Zeroth Order Optimization Based Black-box Attacks to Deep Neural Networks

D. Clementel, P. Fantuz, F. Grimaldi, D. Solimini

Università degli studi di Padova
Course: Optimization for Data Science
Lecturer: F. Rinaldi

June 27, 2020

Contents

1 Introduction

In the last few years deep neural networks (DNNs) have achieved state-of-the-art performances in many fields (e.g. computer vision, NLP, ...), thanks to emerging research. The use of DNNs for sensitive activities however has brought out security concerns. Recent studies[1] have demonstrated that these networks can be vulnerable to adversarial examples: this means that a slightly modified image can be generated to fool a well trained image classifier. It is not hard to see that this type of attack could have a huge impact in many applications, such as traffic sign identification for autonomous driving and malware prevention.
Ideally an adversarial example should be made as indistinguishable from the original example as possible in order to fool the targeted DNN and, if possible, humans too. However the best metric for evaluating the similarity between a benign example and a corresponding adversarial example is still an open question and may vary in different contexts[2].
Here we focus our attention on four different zeroth order black-box attacks:

1. Zeroth Order Stochastic Gradient Descent [2]
2. Frank Wolfe Black-Box Attack [3]
3. Zero Order Stochastic Conditional Gradient and its modified version [4]

We explore the performances of the algorithms analyzing the rate of success and the computational requirements in different scenarios. Three different networks have been employed as targets: a simple *Ad Hoc* Convolutional DNN, *VGG16* [5] and Google's *InceptionV3* [6].

The rest of the report is organized as follows: in Section 2 we introduce the main differences between a White-Box and a Black-Box attack, highlighting the challenges of the latter; in Section 3 we explain how an adversarial attack can be formalized as an optimization problem; in Section 4 we introduce the zeroth-order methods as presented in [4] with a particular focus on the gradient smoothing approach; in Section 5 we describe the algorithms implemented; in Section 6 we start focusing on the implementation details describing the datasets and the model used; finally in Section 7 and 8 we present and discussed the obtained results.

2 White-Box and Black-Box Attacks

When attacking DNNs, one can distinguish between two frameworks: white-box and black-box.

The former assumes model transparency, that allows full control and access to a targeted DNN for sensitivity analysis. Under this framework the ability of performing back propagation is granted, therefore it is possible to compute the gradient and use many standard algorithms (e.g. gradient descent) to perform the attack. More specifically, when dealing with an image classification problem, back propagation allows us to assess the effect of changing pixel values on the confidence scores for the prediction. Under this framework, we recall the popular Carlini & Wagner Attack[7], which exploits the internal configuration of

the targeted DNN, formulating adversarial attacks as an optimization problem and using the L_2 norm to quantify the difference between the adversarial and the original examples. The attack could essentially be seen as a gradient descent based attack, driven by the representation of the logit layer of the targeted DNN and the L_2 distortion. They also showed that their attack can successfully bypass different detection methods used against adversarial examples[8].

However, most real world DNNs do not allow access to the internal configuration. In the following paper we will therefore consider black-box attacks, assuming only the input and the output (confidence score for each class) of a network are accessible. We can further distinguish between two types of black-box attacks:

1. **Untargeted:** craft an adversarial example which leads to misclassification;
2. **Targeted:** craft an adversarial example such that it is classified as the desired class.

Note that the definition of black-box attack to DNNs may vary. In the most challenging case, the targeted DNN is completely unknown, only images and class labels are given and the classifier cannot be queried. This very restrictive setting is sometimes referred as no-box. In this case, research focuses mainly on the attack transferability from a self-trained DNN.

In general, however, the attacker has the possibility to query the network and can exploit this to craft adversarial examples. Being this a black-box framework, one cannot use back propagation but the query process can be iterated until the attacker finds an adversarial example. This more feasible setting is called practical black-box attack. From now on, we will always consider this framework.

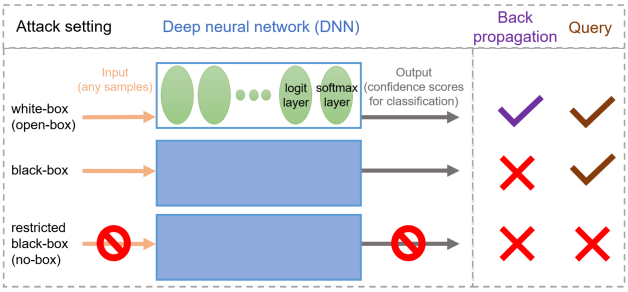

Figure 1: White-box, practical black-box and restricted black-box attacks Frameworks.

Under this black-box setting many existing attacking approaches train a substitute model, considered representative of the targeted network. This model, which is a white-box model, can then be attacked exploiting back propagation and the generated adversarial images are used to attack the originally targeted DNN. This of course assumes that the substitute model is representative of the targeted DNN in terms of its classification rules and heavily relies on the transferability of the attack to the target DNN.

Instead of the previously discussed methods, in this paper we will focus on zeroth order optimization techniques, which do not require the training of a substitute model. These techniques allow to consider the attack as if it was a white-box attack and to perform pseudo back propagation steps, computing gradient estimates through a large number of queries. Analogously, one can view an attacker as an optimizer and an adversarial attack as an objective function to be optimized.

The main advantage of this approach is the fact that it does not rely on substitute models and, hence, there is no loss due to transferability. However, it must be noticed that it brings with it some issues, such as:

1. High number of gradient iterations required to optimize the distortion of the adversarial examples;
2. High number of gradient estimation iterations required with respect to a d-dimensional problem, with d and high dimensions number;
3. Generation of adversarial examples near or upon the boundaries of the perturbation set;

3 Adversarial Attack as an Optimization Problem

In this section we introduce the specific framework in which we are deploying and testing the proposed algorithms for zeroth-order black-box attack. The attacks described in this report are based on modified images, which represents the most common application for this kind of studies in addition of being the most intuitive scenario to evaluate the similarity between the original and the attacked input, even though this methods are not limited to this scenario [9].

Images could belong to black and white or RGB space (3 channels) and present various resolution. In any case each channel for each pixel has an activation value between 0 and 1, therefore the image space can always be represented by $\chi = [0; 1]^n$ for a suitable n. Even a low resolution RGB image easily leads to a solution space whose dimension has an order of magnitude of hundreds of thousands.

A properly trained neural network for image classification can be formally seen as a model $\mathcal{M}$ that, for each output neuron, labeled with an integer between 1 and m, and corresponding to one of the learned concepts, returns the estimated probability that the input sample belongs to that concepts. This kind of model can be represented by a decision function $F_{\mathcal{M}}(x) : \chi \to [0; 1]^m$. The decision function $F_{\mathcal{M}}$, in general, is non convex and can present a very irregular structure.

The original problem is presented by Carlini and Wagner[7] in 2017 and is:

$$\begin{aligned} \min \quad & ||\delta|| \\ \text{sub.to} \quad & C(x+\delta) = t \\ & (x+\delta) \in [0,1]^n \end{aligned}$$

where $C(x) = \arg\max_{i \in \{1,\dots,m\}} F_{\mathcal{M}}(x)$.
Since the above formulation is difficult for existing algorithms to solve directly, as the constraint $C(x+\delta) = t$ is highly non-linear, they expressed it in a different form that is better suited for optimization.

In particular we can introduce a loss function $f : \chi \to \mathbb{R}$ which depends on the types of attack that have been presented in Section 2:

$$\begin{cases} f_t(x) = \max\{\max_{i \neq t} \log[F_{\mathcal{M}}(x)]_i \, - \, \log[F_M(x)]_t \, , \, -k\} & \text{if the attack is targeted} \\ f(x) = \max\{\log[F_{\mathcal{M}}(x)]_{t_0} \, - \, \max_{t \neq t_0} \log[F_M(x)]_i \, , \, -k\} & \text{if the attack is untargeted} \end{cases} \tag{1}$$

where t_0 is the concept the input x belongs to (i.e. the predicted class) and t is the target concept (i.e. the class whose probability we want to maximize) and k is the transferability of the model, which in our case, as explained in Section 2, can be set to zero.

The *untargeted attack* against model $\mathcal{M}$ over a sample x searches for a vector δ such that $x + \delta \in \chi$ and $\arg\max_{i \in \{1\dots m\}} F_{\mathcal{M}}(x+\delta) \neq \arg\max_{i \in \{1\dots m\}} F_{\mathcal{M}}(x)$ and the *targeted attack* against model $\mathcal{M}$ over a sample x in favour of a concept $t \in \{1, \dots, m\}$ searches for a vector δ such that $x + \delta \in \chi$ and $\arg\max_{i \in \{1\dots m\}} F_{\mathcal{M}}(x+\delta) = t$.

