high performance in variety of domains [3, 20].

The goal of the algorithm is to find the Nash equilibrium of a normal-form game without actually solving the complete linear program of the game, thus allowing to solve much bigger game instances that would be otherwise possible [25].

On the beginning, the *restricted game* from the original game is created, which allows players to play only a limited set of actions. This restricted game is solved by linear programming. Then in each iteration of the algorithm, each player chooses best response to the optimal strategy of the opponent in the current restricted game. The best responses are chosen from the unchanged original game. The rows and columns of the actual restricted game are then expanded by these new computed best responses. Then, expanded restricted game is solved again and iterations continues, until no other actions are added to the game.

We show visualization of main steps of the algorithm:

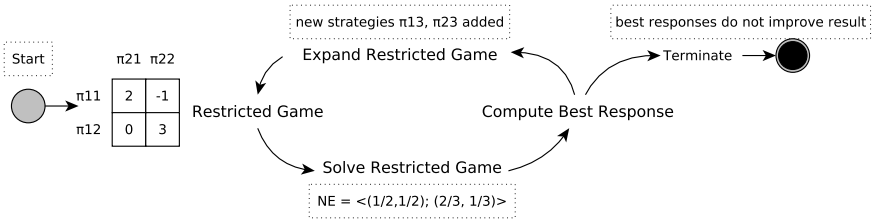


Figure 2.3: The visualization of the Double-oracle algorithm [3].

Double-oracle repeats these three steps until convergence:

1. Create a restricted game from the original game, with restricted set of pure strategies of each player

2. Find Nash equilibrium in the restricted game

3. Compute a pure best response strategy with respect to the equilibrium strategy of the other player (this pure best response strategy can be chosen from any action in the original game), and expand actual restricted game by adding this newly computed best response if possible (if the strategy already exists in the actual restricted game, we are not adding it)

If neither player can add any new action to the actual restricted game, the double-oracle *converged* (But even if only one player can add new action, double-oracle continues).

The expected utility values of the best responses of the respective player's strategies computed during individual iterations provides bounds on the value of the game of the original unrestricted game [25].

The important note is that DO algorithm can exploit *domain-specific knowledge*, since the problem of finding best responses of individual players is a *single-player optimization problem*. Because single-player optimization problem can be solved faster in general, the algorithms for finding respective best responses are termed *oracles* [3].

Algorithm runs in time polynomial to the size of the game, since solving the restricted normal-form game is of polynomial complexity and there is linear number iterations (and if oracles also computes best responses with polynomial complexity). In the worst case, double-oracle adds all actions of players to the restricted game and thus solves the original unrestricted game, but this doesn't happen often [3]. There are not known any guarantees about the number of iterations needed for convergence to the Nash equilibrium [3].

Chapter 3

Double-Oracle Framework

In this chapter, we describe the double-oracle framework we use in this thesis in detail. The original framework of Prokop Šilhavý [4] experiments with three classifiers - the decision trees, Support Vector Machines and deep neural networks. In our work, we focus only on the deep neural networks, as our goal is to leverage specific knowledge of this domain. We therefore describe framework with consideration of only deep neural network as a classifier.

We use the game-theoretical concepts from the previous chapter to define the adversarial classification problem as the game, then we describe how exactly original framework utilizes double-oracle algorithm to find a solution of this game.

3.1 Adversarial Classification Problem as a Game

We briefly described the adversarial setting we study before in the introduction. We can state the adversarial classification problem as follows (we use the same example as in the original work [4]):

Example 3.1. We are managing a part of the computer network, and we want to create a security system which detects a malicious traffic. We can measure only the size of the data transferred during one connection, but we have examples of regular traffic *available*. Let us assume that the attacker tries to transfer as much data as possible. Thus, he has a utility, which linearly grows with the size of the data. When we detect the attack, we can stop it, and the attacker transfers nothing. Finally, we can limit the maximal payload on 10 GB for simplicity.

So we have two players, the *attacker* and the *defender*. The attacker tries to transfer as big traffic as he can - he is inclined to try to choose the biggest payload possible by his utility function. The defender needs to classify if the size of the data transferred could be *attacker's point*, or it is a regular traffic - *benign point*.

As the size of the data transferred can be real valued, the attacker's set of pure strategies is infinite. The space of possible classifiers the defender can use as a pure strategies is also infinite, so the game is infinite.

We formally define the adversarial classification game as:

Definition 3.2 (Adversarial classification game [4]). *Adversarial classification game* is a two-player infinite game, where:

- N = $\{defender, attacker\}$, indexed by $i \in \{1, 2\}$
- $C = C_1 \times C_2$, where
  - C_1 is a compact metric space to the set of classifier's parameters
  - C_2 is a compact metric space to the set of attacker's pure strategies
- $P \subseteq C_2$ is a set of *benign points*
- $f : C_1 \times C_2 \to [0, 1]$ is a defender's *classification function*, where:
  - 0 corresponds to a benign point
  - 1 corresponds to an attacker's point
- $l : C_2 \to \mathbb{R}$ is a *loss function* for a benign point misclassification
- $u_2 : C_2 \to \mathbb{R}$ is an attacker's *default payoff function*
- $u = (u_1, u_2)$, where:
  - $u_2 : C_1 \times C_2 \to \mathbb{R}$ is defined as

$$u_2(c_1, c_2) = (f(c_1, c_2)) \cdot u_2(c_2)$$

  - $u_1 : C_1 \times C_2 \to \mathbb{R}$ is defined as

$$u_1(c_1, c_2) = -u_2(c_1, c_2) - \sum_{p \in P} f(c_1, c_2) \cdot l(p)$$

We can now formally instantiate the game from the example 3.1 as:

Example 3.3. Let's say that regular traffic is $\{1, 1.5, 2\}$ Then:

- $C_1 = [0, 10]$
- $C_2 = [0, 10]$
- $P = \{1, 1.5, 2\}$
- $f : C_1 \times C_2 \to [0, 1]$ is:

$$f(c_1, c_2) = \begin{cases} 0, & \text{if the defender classify the point as an attacker's} \\ 1, & \text{if the defender classify the point as a benign} \end{cases}$$

- $l(p) = 1$
- $u_2(c_2) = c_2$

Because utility functions in the defined adversarial classification game (3.2) take a sum-of-products form, it is a separable game (definition 2.16). Therefore, we have a guarantee that the Nash equilibrium in this game exists and have a finite support.

3.2 Computing Nash Equilibrium by the Double-Oracle algorithm

In this section, we describe how is the adversarial classification problem solved by the DO algorithm.

The DO can be used in the exact same form as defined in previous chapter (subchapter 2.4), although there are several differences because we now solve the infinite game.

The main loop is realized as:

1. Create a restricted game from the original game, with restricted set of pure strategies of each player - by this step, DO creates *finite normal-form* game
2. Find Nash equilibrium in the restricted game - game is now finite and in a normal-form, but is not zero-sum in general - the penalty for misclassification of benign points makes game general-sum, but by limiting this penalty by hard constraint in the linear program for the solving of zero-sum game, the game can be made zero-sum or nearly zero-sum (we define modified LP afterwards)
3. Compute a pure best response strategy with respect to the equilibrium strategy of the other player (this pure best response strategy can be chosen from any action in the original game), and expand actual restricted game by adding this newly computed best response if possible (if the strategy already exists in the actual restricted game, we are not adding it) - this step remains the same, we describe the individual oracles in the next subchapters

Restricted game in step 2 is forced to be nearly zero-sum by constraining expected false-positive rate of the defender's mixed strategies :

$$\begin{aligned}
&\text{minimize} && U_1^* && && (3.1)\\
&\text{subject to} && \sum_{k \in A_2} u_1(a_1^j, a_2^k) \cdot s_2^k \leq U_1^* && \forall j \in A_1 && (3.2)\\
& && \sum_{k \in A_2} s_2^k \cdot fp(a_2^k) \leq FP && && (3.3)\\
& && \sum_{k \in A_2} s_2^k = 1 && && (3.4)\\
& && s_2^k \geq 0 && \forall k \in A_2 && (3.5)
\end{aligned}$$

The constraint (5.4) is added to the original LP for solving zero-sum normal-form game (2.2.4), limiting the expected false-positive rate of defender's strategies by constant FP. If the constraint is chosen to be very small, the penalty for misclassification of benign points can be "neglected", making the game nearly zero-sum. Another benefit is a very low false-positive rate of the actual defender's strategies (in the original work, more means of constraining game to be nearly zero-sum was considered, but this one was considered the best [4]).

This LP yields equilibrium strategy of the defender and the value of the game (the utility U_1^* of the first player which is minimized). The dual of this program (2.2.4) yields equilibrium strategy of the attacker.

The individual oracles for the respective players in the original framework are solving single-player optimization problem:

3.2.1 Defender's Oracle

The defender's strategy space is formed by possible deep neural networks. The defender's oracle goal is to find *best response with respect to the actual attacker's strategy* - such a deep neural network that will classify nearly all benign points (in the optimal setting all) as *negative* and as much as possible attacker's points as *positive*. To train the best classifier as possible, standard classification methods with support of weighting of training points are used. Because constraint on the maximum allowable false-positive rate is in expectation (5.4), it permits the defender to play strategies with classifiers which doesn't necessarily satisfies the false-positive rate constraint.

The original framework use the artificial feedforward deep neural network - multilayer perceptron with one hidden layer. The features of a model respects the input dimensions - if the input is one dimensional, input to the network is one feature etc. The number of hidden layer's neurons is used as a parameter determining DNN's learning potential, the activation function used is the *Rectified Linear Unit (ReLU)*. The output layer has one neuron with *Sigmoid* activation function. Loss function used is *weighted cross entropy*, learning of the DNN is stopped if change of the loss is smaller than the predefined constant. The output of DNN is *soft* - float value between 0 and 1. It is rounded (with *threshold* of 0.5) to 0 or 1 before use in further computations. It can be interpreted as probability that input to the network is positive.

The architecture of the DNN is:

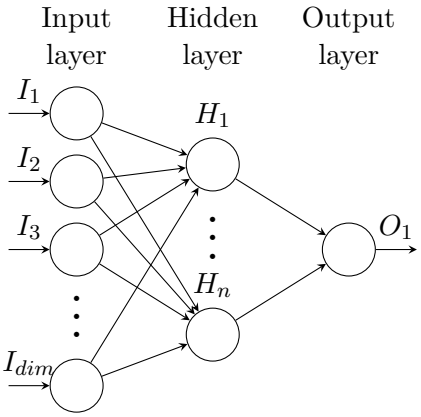


Figure 3.1: Schema of the used neural network (taken from [4].

3.2.2 Attacker's Oracle

As the attacker's strategy space is used the closed interval - e.g. in one dimension $[0, 10]$. The attacker goal is to find *best response with the respect to the strategy of the defender* - he optimizes his actual expected utility - which is calculated as initial utility of a respective point given by his utility function,

multiplied by classification outcome (0 or 1), and weighted by probability of playing respective neural network (pure strategy) in actual defender's strategy (the involved example is discussed in chapter 5). So for the optimal solution, the attacker must find the *global optimum* of a discontinuous function.

As the optimization algorithms to solve the global maximum search problem of the attacker, the original framework offers several possibilities. We consider only the one with which the original work achieved the best results, the discretization with gradient optimization (L-BFGS-B algorithm). This algorithm yields approximation of the optimal value, but it is still very accurate and more than enough for the convergence of DO. It also emerged as the fastest of all evaluated algorithms of the original framework for finding the best response of the attacker [4].

Although the L-BFGS-B optimization algorithm is the state of the art gradient optimization method, it requires many evaluations of the attacker's function, which evaluates actual tested point on all classifiers in actual defender's strategy, which is demanding and slows down the algorithm, making it the actual bottleneck of the framework [4].

3.2.3 Additional comments and observations

The DO algorithm for the infinite game can be seen as some form of smart dicretization - in each iteration DO inserts new pure strategies to the game, which corresponds to inserting new samples to the discretization [4].

As the speed of the attacker's oracle is a substantial bottleneck of the DO framework, precisely in this step we hope to improve DO the most by maximally utilizing the domain knowledge of the deep neural networks.

The DO allows for modifying of main loop - the oracles can be computed simultaneously, or they can alternate - in the original work, both types were considered but simultaneous computation has revealed better course of convergence in general [4], so we concentrate only on the simultaneous computation of the individual oracles.

Chapter 4

Adversarial Attacks and Verification Methods

In the previous chapter, we described the original DO framework and discussed it's bottleneck, which is the computational speed of the attacker's oracle. It is very slow as requires many evaluations of the possible points by the respective trained classifiers in the process of finding the optimum.

Therefore, in this chapter we study the various adversarial attacks and verification methods for the deep neural networks with lens of usability in the attacker's oracle. We hope to find methods which could be used for the optimization of the attacker's utility function, so we would utilize them to increase computational speed of the attacker best response algorithm and thus increase the scalability of the whole DO framework.