These loss functions only take into account the probability of misclassification. Since the attack to be effective also needs to be as close as possible to the original input, we need to introduce also a factor that takes into consideration the L_p norm of we need to introduce also a factor that takes into consideration δ.

This can be done in two ways:

1. we can add a regularization term to the loss function that takes into account the norm of δ as in [2], thus trying to solve:

$$\begin{aligned} \min \quad & \|\delta\| + c \cdot f(x+\delta) \\ \text{sub.to} \quad & (x+\delta) \in [0,1]^n \end{aligned}$$

2. we can modify the feasible region $\chi = [0;\ 1]^n$ by doing an intersection with the L_p Ball of radius set equal to a parameter ϵ and centered in the original input $\mathcal{B}_\epsilon^{(p)}(x)$ as in [4], thus trying to solve:

$$\begin{aligned} \min \quad & f(x+\delta) \\ \text{sub.to} \quad & (x+\delta) \in [0,1]^n \cap \mathcal{B}_\epsilon^{(p)}(x) \end{aligned}$$

The actual details as the choice of p depend on the implementation and will be discussed in Section 5.

Minimizing the objective function f within χ leads to a solution of the problem i.e. a successful attack. Our problem however presents a relevant difference from a generic optimization problem in which the final goal is to reach a good approximation of the minimum of the objective function. In this case, instead, we are only seeking for an effective adversarial example. Therefore the various algorithms that we explore follow the approximated gradient

based on zeroth order information until the attack succeeds. To define a general stopping criterion we introduce the *set of accepted solutions* S which is equal to $\{1, \ldots, m\} \setminus \{t_0\}$ in the untargeted attack case and $\{t\}$ in the targeted one. Under this notation, the details of the stopping criterion procedure are described in Algorithm 1.

Algorithm 1: Stopping criterion (SC)

Input: $x_k \in \chi$, $F_{\mathcal{M}}(x)$ be the model decision function, S the set of accepted solutions.

Let $q = 1$ be the signal used to stop our optimization process;
Set $q = 0$;
if $\arg\max_{i \in \{1,\ldots,m\}} F_{\mathcal{M}}(x_k, w) \in S$ **then**
  $q = 1$;
else
  $q = 0$;
end
Return q

4 Gradient Smoothing Approach and Zeroth Order Methods

Before proceeding with the actual implementation details we present some theoretical background in order to justify the algorithms presented in Sections 5.3 and 5.4.

In [4] are presented various algorithms to solve the following stochastic optimization problem by having access only to a zeroth-order oracle

$$\min_{x \in \chi} \{f(x) = \mathbb{E}_{\xi}[F(x, \xi)]\} \tag{2}$$

where $f : \mathbb{R}^d \to \mathbb{R}$ is the objective function and χ is a closed convex subset of $\mathbb{R}^d$.

4.1 Assumptions

Here we give a list of assumptions that are needed in order to proof some results of Balasubramanian and Ghadimi [4] that will be explained in this section.

Assumption 1. Let $\|\cdot\|$ be a norm on $\mathbb{R}^d$. For any $x, y \in \mathbb{R}^d$, the zero order oracle outputs an estimator $F(x, \xi)$ of f(x) such that:

$$\begin{aligned} \mathbb{E}_{\xi}[F(x, \xi)] &= f(x) \\ \mathbb{E}_{\xi}[\nabla_x F(x, \xi)] &= \nabla f(x) \\ \mathbb{E}_{\xi}[\|\nabla_x F(x, \xi) - \nabla f(x)\|^2] &\leq \sigma^2 \end{aligned}$$

Assumption 2. Function F has Lipschitz continuous gradient with constant L, almost surely for any ξ so that $|F(y, \xi) - F(x, \xi) - \langle \nabla F(x, \xi), x - y \rangle| \leq \frac{L}{2}\|x - y\|^2$.

Assumption 3. The feasible set χ is bounded such that $\max_{x,y\in\chi} \|x-y\| \leq D_\chi$ for some $D_\chi \geq 0$.

As consequence of previous assumptions, it is *a piece of cake* to see that, for all $x \in \chi$, there exist a constant $B > 0$ such that $\|\nabla f(x)\| \leq B$.

4.2 Main results

In Balasubramanian and Ghadimi new zero order optimization algorithms are presented by using the idea that a computation of the approximated gradient of a function $f(x)$ can be achieved by using the gradient of the smoothed function

$$f_v(x) = \mathbb{E}[f(x+vu)] \tag{3}$$

for some $v \in (0,\infty)$ and $u \sim \mathcal{N}(0, I_d)$ be a standard Gaussian random vector.

This is possible due to the *Stein's identity* that show that if $u \sim \mathcal{N}(0, I_d)$ and $g : \mathbb{R}^d \to \mathbb{R}$ is an almost-differentiable function with $\mathbb{E}[\|\nabla g(u)\| \leq \infty$, then:

$$\mathbb{E}[ug(u)] = \mathbb{E}[\nabla g(u)] \tag{4}$$

So, if we set $g(u) = f(x+vu)$ and by using the Stein's equation we have that:

$$\nabla f_v(x) = \mathbb{E}_u[\nabla f(x+vu)] = \mathbb{E}_u[f(x+vu)u]$$

Furthermore in [10] it has been proofed that:

$$\mathbb{E}_u\left[f(x+vu)u\right] = \mathbb{E}_u\left[\frac{f(x+vu)}{v}u\right] = \mathbb{E}_u\left[\frac{f(x+vu)-f(x)}{v}u\right]$$

ending with:

$$\nabla f_v(x) = \mathbb{E}_u\left[\frac{f(x+vu)-f(x)}{v}u\right] \tag{5}$$

This implies that we can estimate the gradient of f_v by using only evaluation on f, giving as the possibility to define the stochastic gradient of $f_v(x)$ as:

$$G_v(x,\xi,u) = \frac{f(x+vu,\xi)-f(x,\xi)}{v}u \tag{6}$$

which, under *Assumption 1*, is an unbiased estimator of $\nabla f_v(x)$ since:

$$\mathbb{E}_{u,\xi}[G_v(x,\xi,u)] = \mathbb{E}_u\big[\frac{f(x+vu)-f(x)}{v}u\big] = \nabla f_v(x)$$

Moreover it can be shown [10] that for any $x \in \mathbb{R}^d$

$$\|\nabla f_v(x) - \nabla f(x)\| \leq \frac{v}{2}L(d+3)^{\frac{3}{2}}\,. \tag{7}$$

Given these results we can compute the estimated gradient of a loss function $f(x)$ parametrized over a model output $F_{\mathcal{M}}(x)$ using only zero order information.

5 Optimization Methods

5.1 Zero Order Optimizer with ADAM/Newton (ZOO)

The algorithm is based on the Carlini & Wagner white-box attack, which tries to find the adversarial example x of the original image x_0 by solving the following problem:

$$\min_{x \in [0,1]^p} \|x - x_0\|_2^2 + c \cdot f(x,t), \tag{8}$$

as presented in 4, where f is defined in 1.

An approximated gradient is computed as:

$$\hat{g}_i := \frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x - he_i)}{2h}\,, \tag{9}$$

where h is a small constant and e_i is a standard basis vector with only the i-th component set to 1.

For any $x \in \mathbb{R}^p$, we need to evaluate the objective function $2p$ times to estimate gradients of all coordinates, which makes the approach too expensive in practice. With just one more function evaluation we can obtain an estimate of the Hessian as:

$$\hat{h}_i := \frac{\partial^2(x)}{\partial x_{ii}^2} \approx \frac{f(x + he_i) - 2f(x) + f(x - he_i)}{h^2}\,. \tag{10}$$

To overcome the computation cost stochastic coordinate descent is used, specifically it is shown that ADAM outperforms vanilla gradient descent and that in practice converges faster than Newton's method.

The algorithms are here reported:

Algorithm 2: ADAM

Input: η, β_1, β_2, ϵ
Initialize: $M = \mathbf{0}$, $T = \mathbf{0}$, $v = \mathbf{0}$, $M, T, v \in \mathbb{R}^p$
while *not converged* **do**
  Randomly pick B coordinates $i \in \{1, ..., p\}$;
  Estimate $\hat{g_B}$;
  $T_B \leftarrow T_B + 1$;
  $M_B \leftarrow \beta_1 M_B + (1 - \beta_1)\hat{g_B}$;
  $v_B \leftarrow \beta_2 v_B + (1 - \beta_2)\hat{g_B}^2$;
  $\hat{M_B} = \frac{M_B}{(1-\beta_1^{T_B})}$; $\hat{v_B} = \frac{v_B}{(1-\beta_2^{T_B})}$;
  $\delta^* = -\eta \frac{\hat{M_B}}{\sqrt{\hat{v_B}+\epsilon}}$;
  $x_B \leftarrow x_B + \delta^*$;
end

For network with large input size p zero order methods can be quite slow. To avoid this problem we introduce a dimension reduction transformation $D(y)$, $D : \mathbb{R}^m \to \mathbb{R}^p$ with $m < p$ and replace it to $x - x_0$ in our minimization problem. In this way the dimension of an input

Algorithm 3: Newton

Input: η
while *not converged* **do**
  Randomly pick B coordinates $i \in \{1, ..., p\}$; Estimate $\hat{g_B}$ and $\hat{h_B}$; **if** $\hat{h_B} < 0$ **then**
    $\delta^* \leftarrow -\eta \hat{g_B}$;
  else
    $\delta^* \leftarrow -\eta \frac{\hat{g_B}}{\hat{h_B}}$;
  end
  $x_B \leftarrow x_B + \delta^*$;
end

remains unaltered but the dimension of the adversarial noise is reduced. The problem is therefore reformulated in this way:

$$\min_{x_0 + D(y) \in [0,1]^p} \|D(y)\|_2^2 + c \cdot f(x_0 + D(y), t). \tag{11}$$

Moreover, for large images and difficult attacks a *hierarchical attack* scheme is proposed: a series of transformations $D1$, $D2$, ... with dimensions $m1$, $m2$, ... is used to gradually increase m during the optimization process.

Another proposed technique is *importance sampling*, which consists in dividing the image in 8×8 regions and assign sampling probabilities according to the pixels values change in that region.

The last techniques is *ADAM state reset* which is used to efficiently decrease the L_2-distance after a valid attack has been found. The reason behind this idea is that after a valid adversarial image has been found and the hinge-loss function reaches 0, the optimizer still tries to reduce the probability of the original image instead to decrease the L_2-distance.

5.2 Black-Box Frank-Wolfe

This algorithm, introduced in [3], exploits a Frank-Wolfe variant in order to perform efficient and effective optimization-based adversarial attacks, with guaranteed convergence rate. While there is a White-Box version of the algorithm either, which is in fact very similar to the Black-Box one, here we discuss the latter setting only. Black-Box Frank-Wolfe algorithm, being based on Frank-Wolfe algorithm, is a projection-free algorithm, hence reducing the number of adversarial examples near the perturbation set boundaries. Proposed attack algorithm involves a momentum term and guarantees a $O\left(\frac{1}{\sqrt{T}}\right)$ convergence rate in the given non-convex setting. Moreover, it can be shown that the query complexity of the Frank-Wolfe attack algorithm is linear in data dimension d. Both the provided convergence rate and query complexity hold under the following assumptions.

Assumption 4. Gradient of f in zero $\nabla f(\mathbf{0})$ is bounded, i.e. it satisfies:

$$\max_y \|\nabla f(\mathbf{0})\|_2 \leq G \tag{12}$$

Theorem 1. Under hypothesis provided by Assumptions 2, 3 and 4, being $\gamma = \sqrt{\frac{f(x_0) - f(x^*)}{C_\beta L D_\chi^2 T}}$, $b = Td$ and $\delta = \sqrt{\frac{1}{Td^2}}$, then the output of Algorithm 4 satisfies:

$$\mathbb{E}[\tilde{g_T}] \leq \frac{D_\chi}{\sqrt{T}} \left(\sqrt{2C_\beta L(f(x_0) - f(x^*))} + C_\beta(L + G + LD_\chi) \right) \tag{13}$$

with $\tilde{g_T} = \min 1 \leq k \leq T g(x_k)$, expectation of $\tilde{g_T}$ taken over the randomness of the gradient estimation, x^* optimal gradient estimation and $C_\beta = (1+\beta)/(1-\beta)$.

From Theorem 1, we get convergence rate of $O(1/\sqrt{}(T))$. Moreover, when the total number of queries required in Algorithm 4 is $Tb = T^2 d$, with T epochs and b number of gradient estimates, it is linear in data dimension d.

If we use as objective function the function f(x) defined in Section 3, we can formulate our optimization problem as:

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & \|x - x_0\|_p \leq \epsilon \end{aligned} \tag{14}$$

where x_0 is the original linearized image, p is an integer and ϵ is a small constant. Moreover, it must be noted that the constraint set is a bounded convex set when $p \geq 1$. Being this algorithm a variant of Frank-Wolfe algorithm, it gets rid of the projection, hence solving the problem of having adversarial examples near the perturbation set boundaries, by querying, at each iteration t, a Linear Minimization Oracle (LMO) over the constraint set χ defined as the L_p Ball centered in x_{ori} and radius ϵ , s.t.

$$\text{LMO} \in \arg\min_{x \in \chi}, \nabla f(x_t)\rangle \tag{15}$$

This yelds this method to be conservative with respect to others, by keeping the iterates within the constraint set χ.

While general form of the LMO can be expensive to obtain, for the previously defined constraint set χ there exists a closed-form solution, which for L_∞ case is

$$v_t = \arg\min_{x \in \chi} \langle x, m_t \rangle = -\epsilon \cdot \text{sign}(m_t), \tag{16}$$

while, for L_2 case is:

$$v_t = \arg\min_{x \in \chi} \langle x, m_t \rangle = -\epsilon \cdot \frac{m_t}{||m_t||}, \tag{17}$$

The Black-Box Frank-Wolfe algorithm works by iterating an user-defined number of epochs and for each of these a new momentum term m_t is computed by making a weighted average between previous term m_{t-1} and the estimated gradient q_t, which is then used in descent direction definition by means of the Linear Minimization Oracle (LMO). After the new descent direction has been defined, new input image x_{t+1} is computed.

Algorithm 4: Black-Box Frank-Wolfe algorithm

Input: x_0, T number of iterations, $\{\gamma_t\}$ list of step sizes, b number of gradient estimation iterations, δ gradient smoothing parameter

Initialize: $m_t = \text{GRAD\_EST}(x_0, b, \delta)$

for $t = 0, ..., T-1$ **do**

$q_t = \text{GRAD\_EST}(x_t, b, \delta)$

$m_t = \beta \cdot m_{t-1} + (1-\beta) \cdot q_t$

$v_t = \arg\min_{x \in \chi} \langle x, m_t \rangle$ // LMO

$d_t = v_t - x_t$

$x_{t+1} = x_t + \gamma_t d_t$

end

At implementation level, a stopping criterion check has been added to the algorithm, such that it stops before reaching the limit number of T iterations if the given target label is maximized or minimized.

Gradient estimation algorithm uses symmetric finite differences in order to estimate the gradient, by sampling sensing vectors according to two different options:

1. Sampling vectors uniformly form Euclidean unit sphere surface
2. Sampling vectors uniformly from standard multivariate Gaussian distribution

Algorithm 5: Gradient estimation algorithm

Input: x input variable (e.g. linearized image), number of gradient estimation iterations b, δ gradient smoothing parameters

Initialize: $q = 0$

for $i = 1, ..., b$ **do**

Option 1: sample u_i uniformly from Euclidean unit sphere with $\|u_i\|_2 = 1$

$$q = q + \frac{d}{2\delta b}(f(x + \delta u_i) - f(x - \delta u_i))u_i$$

Option 2: sample u_i uniformly from standard multivariate Gaussian distribution $N(0, I)$

$$q = q + \frac{1}{2\delta b}(f(x + \delta u_i) - f(x - \delta u_i))u_i$$

end

Return: q

To speed up gradient estimation algorithm and take advantage of matrix operations through GPU processor, iterative behaviour has been partially substituted by a batch approach: by defining a batch size k, the number of iterations are lowered from b to b/k. At each of these fewer iterations, a sample of k examples is taken using one of the two options given, then only q is computed iteratively over the batch, by going through every example in it.

5.3 Zero order Stochastic Conditional Gradient

Zero order Stochastic Conditional Gradient (ZSCG) [4] is an algorithm based on the ideas explained in Section 4. We can compute an approximation of the gradient $G_v(x, \xi, u)$ based on zeroth-order information by using the Gaussian Stein's identity introduced in Equation 4. Once $G_v(x, \xi, u)$ has been computed, first order Taylor expansion of the target function can be easily minimized.