Our domain of interest are the DNNs, so we restrict our focus only on this domain, more precisely on the DNNs which perform the *classification* of an input (i.e. learning a mapping from an input x to a category y; $y = f^*(x)$, where f^* is an approximated function [15]), as our adversarial classification task is to distinguish between benign and malign points.

We would like to use the chosen methods as a *black-box* or to minimally adjust them to be compatible with the attacker's oracle, and to integrate them into the DO framework and afterwards provide experimental analysis about their performance.

4.1 Introduction

In [23], Madry et al. formulated *unifying perspective* of adversarial robustness as a theoretical framework of a robust optimization to provide an unified view on the adversarial attacks and defenses to date. They also present an effective adversarial attack and propose a strong defense, the *adversarial training*, which remains among the most trusted defenses [30, 31, 32].

We found this approach useful, as it provided valuable insights and clarifications into many emerged verification and attack methods and to the problem of generating adversarial examples, so we used it's ideas in this chapter.

We outline the basics of DNNs first, because in this and mainly in the next chapter we work with DNNs extensively. Next we outline basic properties of

adversarial attacks and define adversarial example generation problem as an *optimization problem*. In next subchapters, we discuss the individual attack and verification methods.

4.1.1 Deep Feedforward Neural Networks

The *Artificial deep feedforward neural networks* are composed of the several layers - an *input layer*, one or more *hidden layers*, and an *output layer*. Each layer has multiple nodes (*neurons*), each of these nodes are connected to the preceding layer. Every such connection is accompanied with the predefined *weight*, nodes in hidden layer and output layer have an predefined *bias* in addition. During *training*, these weights are *selected* by some *training algorithm* [15], e.g. most used *gradient descent* [33, 15]. By feeding inputs to the input layer, values are propagated through the network (at each layer values can be computed from the previous layer's values), resulting in propagating the input values to the last layer.

The values of the hidden and output nodes are computed as a *linear combination* of the node values from the previous layer and by adding the respective bias of the node, afterwards and only for the hidden nodes the *activation function* is applied (simulating the firing of neurons in the brain, as the neural networks are inspired by biology of real neurons [15]). If a network doesn't have feedback connections (connections in which the outputs of a network are fed back to itself), we say a network is *fully-connected* [15]. An example of the fully-connected DNN is in Figure 4.1.

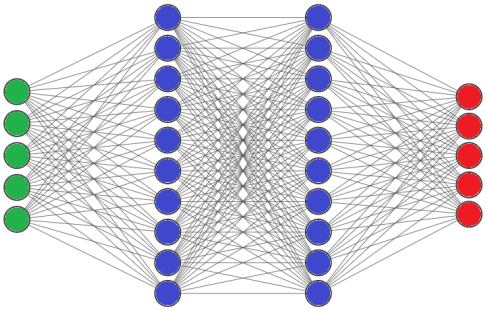


Figure 4.1: An example of a fully-connected DNN with 5 input neurons (green), 2 hidden layers each containing 10 neurons (blue) and 5 output nodes (red).

An example of an activation function is e.g. the Rectified Linear Unit - ReLU (recently the most popular activation function for deep neural networks [34]). It's defined as the positive part of its argument: $f(x) = max(0, x)$, where x is the input [34].

4.1.2 Properties of Adversarial Attacks

During the training when examples are fed through the network and weights and biases are being updated, the classifier learns a *map of the input space*. It will only learn an approximation of the true boundaries between the classes for

many possible reasons, e.g. training cannot update all parameters of a model correctly, or model family is not sufficient enough, or there wasn't enough examples during training [5]. The error between real task *decision boundaries* and model's learned decision boundaries is *exploited* by an adversary for generating adversarial examples [5]. An simplified visualization is in Fig. 4.2.

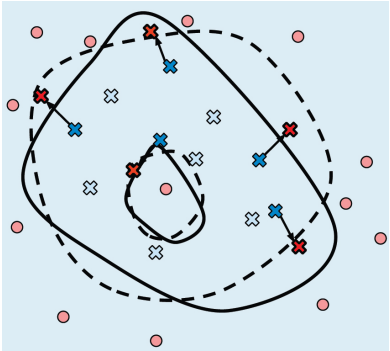


Figure 4.2: An simplified visualization of learned decision boundary and real decision boundary for a problem with two classes [5].

In Fig. 4.2, we see a boundary in the dashed line, which is the true boundary to the problem, solid line, which is learned by model. The arrows represents then the possible adversarial examples.

Adversarial attacks can be characterized as *white-box* or *black-box*, depending on the *amount of access* an adversary has to the model. In a white-box setting, the adversary has full access to a model including gradients, loss function, *hyper-parameters* etc. In a black-box setting, the attacker can access targeted model only by a limited interface, but can create a substitute model, which would mimic the original one. On this substitute, the attack can be prepared. If a substitute model is "close enough" to the attacked one, it is very much possible that the adversarial example for a substitute will be incorrectly classified by the targeted model as well [16, 17, 5].

From our game-theoretical perspective, as all defender's possible strategies are known to the attacker in our adversarial game, we *focus only on the white-box attacks*.

Adversarial attacks can also be *targeted* or *untargeted*. An untargeted attack's goal is only to change the actual *predicted class* of a classifier to any other class, which can be class "close" to the actually predicted class, e.g. for adversarial example in the introduction **??**, where original image is classified as a "pig", the "closest" adversarial example with highest loss is a "wombat" (Australian animal). In the contrast, targeted attack specifies target class in advance - the example in the introduction, which changes a "pig" to an "airliner", is an example of a targeted adversarial attack.

4.1.3 Adversarial Attacks as Optimization Problem

First, we define the *model* and a *loss function*:

Definition 4.1 (Hypothesis function [15]). Given an input space X, where k is a number of classes being predicted, the *hypothesis function* is a mapping:

$$h_\theta : X \mapsto \mathbb{R}^k \tag{4.1}$$

where $\mathbb{R}^k$ is the output space, θ represents all the *hyper-parameters* of the deep neural network, and h_θ represents entire (trained) DNN.

Definition 4.2 (Loss function [15]). Given an $k-$dimensional vector (*logit output* from a DNN) and *true labels* of an inputs, a loss function l is a mapping:

$$l : \mathbb{R}^k \times \mathbb{Z}_+ \mapsto \mathbb{R}_+ \tag{4.2}$$

where $\mathbb{Z}_+$ is a non-negative integer and $\mathbb{R}_+$ is a non-negative real number.

Because we are restricted to the classification, we use *cross entropy* [15] as a loss function.

Typically, an approach to train DNNs is to optimize its parameters (weights and biases) by minimizing the average loss over the *training data*, using the e.g. *stochastic gradient descent (SGD)* [33, 15]. The gradient of a loss with the respect to the DNN's parameters is computed, and by update step with respective *learning rate*, the DNN's parameters are iteratively optimalized [33, 15]. Adversarial attacks exploits these facts, and instead they attempt to maximize the loss:

Definition 4.3 (Maximization problem [23]). Given the model h_θ, the *sample* x, the true label y and a set of allowable perturbations Δ, the problem of finding an adversarial example can be stated as:

$$\max_{\delta \in \Delta} l(h_\theta(x + \delta), y_{true}) \tag{4.3}$$

where δ is an *perturbation*.

The set of an allowable perturbations Δ is often defined as (L_∞ metric):

$$\Delta = \{\delta : \max_i |\delta_i| \ \leq \epsilon\}. \tag{4.4}$$

In [35], authors argue that L_∞ is the optimal distance metric for adversarial example generation. We use it in our experiments, as it is a metric used in many adversarial attacks.

The maximization problem can be solved by three possible ways:

1. Find a lower bound - solve problem by heuristic - find an adversarial example.
2. Solve the problem exactly.
3. Compute the upper bounds - bound the problem by relaxation.

4.2 Finding Lower Bounds - Adversarial Attacks

The maximization problem (4.3) can be solved heuristically - any feasible δ gives the lower bound - the adversarial example. Therefore all "heuristic" methods for a generation of an adversarial examples (adversarial attacks) are trying to find a (*best possible*) lower bound [23].

Szegedy et al. [16] first discovered the adversarial examples and proposed a method to find them - using the box-constrained *L-BFGS*. This method is very successful in generating the adversarial examples, but is time consuming and impractical [36].

A fast method for adversarial attack was proposed by Goodfellow et al. in [17] - the *Fast Gradient Sign Method (FGSM)*. It was later modified by Kurakin et al. into the *Basic Iterative Method (BIM)* [37], and finally modified by Madry et al. to the *Projected Gradient Descent (PGD)* [23].

In [38], Papernot et al. proposed different type of attack using Jacobian of the trained model - *Jacobian Saliency Map Attack (JSMA)*, but this method is as time consuming and impractical as the L-BFGS [5, 36].

To the several approaches to adversarial attacks (e.g. [16, 17]), many defenses have been introduced, but in a quick succession, many proposed defenses were soon defeated by a stronger attacks or were found to perform incomplete evaluations [39, 40, 41, 42, 43]. This competition grew into an "arm race" between attacks and defenses [31, 36].

Madry et al. [23] argues that the *projected gradient descent (PGD)* is the *ultimate first-order adversary* and that *adversarial training* against it provides broader security guarantee. Authors also held an *attack challenge* that invited community to attack their trained networks on the MNIST and CIFAR10 datasets[1] in both white-box and black-box setting.

Another competition which evaluated performance of adversarial attacks (in the black-box setting) was [44]. In both competitions, the *iterative attacks* related fundamentally to the FGSM or BIM showed the best performance [44] (In Madry et al. attack challenge, plain *PGD with random restarts* showed very well performance). Moreover, [31] reveals that iterative attacks generate near-optimal adversarial examples. These findings, as well as simplicity of the implementation of these attacks (they can be written in a few lines of code) leads us to use the iterative attack as our chosen baseline attack method.

We now describe iterative gradient attacks (and attacks related to them) in a greater detail:

4.2.1 Fast Gradient Sign Method

In the canonical FGSM attack, *each component of the input* x is modified by adding or subtracting a small perturbation ϵ to create an adversarial example x^{adv}. Thus, it uses the L_∞ metric [45]. The ϵ is a hyper-parameter to be chosen.

[1] Links to the respective challenges are: `https://github.com/MadryLab/mnist_challenge`, `https://github.com/MadryLab/cifar10_challenge`.

The attack finds a solution to the 4.3 by computing the gradient of the loss function with *respect to the perturbation* δ. The gradient can be conveniently found using the *backpropagation*:

$$g = \nabla_\delta l(h_\theta(x+\delta), y_{true}) \tag{4.5}$$

The *step taken* is the chosen size of the perturbation ϵ, depending upon the sign of the respective gradient g:

$$\delta = \epsilon \cdot sign(g) \tag{4.6}$$

so the attack becomes:

$$\delta = \epsilon \cdot sign(\nabla_\delta l(h_\theta(x+\delta), y_{true})) \tag{4.7}$$

and the adversarial example is found as:

$$x^{adv} = x + \epsilon \cdot sign(\nabla_\delta l(h_\theta(x+\delta), y_{true})) \tag{4.8}$$

Therefore, the FGSM generates adversarial examples using a *linear approximation* of the target model [36, 45, 44].

4.2.2 Basic Iterative Method

Although FGSM gives quickly very good results, its disadvantage is that it makes a potentially *too big step* and fact that the DNNs are *not linear even on a small region* [37, 44].

The simple solution is to just *iterate FGSM with smaller steps* (method is also called *Iterative FGSM (I-FGSM)*):

$$x_0^{adv} = x; \;\; x_{N+1}^{adv} = Clip_{X,\epsilon}\{x_N^{adv} + \alpha \cdot (\nabla_\delta l(h_\theta(x_N^{adv}+\delta), y_{true}))\} \tag{4.9}$$

where $Clip_{X,\epsilon}$ performs element-wise clipping of x. The ϵ, learning rate α and number of iterations N are the hyper-parameters to be chosen.

The naming is not united, sometimes the PGD is called BIM or otherwise. We use the naming from Madry et al. [23] (Generally the only difference is that PGD described in next subchapter can be named BIM or I-FGSM instead).

4.2.3 Projected Gradient Descent

Madry et al. "ultimate first-order adversary" PGD algorithm (sometimes called the BIM or defined slightly differently) improves the BIM setting and generates adversarial examples under the L_∞ metric as:

$$x_0^{adv} = x; \;\; x_{N+1}^{adv} = Clip_{X,\epsilon}\{x_N^{adv} + \alpha \cdot sign(\nabla_\delta l(h_\theta(x_N^{adv}+\delta), y_{true}))\} \tag{4.10}$$

where $Clip_{X,\epsilon}$ performs element-wise clipping of x. The ϵ, learning rate α number of iterations N are the hyper-parameters to be chosen.