In the original ZSCG algorithm, as defined by Balasubramanian and Ghadimi [4], there is no specific stopping criterion and one iteration is sampled from a discrete probability distribution P_R over the iterations set $\{1, \ldots, N\}$ to provide the output. If the number of iteration goes to infinity, under the assumptions and the notations presented in Section 4 and by properly setting the parameters of the algorithm as explained in Equation 2.7 of [4], it can be shown that:

$$\mathbb{E}[g_\chi^R] \leq \frac{f(x_0) - f^* + LD_\chi^2 + 2\sqrt{B^2 + \sigma^2}}{\sqrt{N}} \tag{18}$$

where $g_\chi(x)$ is the Frank-Wolfe Gap used to provide the convergence analysis and is defined as $\langle \nabla f(x_{k-1}), x_{k-1} - \hat{z_k} \rangle$ and $\hat{z}_k = \arg\min_{u \in \chi} \langle \nabla f(x_{k-1}), u \rangle$. Hence the total number of calls to the zeroth order stochastic oracle and linear subproblems required to be solved to find an ϵ-stationary point of the problem are, respectively, bounded by:

$$O\left(\frac{d}{\epsilon^4}\right), O\left(\frac{1}{\epsilon^2}\right). \tag{19}$$

However the algorithm we present in Algorithm 6 has been modified in order to make it suitable to our problem. In particular we do not use the Frank-Wolfe gap defined above but instead the stopping criterion defined in Algorithm 1.

The Algorithm 6 describes in detail the general implementation of the Zero order Stochastic Conditional Gradient Method with stopping criterion.

5.4 Inexact Zero order Stochastic Conditional Gradient

The Inexact Zero order Stochastic Conditional Gradient (Inexact ZSCG) for non convex function comes from the idea that allows us to skip the gradient evaluation from time to time, introducing a subroutine called *Inexact Conditional Gradient method (ICG)* which try to solve the following quadratic equation

$$P_\chi(x, g, \gamma) = \arg\min_{u \in \chi} \left\{ \langle g, u \rangle + \frac{\gamma}{2} \|u - x\|^2 \right\} \tag{20}$$

which is the standard subproblem of stochastic first-order methods applied to a minimization problem when g is an unbiased stochastic gradient of the objective function at x.

A detailed description of ICG is presented in Algorithm 7. We remark that this algorithm is a particular and simplified version of Algorithm 6 where we do not have a generic function f to minimize but its target function is $f'(u) = \langle g, u \rangle + \frac{\gamma}{2}\|u - x\|^2$ as stated in Equation 20. Therefore the gradient of the objective function is $\nabla f'(u) = g + \gamma(u - x)$ and there is no

Algorithm 6: Zero order Stochastic Conditional Gradient Method for adversarial attacks

Input: $x_0 \in \chi$, smoothing parameter $v > 0$, non-negative sequence α_k, m_k a positive integer sequence, $N \geq 1$ be the iteration limit, S be the set of accepted solutions, $F_M(\cdot, w)$ be the output function of out target model M and $f(\cdot)$ be our target function

Set $q = 0$, $k = 0$

while $q = 0$ *and* $k \leq N$ **do**

1. Increment k
2. Generate $u_k = [u_{k,1}, ..., u_{k,m_k}]$, where $u_k \sim \mathcal{N}(0, I_d)$; call the stochastic oracle m_k time to generate:

$$G_v^k \equiv G_v(x_{k-1}, \xi_k, u_k) = \frac{1}{m_k} \sum_{j=1}^{m_k} \frac{f(x_{k-1} + vu_{k,j}, \xi_{k,j}) - f(x_{k-1}, \xi_{k,j})}{v} u_{k,j}$$

3. Solve the linear programming problem:

$$z_k = \arg\min_{u \in \chi} \langle G_v^k, u \rangle$$

4. Set

$$x_k = (1 - \alpha_k) x_{k-1} + \alpha_k z_k$$

5. Check stopping criterion:

$$q = SC\left(x_k, F_M(\cdot, w)\right)$$

end

Return x_k

Algorithm 7: Inexact Conditional Gradient method (ICG)

Input: x, g, γ, μ

Set $\hat{y}_0 = x, t = 1, k = 0$

while $k = 0$ **do**

$$y_t = \arg\min_{u \in \chi} \{h_\gamma(u) = \langle g + \gamma(\hat{y}_{t-1} - x), u - \hat{y}_{t-1} \rangle\}$$

if $h_\gamma(y_t) \geq -\mu$ **then**
  $k = 1$
else
  $\hat{y}_t = \frac{t-1}{t+1} \hat{y}_{t-1} + \frac{2}{t+1} y_t$ and $t = t + 1$
end

end

need for the approximate gradient estimate in Step 2 of Algorithm 6, in addition here we set $\alpha_k = \frac{2}{k+1}$.

By using the ICG, we can modify the ZSCG and achieve better convergence results. In particular, we define our convergence criterion for non convex problem by introducing the *gradient mapping* function

$$GP_\chi = \gamma\big(x - P_\chi(x, g, \gamma)\big) \tag{21}$$

where $P_\chi(x, g, \gamma)$, being the output of ICG algorithm, represents our updated x, such that the norm of GP_χ can be used to provide convergence results. Under this new convergence criterion, it can be shown that given the Assumptions 1, 2 and 3, presented in Section 4 we have:

$$\mathbb{E}\left[\|GP_\chi(x_R, \nabla f(x_R), \gamma_R)\|^2\right] \leq \frac{8L}{N}\big(f(x_0) - f^* + L + B + \sigma^2\big). \tag{22}$$

Algorithm 8: Zero order Stochastic Conditional Gradient Method with Inexact Updates for adversarial attacks

Input: $x_0 \in \chi$, smoothing parameter $v > 0$, m_k, γ_k, μ_k a positive integer sequences, $N \geq 1$ be the iteration limit, S be the set of accepted solutions, $F_M(\cdot, w)$ be the output function of out target model M and $f(\cdot)$ be our target function

Set $q = 0$, $k = 0$

while $q = 0$ *and* $k \leq N$ **do**

1. Increment k
2. Generate $u_k = [u_{k,1}, ..., u_{k,m_k}]$, where $u_k \sim \mathcal{N}(0, I_d)$; call the stochastic oracle m_k time to generate:

$$G_v^k \equiv G_v(x_{k-1}, \xi_k, u_k) = \frac{1}{m_k}\sum_{j=1}^{m_k} \frac{f(x_{k-1} + vu_{k,j}, \xi_{k,j}) - f(x_{k-1}, \xi_{k,j})}{v} u_{k,j}$$

3. Compute new value using ICG:

$$x_k = ICG(x_{k-1}, G_v^k, \gamma_k, \mu_k)$$

4. Check stopping criterion:

$$q = SC\Big(x_k, F_M(\cdot, w)\Big)$$

end

Return x_k

Hence, when using this particular Frank-Wolfe gap GP_χ, the total number of calls to the stochastic oracle and linear subproblems solved to find and ϵ-stationary point of problem (1.1) are, respectively, bounded by:

$$O\left(\frac{1}{\epsilon^2}\right),\ O\left(\frac{1}{\epsilon^2}\right). \tag{23}$$

It is important to remember that this results are obtained with a convergence criterion that is much different both conceptually and practically from the stopping criterion we used.

6 Experimental Setup

6.1 Setup

To evaluate and compare the previously described methods we consider attack time, success rate and distance from the original images. The parameters used are the ones indicated as suggested in 8.
For all our experiments we used Google Colaboratory, running on a Intel Xeon E5-2690v4 CPU with a NVIDIA Tesla K80 GPU.
The framework chosen for dealing with the datasets, the models and the optimization algorithms has been PyTorch, using CUDA to run all the relevant computation in GPU.
It is important to notice that, since Google Colaboratory needs to maintain the flexibility to adjust usage limits and hardware availability on the fly, resources available may vary over time, giving therefore origin to inconsistencies in terms of computational time.

6.2 Dataset

6.2.1 MNIST

The MNIST[11] dataset is one of the most famous dataset used to test algorithms in the image classification fields and more generally in Computer Vision. It consists of 60.000 images of handwritten digits (0-9), the images are in black and white (one channel) and have a dimension of 28x28, for a total of 784 pixels. Each pixel is in the interval [0, 1].

6.2.2 CIFAR10

Cifar10[12] is another one famous dataset used in the field of computer Vision. It consists of 60.000 images of 10 real objects (plane, car, bird, cat, deer, dog, frog, horse, ship, truck). The images have 3 channels and have a dimension of 32x32, such that each image can be represented as a vector x s.t. $x \in [0,1]^{3072}$.

6.3 Models

6.3.1 Ad Hoc - MNIST

We created an ad-hoc network trained using the MNIST datasets. The network consisted of a feature extraction block, with four Convolutional layers (kernel = 2x2) with ReLU activation function, two Pooling layers (kernel = 2x2), and a classifier block, with three feed forward layers, with ReLU activation function and Dropout as regularization.

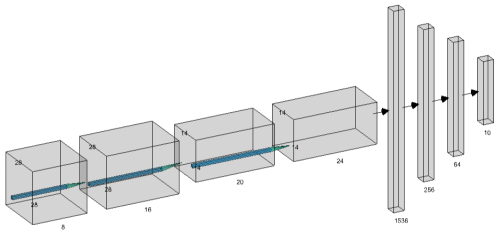

Figure 2: Architecture of the ad hoc model built for the MNIST dataset

6.3.2 VGG16 - Cifar10

Since the multi classification task with the CIFAR10 dataset is much more harder than the one using MNIST with performed transfer learning using VGG16 with Batch Normalization[5]. VGG16 with Batch Normalization is a deep network with a Feature extractor block with 16 convolutional layers, with dropout, batch normalization, ReLU as activation function and 5 Max pooling layers; an Adaptive Pooling block and a Classifier Block with three linear layers with Dropout and ReLU.
VGG16 was trained using ImageNet (3x299x299), but thanks to its Adaptive Pooling layer it acccepts whatever 3D input. The transfer learning was done by replacing the Classifier block with a new one with 5 layers of dimensions 2048, 1024, 1024, 128 and 10, ReLU activation functions and Dropout with probability 0.5.
VGG16 showed a discrete level of generalization, achieving a 0.88 accuracy score on the validation set.

6.3.3 Inceptionv3 - Cifar10

The third and last model used is InceptionV3[6], a very deep network created by google with the introduction of the *inception* module. In contrast to VGG16, Inception doesn't have an Adaptive Pooling Layer and requires an input of 3x299x299, for a total 268203 dimensions, about 88 times more dimensions than the one used for VGG.
To perform transfer learning of InceptionV3 on the Cifar10 dataset, other than replacing the last linear layers with some custom one, as it happened on VGG, we had to upscale the dimension of Cifar10 from 32x32 to 299x299.

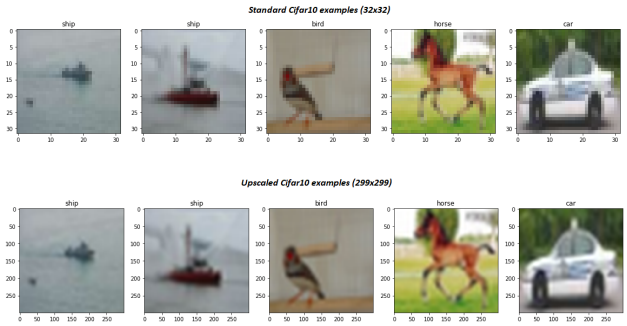

Figure 3: Standard Cifar10 images (top) and the upscaled ones for InceptionV3 (bottom)

Like in the case of VGG16, InceptionV3 achieved an accuracy score of 0.88 on the validation test, showing good generalization abilities.

6.4 Code Implementation

Three main generic objects have been involved in the training and testing pipeline of this project: a *Model*, a *Loss function* and an *Optimizer* .

6.4.1 Model

Model implements three different classes, one for each of the previously described models, namely MnistNet, MyVGG and InceptionV3. MnistNet and MyVGG extend nn.module while InceptionV3 is a torch Inception wrapper.

6.4.2 Loss Function

Loss is an abstract object (not a function itself): the initialization takes three arguments: the target class and two booleans, one indicating if we are trying to maximise or minimise its probability and one indicating if the input is a probability distribution. The two losses described in 3 are implemented in the classes MSELoss and ZOOLoss. In addition to the already presented arguments, ZOOLoss takes as input also a transferability parameter, useful if one is trying to perform a transfer attacks.
Both the Losses implement a forward method, taking as argument the output of the DNN being attacked and ensuring that it is soft-maxed (i.e. it is a probability distribution) before computing and returning the Loss as defined in section 3.

6.4.3 Optimizer

Six different optimizers are implemented:

- ZOOptim (ZOO)
- FWOptim (Black-Box Frank-Wolfe)

- ZeroSGD (Zero order Stochastic Gradient Descent)
- ClassicZSCG (Zero order Stochastic Conditional Gradient)
- InexactZSCG (Zero order Stochastic Conditional Gradient with Inexact Updates)
- InexactAcceleratedZSCG (Accelerated Zero order Stochastic Conditional Gradient with Inexact Updates)

Optimizers are objects parametrized by the *model* we are trying to attack, a *loss object*, to compute the loss function, approximate the gradient and find a descent direction, and a *torch device*, initialize by default to 'cuda'.
All optimizers exposes four main methods: *run*, *step*, *compute_ gradient* and *projected_ boundaries*. Each of these methods takes different arguments according to the optimizer object it belongs but they generally work in a similar way.

Run method always takes as input the target image as a 3D tensor, a batch_size (maximum parallelization in gradient computation), n_gradient (number of normal vector to generate for each step), max_step (maximum number of iterations), verbose (wether to display information or not), additional_out and tqdm_disabled. The method computes and returns, according to the algorithms presented section 5, the adversarial image by minimizing the loss function in at most max_step. Additionally it returns the lists of losses and adversarial images at each step.
A complete list of arguments can be find in the Appendix.

Step method takes as input the current adversarial image and most of run method's parameters and, after calling *compute_ gradient*, computes and return the new adversarial image.

Compute_gradient method uses the *model* and the *loss function* to compute and return an approximation of the gradient.

Project_boundaries method ensures that the box constraint $x \in [0, 1]^p$ is satisfied.

Other frequent methods are *get_ parallel* and *compute_ CG*. The former is used to prepare the parallelization for the gradient estimation, in order to increase code's performances, the latter computes the conditional gradient (used in *ClassicZSCG* and *InexactZSCG*).

6.4.4 Evaluation

Evaluation allows to evaluate the performance of all the optimizers on the different models. It requires as input an optimizer and a loss, together with their arguments. Additionally it requires the desired dataset and the number of examples we want to use to perform the evaluation.
It returns as output a json file containing the arguments it received as input, the success rate, the mean time required by an attack and the mean distance between a generated adversarial image and its original one.

7 Results

To compare the different algorithms more parameters have been taken in consideration: in the case of L_∞ the parameters taken in consideration have been *Success Rate* and *Average Time per attack*, while keeping fix the maximum value of the norm ϵ; otherwise, in the case of L_2 norm, the parameters taken in in consideration have been *Success Rate*, *Average Time per attack* and *Average* L_2 *distance*.
Moreover, since the limited computation power, algorithms have been evaluated against 100 examples in case of *untargeted attack* and 20 examples (180 attacks) in the case of *targeted attack*.

7.1 Ad hoc - MNIST

In this section, the results of four types of attack against the *Ad-Hoc model* trained on MNIST data will be described. The four types of attack against MNIST are:

1. Untargeted attack using the feasible set $\chi = \{x \in [0,1]^{784} : \|x_0 - x\|_\infty \leq \epsilon = 0.2\}$
2. Untargeted attack using the feasible set $\chi = \{x \in [0,1]^{784} : \|x_0 - x\|_2 \leq \epsilon = 2\}$
3. Targeted attack using the feasible set $\chi = \{x \in [0,1]^{784} : \|x_0 - x\|_\infty \leq \epsilon = 0.2\}$
4. Targeted attack using the feasible set $\chi = \{x \in [0,1]^{784} : \|x_0 - x\|_2 \leq \epsilon = 4\}$

Where x_0 is the original input image used in the attack.

7.1.1 Untargeted with L_∞

Results for untargeted attack against MNIST with a feasible set given by $\|x_0 - x\|_\infty \leq \epsilon = 0.2$, showed minor differences between the algorithms in terms of *Success Rate*, since the *Success Rate* achieved has been in range of [0.97; 0.98], but showed big differences in terms of time taken, with *Inexact ZSCG* being the fastest and *Frank-Wolfe* the slowest. Results can be seen in Table 1 and Fig. 4.