Another hyper-parameter is the number of *restarts*. Authors in [23] show that restarting the algorithm from a random point within the ϵ-norm ball considerably increases the performance [44]. It is because the performance of PGD is still limited by the local optima, and the added restarts can "slightly" improve chances to avoid them [23].

The algorithm can be viewed as making many smaller FGSM steps. With enough number of iterations, this attack *almost always* finds an adversarial example (even without the restarts) [44].

As this attack is relatively simple and very effective (considered as actual state of the art under the L_∞ metric [46]), we would like to use it as a *baseline heuristic method*.

4.3 Exact Solution - Complete Verification

Verifying properties of DNNs is very useful approach, because it can prove if tested DNN can be prone to *any* type of attack, as the verifying tests if input violates or satisfies given property [36].

DNN verification methods can be divided as *complete* and *incomplete* [47]. The exact set representing all final-layer activations which can be achieved by applying a bounded perturbation to the input is called the *adversarial polytope* (more about polytopes and settings of *verification problem* is in [48]) Incomplete verifiers works with an *outer approximation* of adversarial polytope, while complete verifiers works with an *actual* adversarial polytope [6, 49]. Illustration is on Fig. 4.3.

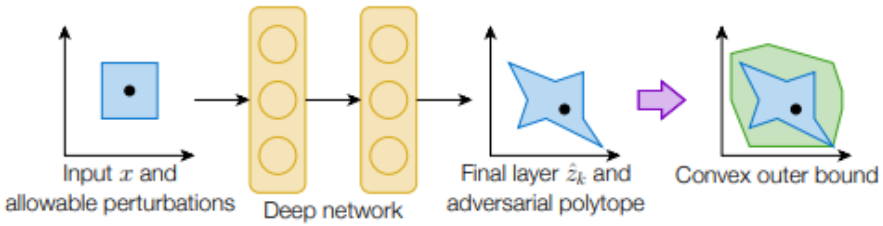


Figure 4.3: An exact adversarial polytope (second from right), and an (convex) outer approximation of the adversarial polytope (first from right) [6].

The complete verifiers give an *actual* adversarial example - the *optimal one* (more precisely, best attack under some chosen perturbation ϵ) - as they optimally solve the maximization problem (definition 4.3). Unfortunately it is for the price of high computational cost - these methods are infeasible for complete verification of a larger DNNs, because the problem is very hard (NP-complete) even for verifying of simple properties of neural networks [50].

This difficulty is caused by the *activation functions*. They render the problem *non-linear* and *non-convex* [50]. Because of this, vast majority of all verification methods reason only on a restricted subset of DNNs, and many methods require only a full-connected feedforward deep neural networks with only ReLUs as an activation functions [51, 52].

There are two main approaches for exact verification [53, 49]. One type of verifiers employ a *combinatorial optimization* to formulate maximization problem as a mixed integer linear program (MILP) [49, 54]. Another approach employs satisfiability modulo theories (SMT) [50, 55].

Several works provides review of verification methods. Bunel et al. [53] compare several baseline exact verification algorithms - including Planet [54] and Reluplex [50]. In [51], authors review primal optimization methods which are using MILP, Boolean satisfiability (SAT) and SMT. Finally [48] provide comprehensive survey of verification algorithms.

We describe the representative methods in a greater detail.

4.3.1 Reluplex

This sound and complete verification method based on the SMT solver is able to verify properties of DNNs with only ReLU activation functions, using the simplex algorithm to support the ReLU constraints. It is written in C++ and uses open-source GLPK LP solver [50].

Unfortunately, Reluplex verification is very slow and is able to verify networks up to several hundreds of nodes [56].

4.3.2 Marabou

The Marabou framework builds upon Reluplex solver and brings many enchancements in its toolbox[2]. It uses the same SMT-based techniques but improves upon Reluplex by using lazy search technique and replaces external GLPK solver by complete simplex-based linear programming core [55]. Like the previous Reluplex, it is written mainly in C++.

It extends its possibilities of application by extending support to the convolutional DNNs with arbitrary piecewise-linear activation functions, and as the Reluplex, its verification is sound and complete [55]. In the evaluation, Marabou showed better performance than the Planet verifier **??** overall, but not better performance than ReluVal [57] in general [56].

4.3.3 MIPVerify

The MILP-based methods for verifying the DNNs are recognized as a very slow, but in [49], Tjeng et al. used a tighter formulation for a non-linearities and a new presolve algorithm, and managed to scale their mixed-integer linear programming approach MIPVerify[3] to the medium-sized networks [7].

According to the experiments, MIPVerify is two to three orders of magnitude quicker than the Reluplex and can verify a network with over 100,000 units [49].

We show the MILP formulation (without the ReLU constraints) of MIPVerify for the L_∞ metric, where x is an input with true label $\lambda(x)$, x' is an adversarial example, $f_i(\cdot)$ is the i^{th} output of the network and $X_{valid} = [0,1]^m$:

[2] `https://github.com/NeuralNetworkVerification/Marabou`.

[3] `https://github.com/vtjeng/MIPVerify.jl`.

$$\min_{x'} \epsilon \tag{4.11}$$

$$\text{subject to} \quad argmax_i(f_i(x^{'})) \neq \lambda(x) \tag{4.12}$$

$$x^{'} \in X_{valid} \tag{4.13}$$

$$\epsilon \geq x_j^{'} - x_j \tag{4.14}$$

$$\epsilon \geq x_j - x_j^{'} \tag{4.15}$$

MIPVerify can handle arbitrarily feedforward DNNs with layers that use linear transformations - fully-connected, convolution, and average-pooling layers, and with layers that use piecewise-linear functions - ReLU and max-pooling layers [49].

As the MIPVerify is a *state-of-the art* in the MILP-based exact solvers [58] and uses well studied combinatorial optimization methods (where we can relatively easily modify the method itself in case of light incompatibility) we would like to use this method as a *baseline exact verification method.*

4.3.4 Worst-case Adversarial Attack

In the Madry adversarial learning tutorial[4], there is an interesting formulation of the *worst-case adversarial attack*, where an attacker is trying to perform a targeted attack - change the class label from the true class y to the target class y_{targ} within some bounded ϵ. The problem can be stated as follows:

$$\min_{z_1,\dots,d+1} \quad (e_y - e_{y_{targ}})^T z_{d+1} \tag{4.16}$$

$$\text{subject to} \quad \|z_1 - x\| \leq \epsilon \tag{4.17}$$

$$z_{i+1} = max\{0, W_i z_i + b_i\}, \qquad i = 1, \dots, d-1 \tag{4.18}$$

$$z_{d+1} = W_d z_d + b_d \tag{4.19}$$

where attack is under the L_∞ metric, e_i means the unit basis (with a one on i-th position), ϵ is an perturbation, x is an input, max represents the ReLU activation function, d is number of layers, i is index of layer z, W_i, b_i are respective hyper-parameters of the DNN, the ouput of DNN is the $z_{d+1} = h_\theta(x)$ (where h_θ is a DNN model 4.1). The output of h_θ are the class logits.

The result of this problem gives the worst possible adversarial attack. If the result is *positive*, then the attack did not found any possible adversarial example within the bounded region ϵ. If the result is *negative*, then the *logit value* of the "attacked" class y^{targ} is lower than true class y and adversarial example is found. Exact full formulation of the MILP model is in madry tutorial.

[4] `https://adversarial-ml-tutorial.org/adversarial_examples/`.

This MILP method can handle arbitrarily feedforward full-connected DNNs with the ReLU activation functions and also convolutional neural networks (CNNs).

This MILP-based exact verification method is similar to the MIPverify, but is not scaled too well. It uses again combinatorial optimization methods, and its great advantage is that it is simple and can be easily modified if need, so can be great *exact baseline method* for testing if MILP methods are even feasible or work as intended.

4.4 Finding Upper Bounds - Incomplete Verification

In the previous subchapter, we discussed various exact methods for the verification of DNNs. These algorithms are slow because verification of DNN properties is NP-complete problem [50]. On the contrary, the incomplete verification algorithms sacrifice "exactness" to improve computational efficiency [7].

Incomplete verification algorithms are more efficient, but they "reason" over an *outer approximation* of the adversarial polytope (Fig. 4.4), meaning these approaches *may not anwer* (aswer may not be decidable) every query about the adversarial polytope [49, 7, 6]. That means that these methods typically doesn't find the *actual adversarial example* [49, 7].

As mentioned before, many verification algorithms and methods are reviewed in [48], which also offers performance experiments.

We will now describe families of various baseline methods:

4.4.1 Convex Relaxations

Convex-based relaxations are the among the most popular methods of the incomplete verification [32].

Instead of finding an exact worst adversary on given perturbation, these methods compute *tractable bound*, using *convex relaxations* of the ReLU constraint [6]. An illustration is on Fig. **??**.

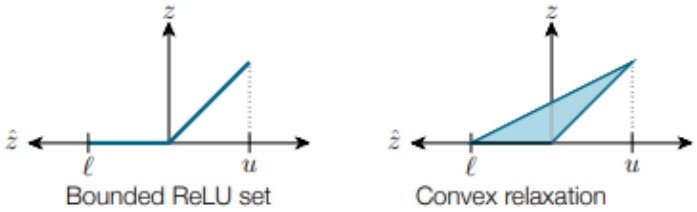


Figure 4.4: An illustration of the convex relaxation of ReLU [6].

Method of [6] uses *duality* to find even better bounds than with only convex relaxations. In [59], authors extends and generalizes this work.

Using a single *semidefinite program* (SDP), duality and convex relaxations,

[60] finds an upper SDP bound on the adversarial loss. In [61], the semidefinite relaxations turns ReLU constraints into a *quadratically constrained quadratic program* (QCQP), then this QCQP is relaxed into an SDP.

In [62], authors formulate dual convex optimization problem and use subgradient methods to solve it.

Another SDP method is proposed in [7], where various properties of activation functions are abstracted using *quadratic constraints*. These quadratic constraints are later used to formulate verification problem as the SDP. An illustration of derivation of an quadratic constraints is in Fig. 4.5.

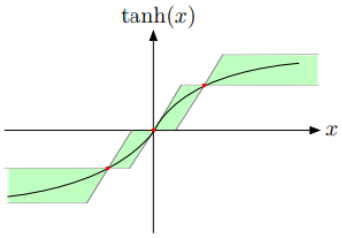


Figure 4.5: An illustrative example of deriving the quadratic constraints for the tanh function [7].

Finally, in [47] authors unify all LP-relaxed verifiers in a convex relaxation framework and performs large-scale experiments on deep ReLU networks to find insights about the gap between the lower-bounds (found using PGD attack [23], subchapter 4.2.3) and upper-bounds (found by [6], subchapter 4.4.1). The resulting bounds are also compared with exact solution from MILP [49] (subchapter 4.3.3).

As the convex relaxation methods are among the most studied incomplete DNN verification methods and there are many individual interesting works (e.g. [6], [61]), we would like to use these methods as a baseline.

4.4.2 Abstract Interpretations

Another approach is to use an *abstract domain* (represented by logical formulas) to approximate the reachable set at each layer [63]. They use *zonotope*, which is an symmetric polytope [48]. This method is called *Ai2* (Abstract Interpretation for Artificial Intelligence), and is incomplete [48].

The work is build upon and following works introduce *DeepZ* [64], *DeepPoly* [65], *RefinePoly* [66] and *RefineZono* [67]. All these approaches are included in the toolbox *ERAN* (ETH Robustness Analyzer for Neural Networks)[5].

4.4.3 Bound Propagation

These incomplete methods "carefully" propagate bounds through a network. *Neurify*[6] [68] combine symbolic interval analysis and linear relaxation.

[5] `https://github.com/eth-sri/eran`.
[6] `https://github.com/tcwangshiqi-columbia/Neurify`.

In toolbox *CROWN-IBP*[7] [32], authors combine relaxation-based verification bound (*CROWN*) [69] with efficient interval bound propagation (*IBP*) [8] (illustrative example of IBP is on Fig. 4.6) and focuses on its use in training of robust models.

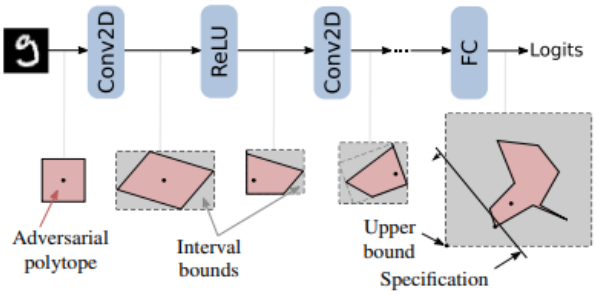


Figure 4.6: An illustrative example of interval bound propagation from IBP method [8].

Although later updated too, the CROWN algorithm replaced techniques *Fast-Lin* [70] (network relaxation) and *Fast-Lip* [70] (Lipschitz estimation) of the same author, used before.

[7] `https://github.com/huanzhang12/CROWN-IBP`.

Chapter 5

Compatibility Framework

In the previous chapters, we have described the DO framework and its bottleneck - attacker's oracle, next we discussed and reviewed methods for adversarial attacks and verification of DNNs, and we have also highlighted some methods which we would like to choose as a baseline methods for further experiments and integration into the DO framework.

In this chapter, we discuss and describe possible compatibility of these methods with reward functions and strategic constraints in the attacker's oracle of the DO framework, and create an compatibility framework which enables easy use of many such methods.

5.1 Methods

As there are three options how to find a solution to the adversarial example generation problem (definition 4.3), we thought it would be convenient to experiment with at least one of each type.

As the heuristic method, we have chosen the *Projected gradient descent (PGD)* attack by Madry et al. [23] (subchapter 4.2.3), as the exact method we have chosen *MIPVerify MILP* method by Tjeng et al. [49] (subchapter 4.3.3) or the interesting simple formulation of *Worst-case Adversarial Attack* (adversarial tutotial by Madry et al.) (subchapter 4.3.4), and as the method upper bounding the problem (incomplete verification), we have chosen any from the *convex-relaxation* family (e.g. Wong and Kolter [6] or Raghunathan et al. [61], subchapter 4.4.1).

The critical question is now, how to use these methods for the computation of the attacker's best response in the DO. One possibility is simple and straightforward - we can attack the defender's networks itself.

5.2 Direct Attack

5.2.1 Outline

We will show a possible scenario of a straightforward use of the adversarial attacks on an example:

Example 5.1. As described in subchapter 3.2, in each iteration of DO, the defender *trains a new DNN* as his best response to the previous attacker's strategy - his generated point he played last iteration of DO. The attacker's oracle computes his best response in the reaction to the previous defenders strategy - his chosen classifier or classifiers weighted by according defender's support values. Then, if for simplicity we will assume that the defender plays only the *pure* strategies (so he always "plays" only one DNN, i.e. plays only one DNN with the probability 1), we can perform an *untargeted*[1] adversarial attack on this DNN to try to change the defender's DNN's output (e.g. by using PGD attack). If the output of the defender's classifier would change (e.g. from classifying a point as a malign to classify it as a benign), the attacker's utility would also certainly change. But even if the adversarial attack would be successful and manage to change the output of the defender's classifier, it is still *not enough* for us, as there is also an attacker's utility function - i.e. the resulting adversarial example within the chosen distance from the original can be classified as benign, but it's utility evaluation for the attacker can be low or even zero, making such adversarial point useless. Therefore, successful crafting of an adversarial example - the straightforward use of adversarial attacks - is not enough for us.

5.2.2 Attack Setup

We can still use the "straightforward" adversarial attack, i.e. to attack directly the defender's DNN, but we must also take into consideration the attacker's utility function.

If we would choose the PGD method to attack (and we would still anticipate that the defender is playing only the pure strategies), we can perform the attack as follows: We will use the standard PGD hyper-parameter setting, only we will modify value of maximal possible perturbation ϵ - as the size of the space of the attacker's strategies, e.g. the size of interval $[0, 10]$ is 10. If we now perform the attack, the PGD algorithm will then try to change the class of the original input, i.e. it will iteratively "move" in the direction of greatest loss with respect to the original input. As we are trying to find the global optimum, we can also set number of iterations to the arbitrary high value (we will end in a global or a local optima anyway).

Now we need to include the attacker's utility function. We can simply resolve this problem by evaluating the actual PGD point in each iteration of PGD - after the next step of PGD is taken, we evaluate the new actual point and store it's actual utility value. By this, we can evade the attacker's utility problem and "remember" the best found actual utility.

The last thing we need is to generalize this attack setting into realistic scenario when the defender use the mixed strategy and in his support are several classifiers (pure strategies). In fact, as discussed in the ending of the DO chapter (subchapter 3.2.3), the defender have nearly always the

[1] We *can* because defender's DNN classify points as adversarial or benign, so there are only two classes.

full support. This can be viewed as a difficult problem, but we can again circumvent it - because adversarial attacks can transfer well between the "close" models [16] (discussed in subchapter 4.1.2), we can easily choose such a model with the highest probability in the defender's support, perform the PGD attack against this model and generate an adversarial point with our "computed" highest attacker's utility. As adversarial attack provides transferability, the other classifiers in defender's strategy will also "probably" misclassify the adversarial point and the attack then would be close to the *best response*.

5.3 Utility Estimation Net

We will depict the problem of the attacker's best response computation in a greater detail.

5.3.1 Modelling Attacker's Best Response by DNN

From the definition 3.2, the attacker's best response is defined as:

$$u_2(c_1, c_2) = (f(c_1, c_2)) \cdot u_2(c_2) \tag{5.1}$$

where $u_2 : C_2 \to \mathbb{R}$ is an attacker's *default payoff function*, $u_2 : C_1 \times C_2 \to \mathbb{R}$ is attacker's *actual utility* function, $f : C_1 \times C_2 \to [0, 1]$ is a defender's *classification function* where 0 corresponds to an attacker's point, 1 corresponds to a benign point.

We can view the equation 5.1 as a setting where the defender plays only the pure strategies. The key observation is that the attacker's actual utility value depends on the multiplication of the probability of detection by the defender's classifier $\times$ utility (reward) function for the attacker's actual point $(u_2(c_2))$. Our new best response algorithm needs to reflect this property.

In the previous subchapter about direct adversarial attack, the attacker's best response did not properly reflect this property (in fact, it evaded this property altogether).

We will resolve this issue by *modelling exactly this attacker's function* (which we want to optimize) by the *neural network* - we will create a *DNN* which will be *trained to estimate the attacker's actual utility as accurately as possible*, i.e. the neural network will have the same inputs as the equation 5.1 and also the same (as accurate as possible) output. The *key idea* is to *attack* this DNN by adversarial attacks (because DNN will now model the multiplication in the equation 5.1), it's gradients would "lead us" exactly to the maximum of this multiplication - to the optimum we need.

So we need to train a DNN model (we call it the *Utility estimation net, UEN*) to learn the equation 5.1, so we would to be able to perform an attacks on it afterwards.

5.3.2 Basic Structure of Utility Estimation Net

Basic Schema

The input of UEN consists of the same values that are required to compute actual attacker's utility - the actual point, the attacker's utility function, the defender's classification function and his actual (mixed) strategy. The actual point is the only *input* of the UEN - the number of input features of the network is equal to the dimension of an input point. In the DO framework, individual defender's pure strategies are represented by individual trained NNs of the defender - we need to *incorporate* these NNs into the UEN to be able to correctly estimate the attacker's actual utility. We incorporate them as a *black-box* - we take the defender's NNs which represents his pure strategies which are in the support of his actual strategy and use them to *construct* the actual UEN (implementation details are in appendix B). The weighting of individual defender's strategies in support of mixed strategy is also incorporated into the UEN (appendix B). The output consists of only one neuron - we estimate the real value - the attacker's actual utility, therefore UEN performs an *regression*, not a classification. The basic schema where defender plays only pure strategies is depicted in the Fig. 5.1.

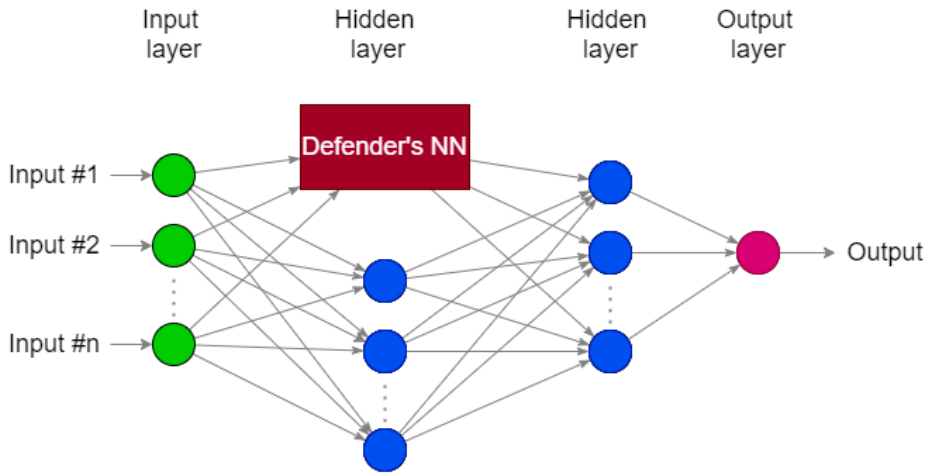


Figure 5.1: The schema of UEN with n input neurons (green) as features, two hidden layers with custom number of neurons (blue) and one neuron (red) in output layer. The defender's pure strategy is "incorporated" into the UAE roughly the same as is depicted on the schema (in the schema defender plays only one pure strategy with probability 1).

Multiplication Net

The two hidden layers of UEN perform an *multiplication* between the input point *evaluated by attacker's utility function* and the probability that the input point is benign (the output of the defender's NN - rounded to 0 or 1).

The multiplication network is separately trained to learn both the multi-

plication and the attacker's function. It has two separate inputs - the input point which enters the network in the leftmost input layer, and the second input which is the probability that the input point is benign mentioned before, which enters the multiplication net in the second hidden layer. After the multiplication net is learned, the complete utility estimation net can be build by attaching the multiplication net and the defender's NNs into it, to perform the attacker's actual utility estimation.

The schema of the multiplication net is depicted in the Fig. 5.2.

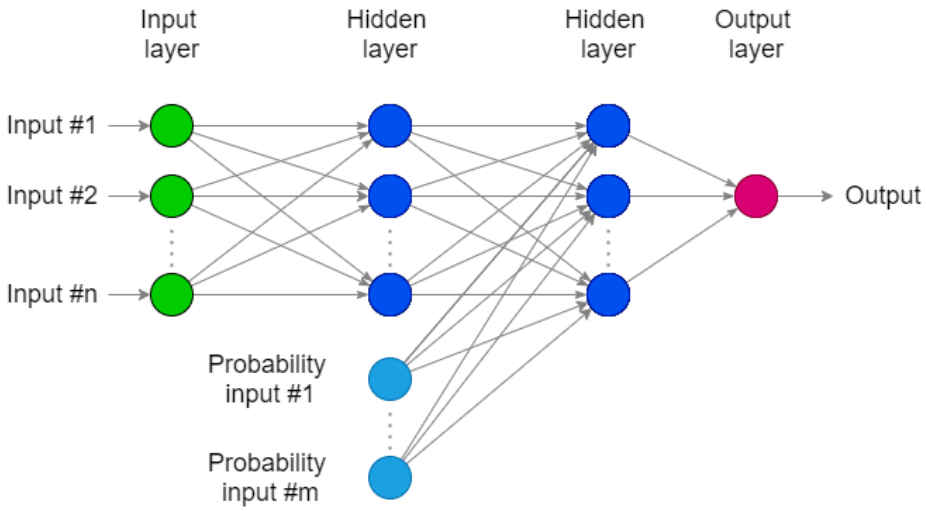


Figure 5.2: The schema of Multiply net with n input neurons (green) as features, two hidden layers with custom number of neurons (blue) and one neuron in output layer (red). The probability inputs connected to the second hidden layer represents possible pure strategies of the defender (teal nodes).

5.3.3 Support Size Minimization

We have described the UEN and its ability to effectively estimate the value of the actual attacker's utility, but given the fact that the defender's pure strategies represented by NNs must be directly incorporated into the UEN, the size of the defender's support cannot be equal to the number of his pure strategies (full support), as suach problem make the UEN creation infeasible, as it would mean incorporating hundreds of networks. Because of this, we try to reduce the size of the support.

The actual support of the defender is computed by LP from chapter 3 (3.1). We will use the value of the game computed by this LP, create a similar model with the same constraints, fix the maximal utility in these constraints by computed value of the game and we will connect the probabilities of playing the respective strategies to the binary variables, and we will minimize this sum in the objective.

$$\text{minimize} \quad \sum_{x \in A_2} x_k \qquad \forall k \in A_2 \qquad (5.2)$$

$$\text{subject to} \quad \sum_{k \in A_2} u_1(a_1^j, a_2^k) \cdot s_2^k \leq U_1^* \qquad \forall j \in A_1 \qquad (5.3)$$

$$\sum_{k \in A_2} s_2^k \cdot fp(a_2^k) \leq FP \qquad (5.4)$$

$$\sum_{k \in A_2} s_2^k = 1 \qquad (5.5)$$

$$s_2^k \geq 0 \qquad \forall k \in A_2 \qquad (5.6)$$

$$s_k \leq x_k \qquad \forall k \in A_2 \qquad (5.7)$$

$$x_k \in \{0, 1\} \qquad \forall k \in A_2 \qquad (5.8)$$

where U_1^* is the value of the game computed before. The LP is extended by constraints 5.7 - which makes connection between binary variables and probabilities of defender to play respective pure strategies and 5.8, which defines binary variable x and changes LP to MILP. The objective 5.2 is changed to minimization of sum of the binary variables x.