Optimizer	ϵ	Success rate	Average time (s)
inexact	0.2	0.97	0.154
classic	0.2	0.97	0.860
frank-wolfe	0.2	0.98	0.957

Table 1: **Result of untargeted attack with L_∞ against MNIST**

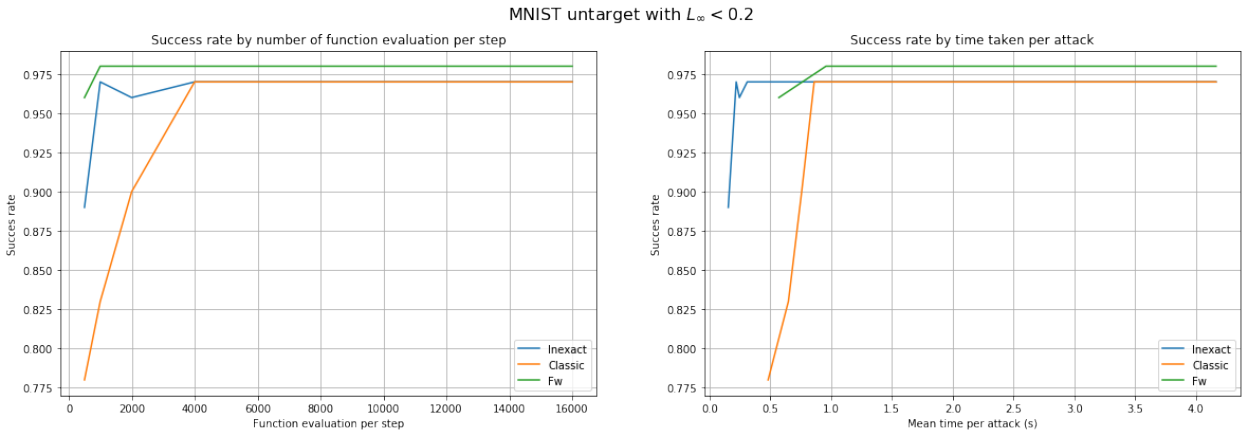

Figure 4: **Untargeted attack against MNIST with L_∞ norm:** : Success Rate in terms of function evaluation and average time

7.1.2 Untargeted with L_2

In the case of untargeted attack with the L_2 norm, *ZOO* outperformed all the other, both with ADAM and Newton as solvers, reaching a Success Rate of 1.00, with an average time of 5.68s with ADAM and an average L_2 distance of 1.36. The fastest algorithm has been the Inexact ZSCG which achieved an 0.83 Success Rate in 2.76s. Results can be seen in Table 2 and Fig. MU2.

Optimizer	Average L_2 distance	Success rate	Average time (s)
inexact	1.75	0.83	2.77
classic	1.30	0.75	2.34
frank-wolfe	1.36	0.79	8.31
zoo-adam	1.36	1.00	5.68
zoo-newton	1.36	1.00	12.30

Table 2: Result of untargeted attack with L_2 norm against MNIST

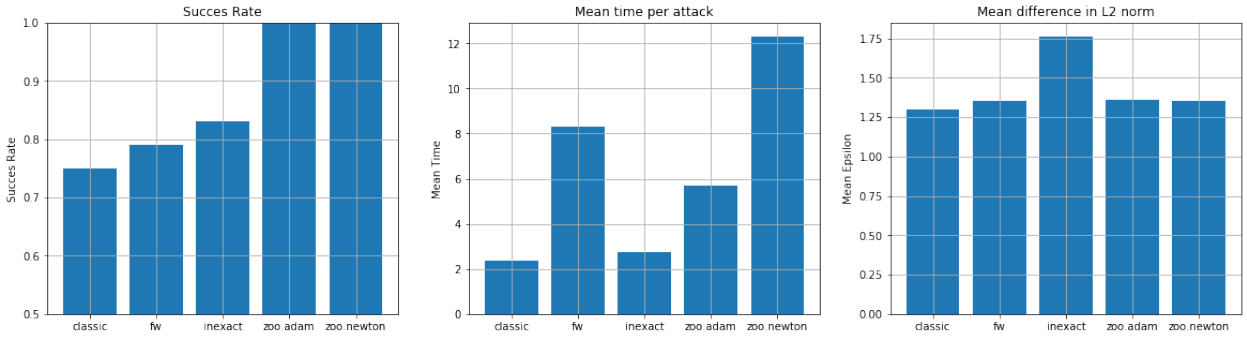

Figure 5: **Untargeted attack against MNIST with L_2 norm:** : Success Rate, Average Time and Mean difference in L_2 norm for each algorithm.

7.1.3 Targeted with L_∞

Results for targeted attack against MNIST with, as before, a feasible set given by $\|x_0 - x\|_\infty \leq \epsilon = 0.2$, showed, as expected, a general decrease of performances. This time in terms of *Success Rate* we had the best results given by *Frank-Wolfe* and *Inexact ZSCG*, since they achieved around 0.70 Success Rate against a 0.56 of *Classic ZSCG*. In terms of time taken the fastest has been *Inexact ZSCG* and *Frank-Wolfe* the slowest. Results can be seen in Table 3 and Fig. 6.

Optimizer	ϵ	Success rate	Average time (s)
inexact	0.2	0.70	5.069
classic	0.2	0.56	6.632
frank-wolfe	0.2	0.71	16.606

Table 3: Result of targeted attack with infinity norm against MNIST

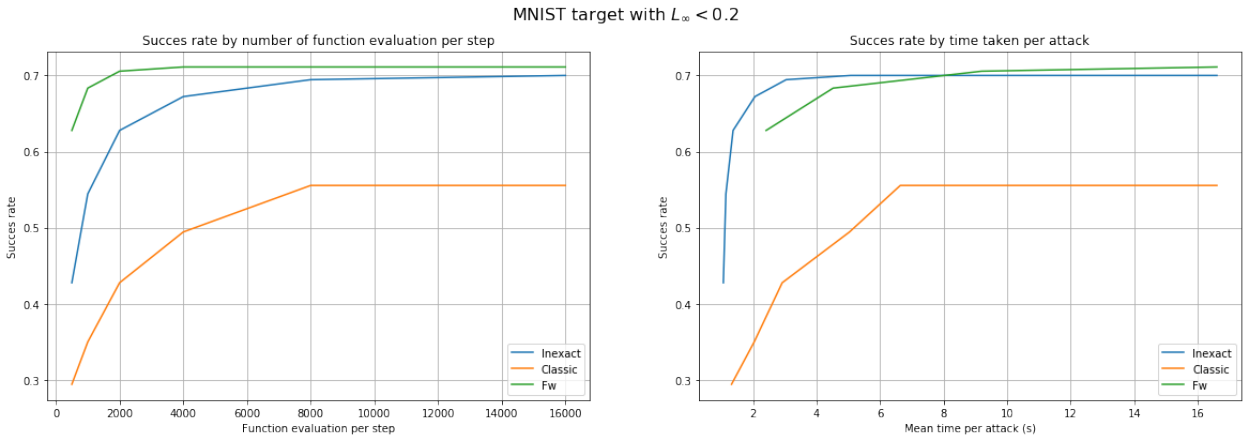

Figure 6: **targeted attack against MNIST with L_∞ norm:** Success Rate in terms of function evaluation and average time.

7.1.4 Targeted with L_2

All the algorithms, except *Classic ZSCG*, achieved almost 1.00 Success Rate. *Inexact ZSCG* like in the previous cases has showed to be the fastest (0.36 seconds per attack), being more than 10 times faster than the other algorithms, but in the other hand has the highest L_2 distance, with a value of 3.52, which is around 50% higher than the L_2 distances of the other algorithms. Results can be seen in Table 4 and Fig. 7

Optimizer	Average L_2 distance	Success rate	Average time (s)
classic	2.32	0.89	4.98
frank-wolfe	2.59	0.96	5.51
inexact	3.52	0.97	0.36
zoo-adam	2.42	0.98	6.20
zoo-newton	2.14	0.98	21.06

Table 4: Result of targeted attack with L_2 norm against MNIST

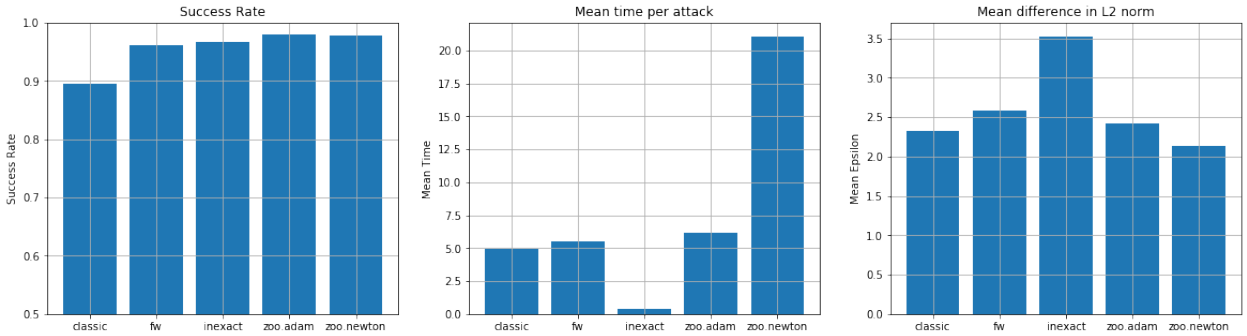

Figure 7: **targeted attack against MNIST with L_2 norm:** Success Rate, Average Time and Mean difference in L_2 norm for each algorithm.