Unfortunately, the complexity of this MILP problem (5.2) is np-complete, which can create another bottleneck in the DO, but it permits us to use the UEN.

5.3.4 Compatibility with PGD Attack

We outlined the basic structure of UEN, now we describe how to use our chosen PGD attack to perform the adversarial attack on it.

Using PGD Attack

We can now attack the UEN model (absolutely trivially) by the *untargeted PGD attack*. The PGD attack will try to maximize the loss for the current "class" - if we set the current "class" label of the PGD attack as a 0, then PGD attack will try to maximize the loss for the *mean squared error (MSE)* loss function (MSE is the standard loss function for an regression tasks) and yield such an adversarial example that will have maximized loss for "class" 0. Therefore, because we are estimating utility of the attacker and we need this value as high as possible, this setting will give us an adversarial example with the maximum possible value of the output from the utility estimation net, therefore exactly the result of the optimization we need.

We have an adversarial attack, now we describe how to build the UEN to be compatible with this attack. Since heuristic attacks doesn't have much restrictions in the structure of the attacked network, we do not need to adjust the UEN structure for these types of attacks much (it also means that when we create the compatible UEN for the PGD adversarial attack, the PGD can be *very easily replaced* by any other compatible attack).

Building Compatible UEN

The PGD attack is not limited to any specific activation functions (they only need to propagate gradient) and our feedforward DNN structure from the Fig. 5.1 is absolutely sufficient. But thanks to this, there are some details we can *exploit*.

As described in subchapter 3.2.1, the defender's NN returns the probability of a point being attacker's or benign. Its output as defined in the adversarial game 3.2 is attained by rounding the value (threshold 0.5). As the attacker's actual utility function requires this rounded value, it is not clear how to attain this rounding by UEN. Because PGD needs only the gradients to perform the attack, we can use the gradient-compatible functions from the PyTorch framework (using torch.autograd) [19] to dodge this problem. This allow us to use the exact rounding without a need to resort to the other options. The weighting of the pure strategies of the defender is done in the same manner - we use gradient-compatible functions to implement the weighting.

We do not need to make any more compatibility adjustments for the PGD attack and we can use actual utility estimation net for the experiments. In every iteration of the DO framework, when the best response for the attacker must be computed, the new utility estimation net is build from the prepared multiplication net (we train this net for various dimensions and attacker's utility functions ahead) and the defender's networks (according to his mixed strategy and support).

The basic algorithm how to build the UEN for the PGD attacks can be outlined as follows: We train the multiply net on according dataset. Next we define the UEN in pytorch as a plain net accepting multiply net and list of nets and list of probabilities of "playing" these nets. Next, we in each iteration of DO we build UEN again, from prepared learned multiply net, and from the pure strategies of the defender (his NNs). Thanks to pytorch, we don't need to build the complete full-connected net, but we can use the "mock" UEN to connect our learned models together. We can use standard pytorch functions which works with tensors to multiply output values from defender's NNs as weighting of his pure strategies.

5.3.5 Compatibility with MIPVerify and Worst Case MILP

To satisfy the compatibility requirements for the MIPVerify and Worst Case MILP, we need to make *many more* adjustments. As mentioned in subchapter about MIPVerify (MIPVerify), this MILP method works only for the layers that use linear transformations, e.g. fully-connected DNNs, CNNs, with the ReLU activation function only. This is not so much restricting for us, as defender's NN use only the ReLU layers. As fully-connected DNNs with ReLU activation functions is *common denominator* for nearly all verification methods (complete or incomplete), if we manage to adjust the compatibility with this type of network, we again create a compatible framework for many possible methods (indeed, if we make it compatible with MIPVerify, the worst case MILP will work as well).

Transform into Classification

The first big problem we encountered is that these methods work only for the *classification*, and our utility estimation net performs the regression (they are build for this type of DNN, because major testing datasets are the MNIST and the CIFAR10, as testing on an images is very popular). So to make the UEN compatible, we need to *transform* the regression into the classification. We can use standard idea - the discretization - we will divide the output interval into the sections and all uncountable values in these sections will belong to the same class. It doesn't matter for us, because we are opting for the maximum value from the UEN, therefore our target class (both the MIPVerify and the worst case MILP supports targeted attack) can be chosen as the highest possible class beyond possibility (e.g. 11), and MILP verifiers will give us the maximum possible value *regardless* (They will try to change the class to the 11, which corresponds to maximum value of 10).

Therefore, we can simply divide the output interval into 11 sections. The 0-1 interval, 1-2 second etc. to the 10-11, which will be the last.

Rounding Problem

We now have UEN performing the classification, but there is yet another big problem - this time, we must make the *rounding* of the output of the defender's NNs without the help of external functions. This problem is hard, as there are not many possibilities how to solve it, as the solution must be transferable into the fully-connected DNN with only ReLUs. We managed to overcame it with the use of the Threshold net - we will create the additional network, which will be trained to round the *logit* output of the defender's network - to 0 or 1 with the threshold 0.5 (as we cannot use defender's sigmoid functions in the output layer). By this strategy, we can bypass even this problem.

Building Fully-connected UEN

The last problem is that the UEN must be now fully-connected without even any skip connections (we used the skip connections in the UEN compatible with the heuristic family of the methods, because it allowed us to make less complex implementation). The solution is again simple but *demanding* - we will create a *plain untrained* fully-connected DNN with ReLUs which will represent the final UEN. Next, we will prepare all component networks needed - the possible many defender's NNs, on each of this defender's network we will *add* the threshold net layer, and finally the multiplication network (transformed into the classification, as this network directly make up for the output layers of UEN). Then, we need to *carefully transform* all necessary weights and biases from all these component networks (even the defender's), and precisely *change the connections* in such a way that the final UEN will have the precisely *same logit output vector* as the combination of these networks (this property can be verified).

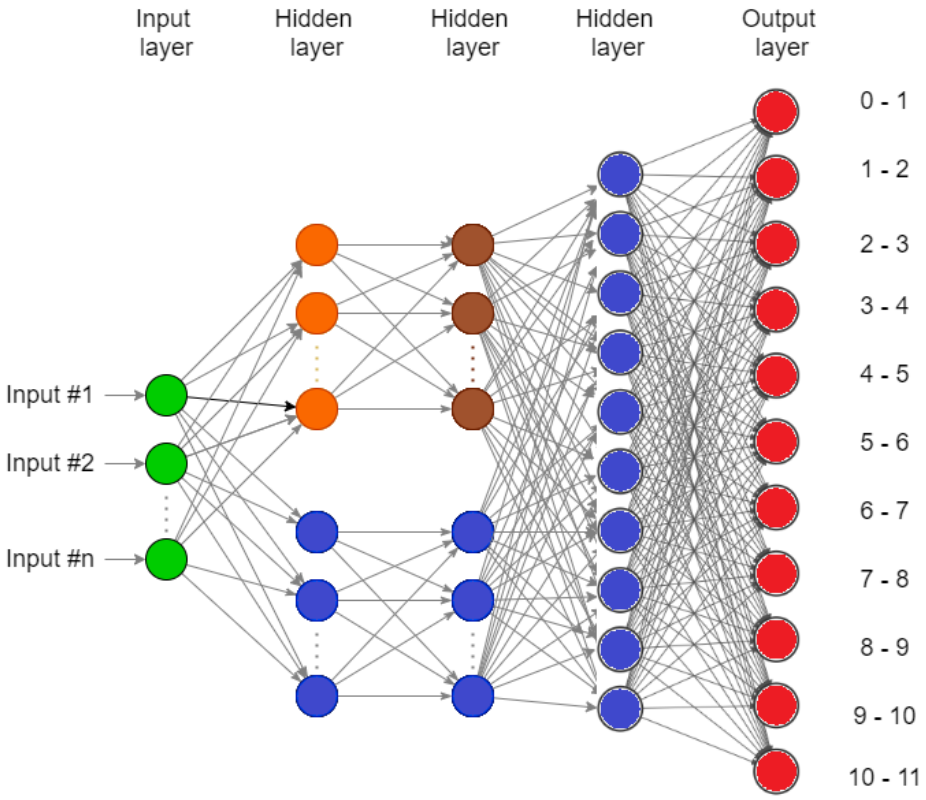


Figure 5.3: The schema of the utility estimation net performing an classification, with n input neurons (green) as features, three hidden layers and eleven neurons (red) in the output layer (classes). This network represents the defender playing only pure strategies. The orange nodes represents the defender's NN. The brown nodes represents threshold net. Orange and brown nodes together make the DetectionThreshold net (with input weights). Blue nodes together with red nodes represents the multiplication network (with input weights). On the schema, weights with predefined zero value are not depicted. the schema therefore represents the actually copied weights from all required networks. The output layer has marked the individual points which belongs to the respective class.

The tranforms done to the models are simple in nature, but can be complex to program for working with arbitrary number of pure strategies. The algorithm is the simple reconnecting the connections of component networks to the UEN for the classification. All component networks must have the adequate neuron sizes. The copying works as working with graphs, and we help ourselves from situations where the connections pass forward through bias by removing bias from output layers of component networks, which allows us to freely multiply weight values of connections and even connecting connections of arbitrary networks together by mupliplying them between themselves and this new weight we assign to new connection created from them.

The final UEN with described components is depicted in Fig. 5.3.

As can be seen in Fig. 5.3, this time we had to make *many changes* to the

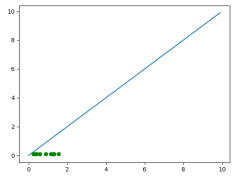
baseline UEN. We had to add the *additional layer*, and the output layer was redone to reflect the classification. Again, in each iteration of the DO this network has to be again build from its components, although we can prepare the important threshold net and the multiplication net and train them ahead.

Chapter 6

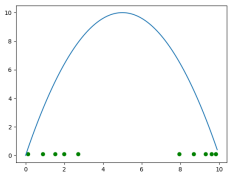
Experimental Analysis

In this chapter, we experimentally evaluate the compatibility framework and the individual optimizing methods we integrated into the DO framework.

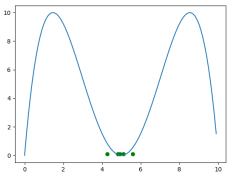
We first introduce the used datasets for the experiments, then we describe our framework settings we used, and then the individual experiments and discussion.



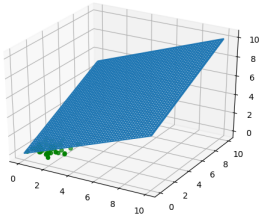
(a) : Linear function with benign points.



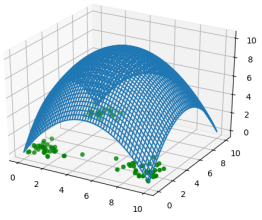
(b) : Quadratic (in ine dimension) attacker's function. The benign points are only on the sides.



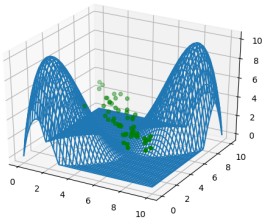
(c) : The benign points are in the middle of the two attacker's extrema.



(d) : Linear function with benign points.



(e) : The benign points are in the "corners" of the attacker's utility function.



(f) : The two extrema function, benign points are in the middle.

Figure 6.1: The example of attacker's functions and the benign points [4].

6.1 Experimental Data

We use the same datasets as in the previous work [4], as they comes incorporated with the DO framework. It is also convenient to test new optimalization methods on the same data, because it allows for easy comparison of the results.

The data are defined by dimension and the type of attacker's function.

There are three types of attacker's function - the linear, one with local maximum (quadratic in one dimension), and function with two maxima (these traits remain in every dimension).

The individual attacker's functions are dispayed on the Fig. 6.1

All functions are defined on the interval as the adversarial game 3.2 - $[0, 10]$, and the range is normalized also in $[0, 10]$.

The benign data are randomly generated, and they form a normal distributions in the lower values of utility function [4]. The individual functions are defined as:

$$f(x) = \frac{1}{dim} \sum_{n=i}^{dim} x_i \qquad \text{(Linear function)}$$

$$f(x) = 10 - \frac{\sum_{n=i}^{dim} (x_i - 5)^2}{dim \cdot \frac{5}{2}} \qquad \text{(Function with one maximum)}$$

$$f(x) = \max \left\{ -\frac{\sum_{n=i}^{dim} ((x_i - 5) \cdot dim)^4 - 25 \cdot dim \cdot \left(\sum_{n=i}^{dim} (x_i - 5) \cdot dim \right)^2}{\frac{625}{40} \cdot dim^5}, 0 \right\}$$

(Function with two maxima)

The number of the benign points being generated is computed by the formula:

Dataset 1	Dataset 2	Dataset 3
$10 + 40 \cdot (dim - 1)$	$5 + 30 \cdot (dim - 1)$	$5 + 50 \cdot (dim - 1)^2$

Table 6.1: The formulas for the number of benign points being generated generated [4].

The depiction of the individual attacker's utility functions, with generated benign points is displayed on the Fig. 6.1.

Using the specialized dicretization, the exact solutions to the game with the defined datasets was computed before in [4]. We outline it here for easy comparison in Table 6.2.

dim	linear	one extreme	two extremes
1	1.39609	7,11946	0,79075
2	2,00348	6,22406	0,92138
3	1,84211	5,95752	0,87073

Table 6.2: The exact values of the Nash equilibria from [4].

The equilibria above are computed using the FP rate of 0.01. We use the same rate in all experiments

6.2 DO Framework Settings

For our experiments, we set the DO framework to the highest performing setting described in [4]. We use the simultaneous computation of the respective best responses in DO, as in [4] was showed that simultaneous best response computation converges better. We also use nearly the same settings for the learning of the defender's NN. The defender's NN is trained by weighted cross entropy, and is trained until the loss falls under the threshold value of 0.01, as this setting showed better results previously (DO do not fails) [4].

The training itself is performed in loops of 100 iterations, in each iteration the NN learns for the 1000 epochs, and afterwards is checked for the utility value. We do not use the weights from a previous training as an initialization of weights in next training, as is used in the previous work [4], but we use randomized weights as initialization. From our experiments, such learned network fares better and converges with higher values of DO - which favors the defender. We set weights of all benign points during training to the 1, as this setting was used in the previous work [4].

Because we have found a way how to minimize the size of the support in the game, we use it to compute the minimized strategy of the defender in all our experiments. We have tested the effect of the minimized support of the defender on the DO convergence and its value in the original framework. In our tests with original framework and optimization methods, the results stays the same. The size of the unminimized support and minimized support are shown for the comparison in the Fig. 6.2 for the input dimension pf 2 and in the Fig. 6.3 for the input size of 3.

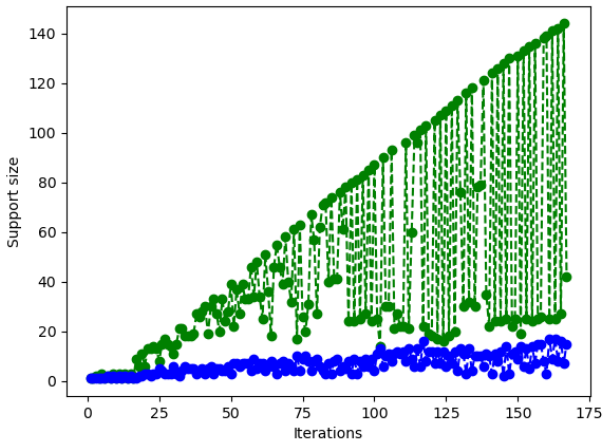


Figure 6.2: The size of the support in the respect to the number of iterations for PGD attack on estimation net, with the two-dimensional linear utility.

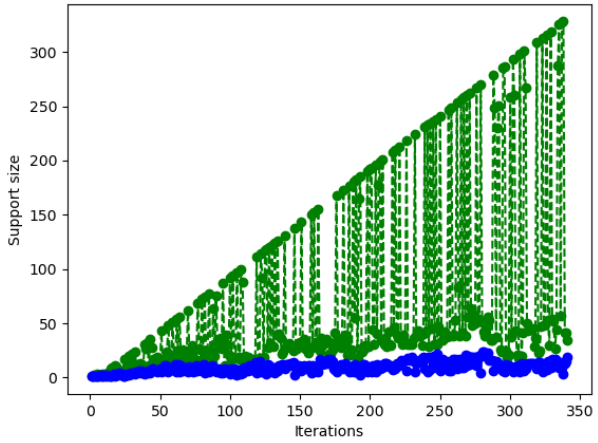


Figure 6.3: The size of the support in the respect to the number of iterations for PGD attack on estimation net, with the three-dimensional linear utility.

6.3 Direct Attack

The direct PGD heuristic attack performs well in the one dimension with the linear utility. But its performance worsens rapidly in higher dimensions. The PGD hyper-parameters are the learning rate (or step), number of iterations, the maximal perturbation under the L_∞ norm and number of restarts. In our case of bending the algorithms and use them in domains for which they wasn't designed for, we can view the restarts anywhere in our interval, as we can set the $\epsilon = 10$. Then, if we can view the benign points as the perfect restart points. We set the high number of iterations, the 1000, and we set the step relatively small as $\alpha = 0.01$.

In the initialization of the PGD direct attack, we moreover inject the one point - the maximum of the current optimized function - computed by the other exact methods of DO, to the set of starting points (The original points). The reason is that for the other than the linear case, from our experiments follows that on the beginning of DO, the PGD for the direct attack struggles under the other attacker's utility functions than the linear, e.g. it doesn't have a good starting point, as other benign points are in the beginning far from the optimum.

In the Fig. 6.4, it is shown the course of convergence for 100 iterations of DO for dimension 2, the linear utility and the parameters are $\alpha = 0.01$, $\epsilon = 10$, 1000 iterations of PGD and the three restarts. The PGD which uses the calculation of the actual attacker's utility in each step is computationally demanding, so we must restrict the number of restarts. The second option we experimented with was the setting when the number of restarts was 8 (other parameters the same), but we stopped the PGD iteration when it encountered loss after the step, as it means it has found the local minima. This modification rapidly accelerate computational speed (much less iterations

are made), but because the PGD in the direct attack makes the step in the direction of the greater loss of the classifier - in this case the detection NN, then it steps "blindly", because takes into account only the probability of detection, and not the utility of the attacker or the multiplication of the utility and the probability. The course of convergence of second case for 100 iterations of DO is on Fig. 6.5.

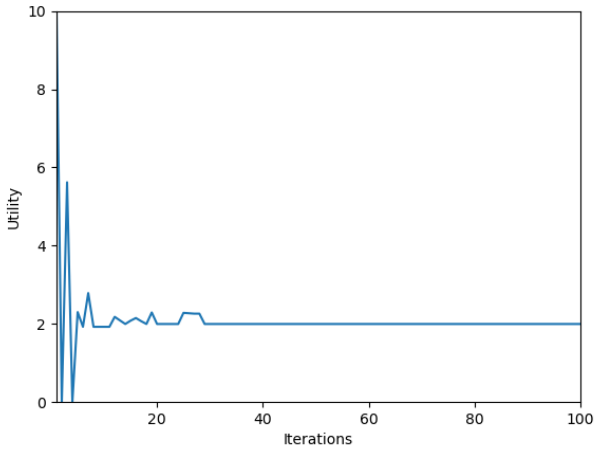


Figure 6.4: The convergence of PGD *direct attack* with the two-dimensional linear utility, for 100 DO iterations, with only 3 restarts. The average time needed for the one PGD direct attack best response was 0.56 sec. The defender's training took 2968 sec.

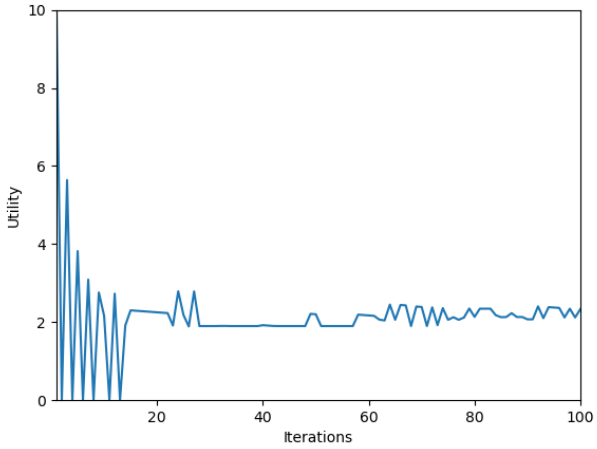


Figure 6.5: The convergence of PGD *direct attack* with the two-dimensional linear utility, for 100 DO iterations, with only 8 restarts but stopped after PGD encountered loss in a value of the actual point. The average time needed for the one PGD direct attack best response was 8.74 sec. The defender's training took 5606 sec.

6.3.1 Advantages and disadvantages and discussion

The PGD direct attack method is too slow, thanks to the frequent evaluations of the attacker's actual utility, which assess the PGD search. On the other hand, it can find the heuristic best responses reasonably efficiently and it seems to be close to the converge, although in our experiments, after several hours the DO got over 150 iterations with no evidence of final convergence.

The undeniable advantage of this type of heuristic direct attack is the simplicity to use - we do not need to train any NNs or make any complex compatibility adjustments.

6.3.2 Training of NNs

For the our compatibility framework, we need to train many NNs. These NNs provide basis of the algorithms, because their estimations must be as accurate as possible to provide credible results. Therefore, we carefully train the each one of these NNs.

The setting same for the training of all the NNs needed to assemble the respective utility estimation networks (UENs) is using the rule of thumb [15] of 5000 training examples with respective labels, divided into the training data of 4000 and test data 1000 for cross-validation [15].

We need to train the specific UEN for the each dimension as we need to define the input features for the UEN, each specific attacker's utility function and for the size of support which the UEN will support. E.g., to train for the linear utility on the dimension 2, we need one UEN, the if we want to allow the support of maximum size of 5, we need 5 specific NNs, as if we would train the one supporting the five probability inputs, and for example return zeroes for the inputs we don't use, it would generate additional unwanted noise for our experiments.

As the UEN for the PGD estimation net attack and UEN for the MILP methods are structurally similar, the PGD UEN can have much larger individual number of neurons, as the PGD scales even to the large networks. An UEN for the PGD attack have two hidden layers (Fig. 5.2) with 200 neurons each - we experimented with different sizes, but we observed best loss (MSE loss) for this values. For the defender's NNs we incorporate into UEN, we use size of 10 neurons for the experimental purposes (with such a low value, we can concatenate ten or more such networks).

The UEN for the verification methods is very small - as small as possible, because we use the methods which do not scale well. The individual sizes we use in the experiments are - the defender's NN = 5 neurons, threshold net - same as defender's NN, 5 neurons, the multiplication net for the classification problem has three layers - we use values 5, 5, and 10 for the respective hidden layers 1, 2 and 3, with 11 additional neurons in the output layer for the classes.

We carefully verified the propagation of an inputs through the respective UEN networks - to test that our algorithm is correct. We implemented several methods for this purpose in the DO framework.

6.4 Estimation net Attack using PGD

For the PGD attack on the UEN, we have experimentally chosen these hyper-parameters: We set the number of iterations of the PGD to the 1500 - arbitrary large number, because we stop the iterations prematurely as soon as the PGD falls into the local extrema (we use the same technique, we evaluate the actual point of the PGD again in the loop, save it and compare it to the last one, if the last one is greater, we stop). The number of restarts is arbitrary - the higher the number, the more powerful adversary. We therefore choose number 5 in our experiments, as we found this number of restarts balanced between performance and the speed of the computation. The next parameter is the ϵ - we set it to the value of 10 as before for in the direct PGD attack, as we bound the outputs of PGD attack (to not leave the defined intervals), and moreover we do not observed the often deviations from defined bounds. The last parameter is the learning rate, which we again set the same as in the case of direct attack - we observed that this value is big enough for the step and at the same time not too big to endanger the convergence of DO. Our experimental results are shown in tables 6.3, 6.4, 6.5, 6.6, 6.7, 6.8.

	1	2	3
1	26	39	248
5	67	153	335

(a) : Number of the iterations

	1	2	3
1	1.397	2.073	1.969
5	1.496	2.203	2.137

(b) : Final value of the game

Table 6.3: Experimental results for the PGD estimation network attack linear utility and individual dimensions in columns and size of the maximum allowed best response on the rows. All results are for the defender's Nn with 10 neurons.

	1	2	3
1	3.510	3.177	2.469
5	2.510	6.253	3.829

(a) : Average time needed for best response (sec)

	1	2	3
1	78.7	336.8	2508.8
5	403.4	2908.68	4563.7

(b) : Time needed for the defender's training (sec)

Table 6.4: Experimental results for the PGD estimation network attack with linear utility and individual dimensions in columns and size of the maximum allowed best response on the rows. All results are for the defender's NN with 10 neurons.

We did not encountered any serious issues during experiments with the PGD estimation network attack.

We show the comparison of the algorithm from the original DO (discretization with the gradient optimization) and the PGD estimation network attack in Fig. 6.6 and Fig. 6.7.

	1	2	3
1	13	43	186
5	50	50	287

(a) : Number of the iterations

	1	2	3
1	7.474	5.982	6.385
5	6.970	6.558	7.155

(b) : Final value of the game

Table 6.5: Experimental results for the PGD estimation network attack with one-maxima utility and individual dimensions in columns and size of the maximum allowed best response on the rows. All results are for the defender's NN with 10 neurons.