7.1.5 MNIST Summary

When we used L_∞ *Inexact ZSCG* achieved peak performances together with *Frank-Wolfe* but being much more faster (from 3 to 6 times faster), while in the case of L_2 the algorithm with the best Success Rate is *ZOO* with both the solvers, but it is also the slowest.

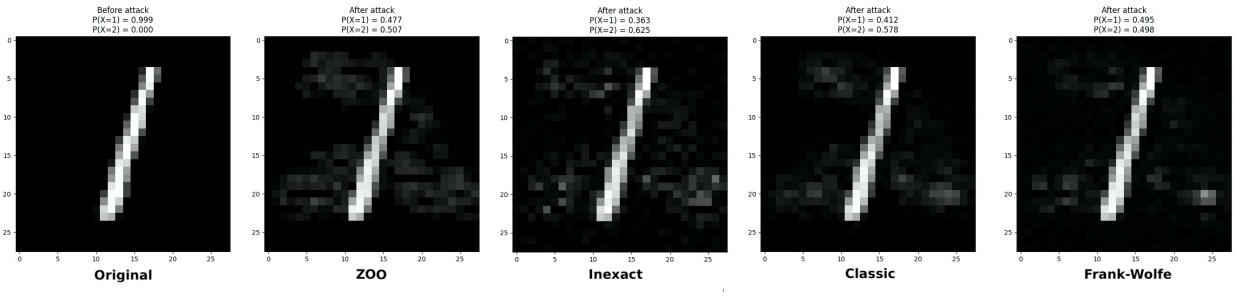

Figure 8: **Untargeted attack against MNIST with L_2 norm:** : Comparison of the generated Adversarial Images.

7.2 VGG16 - Cifar10

Like in the case of the *Ad-hoc model* trained on MNIST, four types of attack have been performed, with different level of ϵ in order to attack *VGG16* retrained on Cifar10 data, since using the same level of ϵ would have resulted in attack too easy to complete.

1. Untargeted attack using the feasible set $\chi = \{x \in [0,1]^{3072} : \|x_0 - x\|_\infty \leq \epsilon = 0.01\}$
2. Untargeted attack using the feasible set $\chi = \{x \in [0,1]^{3072} : \|x_0 - x\|_2 \leq \epsilon = 0.5\}$
3. Targeted attack using the feasible set $\chi = \{x \in [0,1]^{3072} : \|x_0 - x\|_\infty \leq \epsilon = 0.01\}$
4. Targeted attack using the feasible set $\chi = \{x \in [0,1]^{3072} : \|x_0 - x\|_2 \leq \epsilon = 1\}$

Where x_0 is the original input image used in the attack.

7.2.1 Untargeted with L_∞

Untargeted attack with $\epsilon \leq 0.01$ showed to be pretty easy to perform, resulting in high Success Rate for all the three algorithms used. *Frank-Wolfe* showed to be, like in the case of MNIST, the most accurate, while *Inexact ZSCG* showed to be a bit less accurate than the *classic* version (0.90 vs 0.96 Success Rate) but also the fastest (8.5 times faster than *classic ZSCG* and almost 2 times faster than *Frank-Wolfe*). Results are reported in Table 5 and Fig. 9.

Optimizer	ϵ	Success rate	Average time (s)
inexact	0.01	0.90	5.26
classic	0.01	0.96	43.21
frank-wolfe	0.01	1.00	9.30

Table 5: **Untargeted attack with L_∞ against Cifar10**

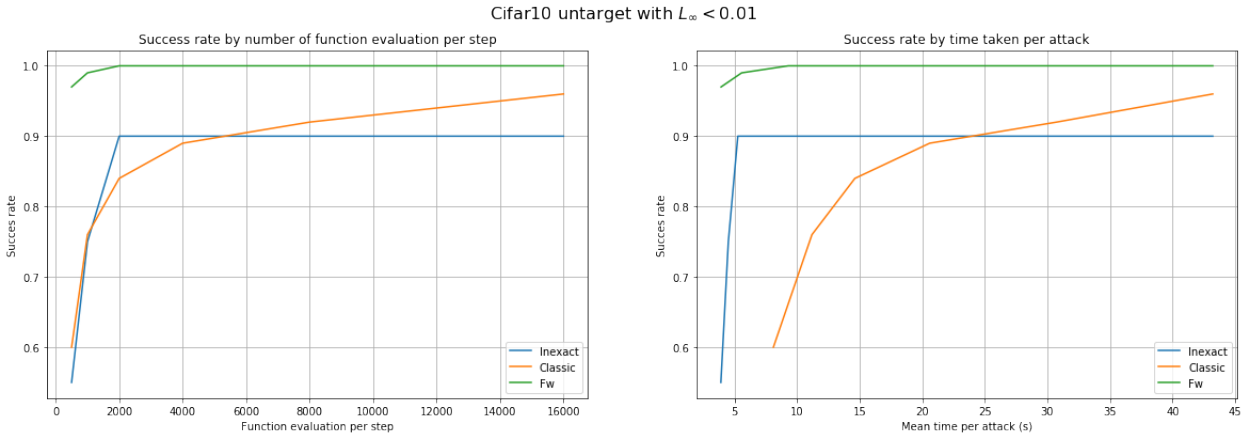

Figure 9: **Untargeted attack with L_∞ against Cifar10:** Success Rate in terms of function evaluation and average time. We can see how *Frank-Wolfe* fast reach perfect Success Rate, while *Inexact ZSCG* has better result than the classic version if given low amount of time, but the more time is given the more *Classic ZSCG* works better

7.2.2 Untargeted with L_2

Untargeted attacks using L_2 norm for the feasible set χ showed to be extremely easy, since all the four algorithms showed a perfect Success Rate. The fastest algorithm has been *Frank-Wolfe* taking 0.73s for attack, which means being from 4 times to 55 times faster than the other algorithm, it also achieved a low average L_2 distance, being the second lowest after the one achieved by *Classic ZSCG* (0.28 vs 0.19). Results can be seen in Table 6 and Fig. 10.

Optimizer	Average L_2 distance	Success rate	Average time (s)
inexact	0.50	1.0	3.07
classic	0.19	1.0	4.14
frank-wolfe	0.28	1.0	0.73
zoo-adam	0.36	1.0	5.02
zoo-newton	0.39	1.0	39.92

Table 6: Result of untargeted attack with L_2 norm against Cifar10

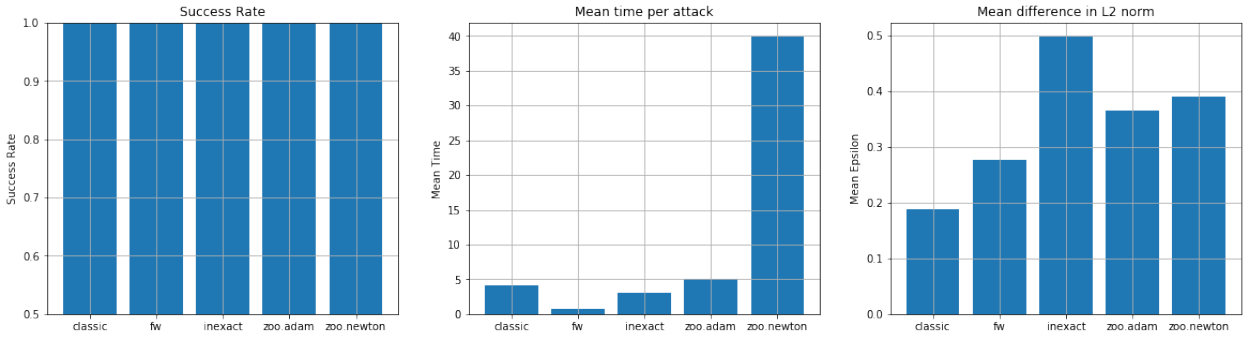

Figure 10: **Untargeted attack with L_2 against Cifar10:** Success Rate, Average Time and Mean difference in L_2 norm for each algorithm.