	1	2	3
1	2.126	10.859	2.819
5	3.226	3.140	4.651

(a) : Average time needed for best response (sec)

	1	2	3
1	51.7	247.1	4068.4
5	181.6	315.3	4636.1

(b) : Time needed for the defender's training (sec)

Table 6.6: Experimental results for the PGD estimation network attack with one-maxima utility and individual dimensions in columns and size of the maximum allowed best response on the rows. All results are for the defender's NN with 10 neurons.

	1	2	3
1	8	37	N
5	5	28	N

(a) : Number of the iterations

	1	2	3
1	3.620	4.284	N
5	5.492	4.740	N

(b) : Final value of the game

Table 6.7: Experimental results for the PGD estimation network attack with two-maxima utility and individual dimensions in columns and size of the maximum allowed best response on the rows. All results are for the defender's NN with 10 neurons.

	1	2	3
1	2.136	1.318	N
5	2.433	1.053	N

(a) : Average time needed for best response (sec)

	1	2	3
1	51.5	300.8	N
5	4.1	170.5	N

(b) : Time needed for the defender's training (sec)

Table 6.8: Experimental results for the PGD estimation network attack with two-maxima utility and individual dimensions in columns and size of the maximum allowed best response on the rows. All results are for the defender's NN with 10 neurons.

6.4.1 Advantages and disadvantages and discussion

The algorithm is fast, simple and the random restarts fits very well with the benign points in our adversarial settings. It scales well in the first three dimensions, as the values of the average BR duration seems like random. As it is a heuristic attack, it strongly depends on the current settings in the individual experiments, so number of iterations of individual PGD restarts varies, so the time of average duration of BR. The size of allowed support

greatly affects the performance (in positive way), and from the result can be observed that with increasing ability to perform better best responses grows the need for the DO iterations.

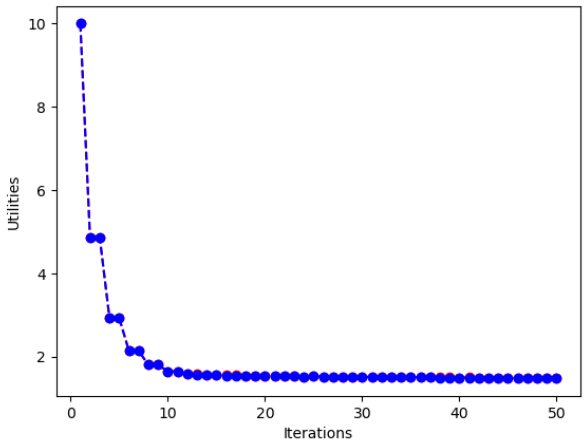


Figure 6.6: The difference between the attacker's actual utility values for PGD estimation network attack and discretization with the gradient optimization (original) algorithm on 50 iterations of DO in dimension 1, linear utility, for PGD with 5 restarts.

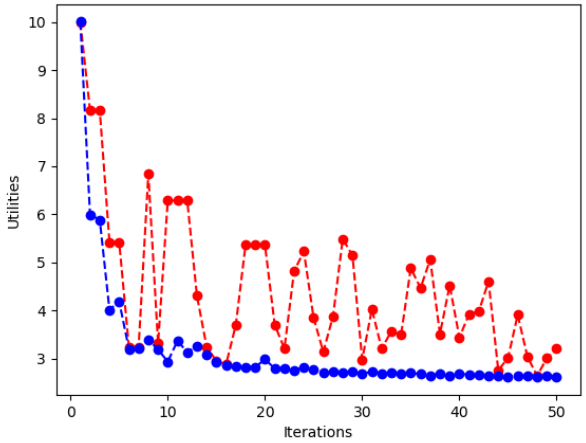


Figure 6.7: The difference between the attacker's actual utility values for PGD estimation network attack and discretization with the gradient optimization (original) algorithm on 50 iterations of DO in dimension 2, linear utility, for PGD with 5 restarts.

The disadvantages are unreliability and bad performance when a grater "leap" from the original point to the adversarial point is needed. The additional computational "reliability" can be gained only from the randomization.

This algorithm showed new bottleneck of the DO - as the size of game grows, the time needed for calculations of minimized support grows rapidly. In the experiment which ended in timeout - tables 6.3 and 6.4, the time needed for the computation of minimized support is two fold as high as the rest of the algorithm combined (over 8000 sec).

6.5 Estimation net Attack using MILP

Evaluating the MILP worst-case adversarial attack 4.16 was not easy. As we mentioned earlier, we build the UEN for the verification algorithms smallest as possible, because the worst-case adversarial attack MILP doesn't scale so well. This holds the major disadvantage, as so small number of neurons negatively affects the performance of the NN. The best loss (cross-entropy) on the test set we achieved was 0.1464.

We set the parameters of the algorithm as following: In each start of the algorithm, the current label of the point we start the attack (we choose the last attacker point as the starting point in each run of the algorithm, except the first start - in this case, we start from the middle of the defined game interval - 5 for the first dimension) is detected by propagating the sample to the UEN, then the source of attack is defined as this label, the target is chosen as the one step higher class label, e.g. for label of 4, targeted class would be 5. By this heuristic, we minimize chances of worst-case attack to find badly learned parts of the UEN - during the evaluation, we often observed that worst-case MILP finds the best adversarial example - but this example is often not the real optimal adversarial example, because the UEN is not estimating the actual attacker's utility too well.

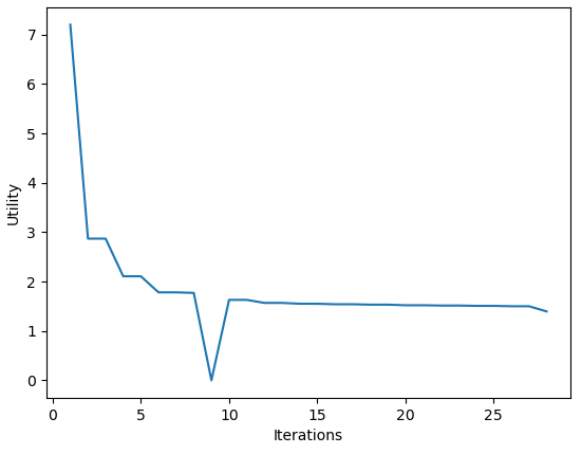


Figure 6.8: The utility values of MILP *Worst-case* attack with the one-dimensional linear utility and maximum allowed support size of 3.

The another parameter - the ϵ, we heuristically determine for the each dimension. For the dimension 1, the $\epsilon = 2$ worked best for us. For the

dimension 2, the $\epsilon = 5$ worked well. If the ϵ is not set by such a way, the MILP has tendency to find wrong adversarial sample - one for which the UEN wrongly predicts better utility than the real optimal adversarial example.

As we cannot learn the UEN with more neurons due to the scalability of the worst-case MILP, we had difficulties with the performance of the method. We show the worst-case MILP utility values of individual points for dimensions 1 and 2 in the Fig.6.8 and Fig. 6.9.

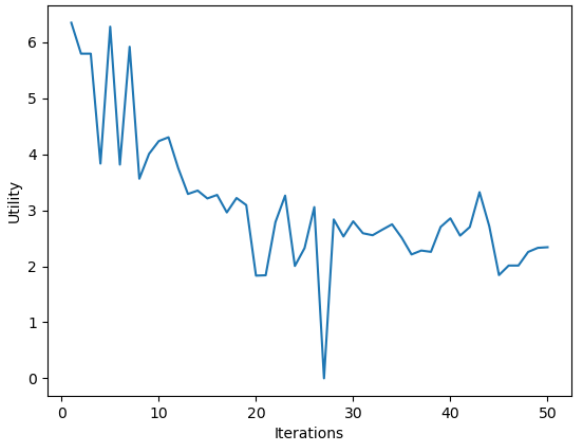


Figure 6.9: The utility values of MILP *Worst-case* attack with the two-dimensional linear utility and maximum allowed support size of 3.

6.5.1 Advantages and disadvantages and discussion

Nevertheless, the advantages are the speed of the algorithm - it doesn't need any iterations or search like the heuristics, but if the UEN is sufficiently small, the resulted adversarial example is computed instantaneously.

The disadvantage is the great dependency on the quality of the learning of the UEN. The small models we experimented with were sufficient, but only with heuristic approach. It scales only to the small models - tens of neurons.

6.6 Issues

We have implemented MIPVerify method (4.3.3) ahead of the MILP worst-case adversarial attack MILP method, but we were unable to properly experimentally evaluate it - it turned out that the MILP model which the method uses isn't compatible with our objective - we need to find the adversarial example on the target class, but we need the best possible values of this example even if the adversarial example will not have exactly the target label - exactly this objective has the MILP model we use - subsection 4.16. But the MIPVerify has in the MILP model the condition that the solution is infeasible if the target class is not achieved (4.11), which is a major obstacle -

we can only find adversarial example when it is really *possible* to change label to the target class. We cannot efficiently evaluate experiments under that condition - we could discretize the classes of the UEN even more, to 100 and try by e.g. binary search which adversarial example could be achieved, but because MIPVerify provides careful preprocessing to handle bigger NNs, it takes a long time to compute even for small networks, making such a method infeasible.

6.7 Other methods

We also wanted to try the methods from the incomplete verification "family". But there are difficulties with these methods - many of them are designed for training the robust models, and "it is enough for them" to compute only the estimated upper bound on the adversarial sample, and use this bound during training [7]. Many methods are reasoning over an outer approximation of the adversarial polytope (subsection 4.4), e.g. even if they could give the actual adversarial sample, it would be computed from the over-approximation of the NN's activations - it does not correspond to the real NN activations.

Therefore, we preferred testing of the other methods.

6.8 Implementation Details

The implementation details are in Appendix B. All experiments were run on Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz 2.50GHz, 8.00 GM RAM, 64-bit OS.

Chapter 7
Conclusion

In this thesis, we study the problem of finding the Nash equlibria in the infinite adversarial games using the double-oracle algorithm. We have outlined the basics of game theory, used these definitions and state the adversarial problem as the game. We described how to solve this game using the double-oracle algorithm and described workings of existing DO framework which solves it.

We have outlined the bottleneck of this framework, the attacker's oracle. We studied adversarial attacks and verification methods in hope that we could use this domain-specific knowledge of NNs to accelerate the search for attacker's best response.

We have evaluated and reviewed these methods and their possible compatibility with utility functions and strategic constraints and created a compatibility framework, which could be used to many other families of similar methods as well and integrated three of them into the DO framework to perform the experimental analysis.

We were surprised that the projected gradient descent adversarial attack proved to be so very effective in this settings, as PGD estimation network attack proved ideal for our adversarial setting with the benign points, which serve as the perfect starting point for generating new adversarial samples.

The PGD attack itself allowed us to test a little higher dimensions - to learn that we have yet another bottleneck in the DO framework.

Nevertheless, the attack itself proved efficient and scales well, and thanks to our compatibility frameworks, it is very easy to integrate another methods of finding an adversarial examples into the double oracle.

7.1 Future Work

Thanks to the compatibility framework, the opportunity for testing other powerful methods from the fast growing adversarial learning domain presents itself. The future work on the DO framework could include another less computionally complex ways how to minimize defender's support, to further scale the algorithm, because the PGD showed to perform very well in this setting, showing great performance and unparalelled scalability.

Appendix A

Bibliography

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[2] B. Skyrms, "The stag hunt," *Proceedings and Addresses of the American Philosophical Association*, vol. 75, no. 2, pp. 31–41, 2001.

[3] B. Bosanský, "Iterative algorithms for solving finite sequential zero-sum," 2014.

[4] P. Šilhavý, "Using double oracle algorithm for classification of adversarial actions," 2019.

[5] P. M. I. Goodfellow and N. Papernot, "Making machine learning robust against adversarial inputs," *Communications of the ACM*, vol. 61, pp. 56–66, Jul 2018.

[6] J. Z. Kolter and E. Wong, "Provable defenses against adversarial examples via the convex outer adversarial polytope," *CoRR*, vol. abs/1711.00851, 2017.

[7] M. Fazlyab, M. Morari, and G. J. Pappas, "Safety verification and robustness analysis of neural networks via quadratic constraints and semidefinite programming," 2020.

[8] S. Gowal, K. Dvijotham, R. Stanforth, R. Bunel, C. Qin, J. Uesato, R. Arandjelovic, T. A. Mann, and P. Kohli, "On the effectiveness of interval bound propagation for training verifiably robust models," *CoRR*, vol. abs/1810.12715, 2018.

[9] I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, and V. Shet, "Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks," *arXiv e-prints*, p. arXiv:1312.6082, Dec. 2013.

[10] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, "Deepface: Closing the gap to human-level performance in face verification," in *2014 IEEE*

Conference on Computer Vision and Pattern Recognition, pp. 1701–1708, June 2014.

[11] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[12] O. Vinyals, I. Babuschkin, W. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. Agapiou, M. Jaderberg, A. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, pp. 1–5, 2019.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[14] F. Ansari, S. Erol, and W. Sihn, "Rethinking human-machine learning in industry 4.0: How does the paradigm shift treat the role of human learning?," *Procedia Manufacturing*, vol. 23C, pp. 117–122, 04 2018.

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

[16] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," 12 2013.

[17] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *CoRR*, vol. abs/1412.6572, 2015.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[19] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[20] M. Jain, D. Korzhyk, O. Vaněk, V. Conitzer, M. Pechoucek, and M. Tambe, "A double oracle algorithm for zero-sum security games on graphs," in *AAMAS*, 2011.

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

[22] P. Dasgupta and J. Collins, "A survey of game theoretic approaches for adversarial machine learning in cybersecurity tasks," *AI Magazine*, vol. 40, p. 31–43, Jun 2019.

[23] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations*, 2018.

[24] N. Stein, "Characterization and computation of equilibria in infinite games," 02 2008.

[25] B. Bosansky, V. Lisy, M. Lanctot, J. Cermak, and M. Winands, "Algorithms for computing strategies in two-player simultaneous move games," *Artificial Intelligence*, vol. 237, 04 2016.

[26] J. v. Neumann, "Zur theorie der gesellschaftsspiele," *Mathematische Annalen*, vol. 100, pp. 295–320, Dec 1928.

[27] J. Rehbeck, "Note on unique nash equilibrium in continuous games," *Games and Economic Behavior*, vol. 110, pp. 216–225, 2018.

[28] N. D. Stein, A. Ozdaglar, and P. A. Parrilo, "Separable and low-rank continuous games," *International Journal of Games Theory*, vol. 37, pp. 475–504, 12 2008.

[29] N. D. Stein, P. A. Parrilo, and A. Ozdaglar, "Characterization and computation of correlated equilibria in infinite games," in *2007 46th IEEE Conference on Decision and Control*, pp. 759–764, Dec 2007.

[30] A. Shafahi, M. Najibi, A. Ghiasi, Z. Xu, J. P. Dickerson, C. Studer, L. S. Davis, G. Taylor, and T. Goldstein, "Adversarial training for free!," *CoRR*, vol. abs/1904.12843, 2019.

[31] N. Carlini, G. Katz, C. W. Barrett, and D. L. Dill, "Ground-truth adversarial examples," *CoRR*, vol. abs/1709.10207, 2017.

[32] H. Zhang, H. Chen, C. Xiao, S. Gowal, R. Stanforth, B. Li, D. Boning, and C.-J. Hsieh, "Towards stable and efficient training of verifiably robust neural networks," 2019.

[33] A. Shrestha and A. Mahmood, "Review of deep learning algorithms and architectures," *IEEE Access*, vol. 7, pp. 53040–53065, 2019.

[34] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *CoRR*, vol. abs/1710.05941, 2017.

[35] *Adversarial Perturbations of Deep Neural Networks*, pp. 311–342. 2017.

[36] X. Yuan, P. He, Q. Zhu, R. R. Bhat, and X. Li, "Adversarial examples: Attacks and defenses for deep learning," *CoRR*, vol. abs/1712.07107, 2017.

[37] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," *CoRR*, vol. abs/1607.02533, 2016.

[38] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pp. 372–387, March 2016.

[39] N. Carlini, A. Athalye, N. Papernot, W. Brendel, J. Rauber, D. Tsipras, I. J. Goodfellow, A. Madry, and A. Kurakin, "On evaluating adversarial robustness," *CoRR*, vol. abs/1902.06705, 2019.

[40] A. Athalye, N. Carlini, and D. A. Wagner, "Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pp. 274–283, 2018.

[41] N. Carlini and D. A. Wagner, "Adversarial examples are not easily detected: Bypassing ten detection methods," *CoRR*, vol. abs/1705.07263, 2017.

[42] N. Carlini, "Is ami (attacks meet interpretability) robust to adversarial examples?," *CoRR*, vol. abs/1902.02322, 2019.

[43] N. Carlini and D. A. Wagner, "Defensive distillation is not robust to adversarial examples," *CoRR*, vol. abs/1607.04311, 2016.

[44] A. Kurakin, I. J. Goodfellow, S. Bengio, Y. Dong, F. Liao, M. Liang, T. Pang, J. Zhu, X. Hu, C. Xie, J. Wang, Z. Zhang, Z. Ren, A. L. Yuille, S. Huang, Y. Zhao, Y. Zhao, Z. Han, J. Long, Y. Berdibekov, T. Akiba, S. Tokui, and M. Abe, "Adversarial attacks and defences competition," *CoRR*, vol. abs/1804.00097, 2018.

[45] H. Khedher, M. Ibn Khedher, and M. Hadji, "Mathematical programming approach for adversarial attack modelling," 02 2021.

[46] W. Brendel, J. Rauber, M. Kümmerer, I. Ustyuzhaninov, and M. Bethge, "Accurate, reliable and fast robustness evaluation," 2019.

[47] H. Salman, G. Yang, H. Zhang, C. Hsieh, and P. Zhang, "A convex relaxation barrier to tight robustness verification of neural networks," *CoRR*, vol. abs/1902.08722, 2019.

[48] C. Liu, T. Arnon, C. Lazarus, C. W. Barrett, and M. J. Kochenderfer, "Algorithms for verifying deep neural networks," *CoRR*, vol. abs/1903.06758, 2019.

[49] V. Tjeng and R. Tedrake, "Verifying neural networks with mixed integer programming," *CoRR*, vol. abs/1711.07356, 2017.

[50] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," *CoRR*, vol. abs/1702.01135, 2017.

[51] F. Leofante, N. Narodytska, L. Pulina, and A. Tacchella, "Automated verification of neural networks: Advances, challenges and perspectives," *CoRR*, vol. abs/1805.09938, 2018.

[52] H. Tran, S. Bak, W. Xiang, and T. T. Johnson, "Verification of deep convolutional neural networks using imagestars," *CoRR*, vol. abs/2004.05511, 2020.

[53] R. Bunel, I. Turkaslan, P. H. S. Torr, P. Kohli, and M. P. Kumar, "Piecewise linear neural network verification: A comparative study," *CoRR*, vol. abs/1711.00455, 2017.

[54] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," *CoRR*, vol. abs/1705.01320, 2017.

[55] G. Katz, D. A. Huang, D. Ibeling, K. D. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljic, D. Dill, M. J. Kochenderfer, and C. W. Barrett, "The marabou framework for verification and analysis of deep neural networks," in *CAV*, 2019.

[56] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Towards proving the adversarial robustness of deep neural networks," *Electronic Proceedings in Theoretical Computer Science*, vol. 257, p. 19–26, Sep 2017.

[57] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," *CoRR*, vol. abs/1804.10829, 2018.

[58] K. Y. Xiao, V. Tjeng, N. M. Shafiullah, and A. Madry, "Training for faster adversarial robustness verification via inducing relu stability," 2019.

[59] E. Wong, F. R. Schmidt, J. H. Metzen, and J. Z. Kolter, "Scaling provable adversarial defenses," *CoRR*, vol. abs/1805.12514, 2018.

[60] A. Raghunathan, J. Steinhardt, and P. Liang, "Certified defenses against adversarial examples," 2020.

[61] A. Raghunathan, J. Steinhardt, and P. Liang, "Semidefinite relaxations for certifying robustness to adversarial examples," *CoRR*, vol. abs/1811.01057, 2018.

[62] K. Dvijotham, R. Stanforth, S. Gowal, T. A. Mann, and P. Kohli, "A dual approach to scalable verification of deep networks," *CoRR*, vol. abs/1803.06567, 2018.

[63] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 3–18, 2018.

[64] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev, "Fast and effective robustness certification," in *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, Curran Associates, Inc., 2018.

[65] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," vol. 3, Jan. 2019.

[66] G. Singh, R. Ganvir, M. Püschel, and M. Vechev, "Beyond the single neuron convex barrier for neural network certification," in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, Curran Associates, Inc., 2019.

[67] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "Robustness certification with refinement," in *International Conference on Learning Representations*, 2019.

[68] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Efficient formal safety analysis of neural networks," *CoRR*, vol. abs/1809.08098, 2018.

[69] H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel, "Efficient neural network robustness certification with general activation functions," 2018.

[70] T.-W. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, D. Boning, I. S. Dhillon, and L. Daniel, "Towards fast computation of certified robustness for relu networks," 2018.

Appendix B

Framework Source Code

We have created a compatibility framework and integrated it with three optimization methods into the DO framework. The framework uses the Python 3.6. The used libraries with versions are listed in Table B.1

Library	version
Python[1]:	3.6.2
NumPy[2]:	1.16.1
SciPy[3]:	1.1.0
CVXOPT[4]:	1.2.0
scikit-learn[5]:	0.23.1
PyTorch[6]:	1.7.1
matplotlib[7]:	3.0.3
glpk[8]:	4.6.5

Table B.1: The version of software used in the framework

The abstract class optimizationInterface of the DO framework is expanded by the optimizationPgdDetection (direct PGD attack), optimizationPgdEstimation.py (PGD estimation network attack) and optimizationMILP (Worst-case MILP attack). These optimization methods are directly incorporated into the DO framework and can be run from the main.py.

In the optimization folder, the NN models folder contains the implementation of the compatibility framework. In the models folder, the individual NN models are saved for further use. The adversarialAttacks contains test implementations of various adversarial algorithms and served as testing ground. The NN model is an abstract class for the individual classes, which manages individual components of UEN - detectionNN, multiplyNN, thresholdNN, utilityEstimationNN - each manages the exact component from the algorithm.

[1] `https://www.python.org/`
[2] `https://www.numpy.org/`
[3] `https://www.scipy.org/`
[5] `https://cvxopt.org/`
[6] `https://scikit-learn.org/`
[7] `https://pytorch.org/`
[8] `https://matplotlib.org/`
[8] `https://www.gnu.org/software/glpk/`

the trainNeuralNetworks contains training scripts to train individual NNs. The traningData contains a few saved datasets for the experimental training.

```
Usage of the script:

python main.py function points fp_threshold algorithm *params
        [optimizer] [step] [weights]

function:       0 -> Linear utility
                1 -> Utility with one maximum
                2 -> Utility with two maxima

points:         a path to the *.npy file with the benign points

fp_threshold:   float - a number between 0 and 1 to limit
                        a false-positive rate
                None  - algorithm expects classifier
                        with hard false-positive constraint

algorithm:      discretization -> discretization algorithm
                SVM            -> Double Oracle with SVM classifier
                NN             -> Double Oracle with neural network
                DT             -> Double Oracle with decision tree

optimizer:      0 -> discretization optimizer
                1 -> Basin-Hopping optimizer with discretization
                2 -> Basin-Hopping optimizer

step:           0 -> simultaneous computation of the attacker's BR
                1 -> alternating computation of the attacker's BR
                2 -> simultaneous computation of the attacker's BR
                        on weighted a few strategies in history

weights:        0 -> benign points has weight 1
                1 -> benign points has weight 1/n



*params:        depends on the algorithm settings

    discretization:
        density: int  - density of sampling
                 None - exact computation

    SVM:
        degree: int - degree of polynomial kernel

    NN:
        hidden_size:    int   - number of neurons in hidden layer
        epochs:         int   - number of epochs before check of
                                classifier utility
        iterations:     int   - number of repetitions of training
                                and utility checks
```

```
    last for init:  bool  - initialize the NN with weights
                            from previous training
    gradient:       float - required descent of loss,
                    None  - the addition of the first better

    verbose:        bool  - sets which logging level is used -

    basinHoppin:    bool  - use basinhoppin algorithm for

    additionalOpt   int     0 -> PGD direct attack
                            1 -> PGD estimation network attack
                            2 -> MILP worst-case attack

DT:
    max depth:  int   - a maximal depth of the tree
                None  - unlimited
    gradient:   float - required descent of weighted
                        misclassification change,
                None  - unlimited
                First - the addition of the first better
```

Appendix C

CD Content

The enclosed CD contains following files and directories:

- **skoumond.pdf** - The text of this thesis
- **text_source** - The source code of the text
  - **appendices** - The appendices of the work
  - **chapters** - The chapters source code
  - **img** - The figures
  - **specification** - The specification of the thesis
- **data** - The datasets used for the experiments
  - *The generated datasets*
  - **generate_dataset.py** - The script for generation of datasets
- **framework** - The source code of the framework

The text source code is written in LaTeXusing the template CTUstyle created by Petr Olšák[1].

For generation of new datasets, you can use the script generate_dataset.py.

```
Usage of the script:

python generate_dataset.py generator dimensions name

generator:      0 -> First dataset
                1 -> Second dataset
                2 -> Third dataset

dimensions:     the number of dimensions of the points

name:           name of the generated file
```

[1] http://petr.olsak.net/ctustyle.html