7.2.3 Targeted with L_∞

Targeted attack with L_∞ showed a much worse performance of *Inexact ZSCG*. In fact, if in the MNIST case *Inexact ZSCG* achieved the best performance together with *Frank-Wolfe*, here it has the worst Success Rate, reaching only a value of 0.40, while also being slower than its classic version (*classic ZSCG*). While *Inexact ZSCG* had poor performances, *Frank-Wolfe* showed to be incredibly efficient, reaching a Success rate of 0.86, but also being the slowest. Results are illustrated in Table 7 and Fig. 11.

Optimizer	ϵ	Success rate	Average time (s)
inexact	0.01	0.40	50.02
classic	0.01	0.53	44.47
frank-wolfe	0.01	0.86	83.03

Table 7: Result of targeted attack with L_∞ norm against Cifar10

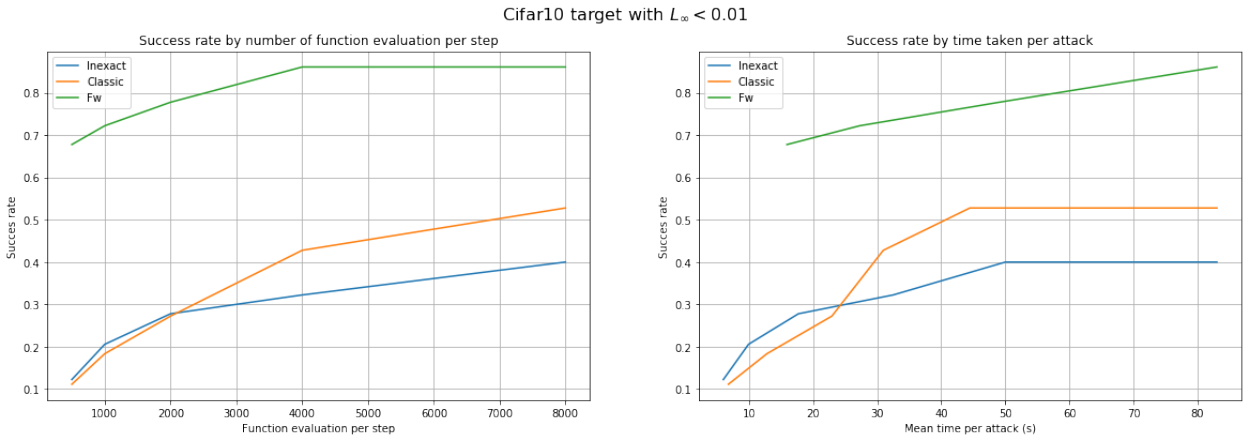

Figure 11: **Targeted attack with L_∞ against Cifar10:** Success Rate in terms of function evaluation and average time

7.2.4 Targeted with L_2

Targeted attack with L_2 showed to be very easy for all the algorithm which had a Success Rate between 0.98 and 1.00, except for *Inexact ZSCG*, which achieved only 0.63 Success rate. Like in the case of targeted attack using L_∞, *classic ZSCG* showed to be the one with minimum L_2 distance (0.39 vs 0.58, 70, 71), proofing to be able to perform an attack successfully with minimum distortion, while *Frank-Wolfe* was the one which performed attacks the fastest (3.77s vs 5.77, 19.55, 36,50). In the case of *ZOO*, using ADAM as a solver proofed once again to be much more faster than Newton, while reaching the same L_2 distance and Success Rate. Results can be seen in Table 8 and Fig. 12.

Optimizer	Average L_2 distance	Success rate	Average time (s)
inexact	0.99	0.63	32.96
classic	0.39	0.98	19.55
frank-wolfe	0.58	1.00	3.77
zoo-adam	0.71	1.00	5.77
zoo-newton	0.70	1.00	36.50

Table 8: Result of targeted attack with L_2 norm against Cifar10

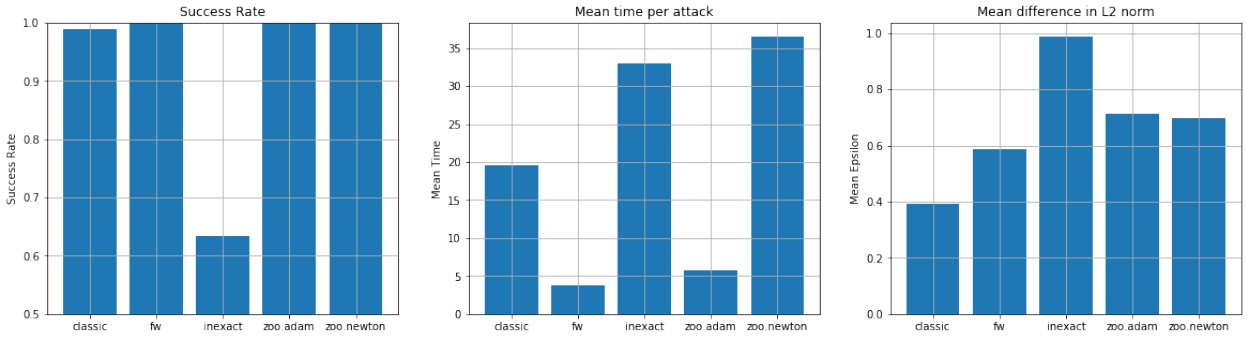

Figure 12: **Targeted attack with L_2 against Cifar10:** Success Rate, Average Time and Mean difference in L_2 norm for each algorithm.

7.2.5 Cifar10 Summary

Attacks against Cifar10, showed that the *Frank-Wolfe* algorithm is the most robust and can achieve peak performances in all the cases. It also showed a change of trajectory in the case of *Inexact ZSCG*, in fact if in the case of MNIST managed to be one of the best combining good Success rate and very fast attacks, now had mediocre performances in the case of untargeted attacks, having almost the same performances of its classic version (*Classic ZSCG*) and had terrible performances in the case of targeted attacks.
This disappointing behaviour of *Inexact ZSCG* in the case of targeted attacks against Cifar10 is not easy to interpret, but it's important to remind that given the limited computational resources a full grid search on the parameters couldn't been performed.

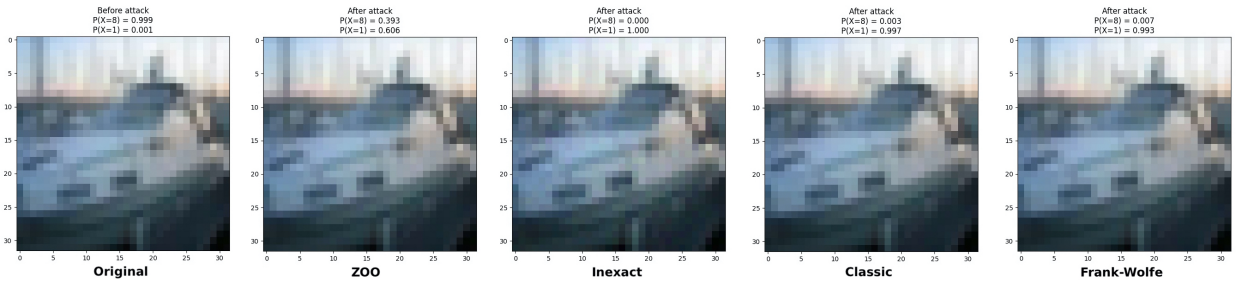

Figure 13: **Untargeted attack against Cifar10 with L_2 norm:** : Comparison of the generated Adversarial Images.

7.3 InceptionV3 - Cifar10

Differently, from the attacks against previous models, attacks against *InceptionV3* fine-tuned on Cifar10, have been only untarget, since to perform targeted attacks we would have needed a number of function evaluation prohibitively high (around 100 times more) while also having

problem of memory size, in fact optimization attacks that could run 4000 function evaluations in parallel, now can only perform 50 function evaluation in parallel before consuming all the GPU VRAM available in Colab, which is around 12 GB.
For this reason not only targeted attacks have been avoided but it also been given a maximum time to perform an untargeted attack, which is 300 seconds in the case of attack with L_∞(with $\epsilon \leq 0.03$) and 900 seconds in the case of attacks with L_2 (with $\epsilon \leq 4$).
Moreover since high maximum time given for an attack in concurrence for GPU limits and possible time outs given by Colab, we perform an evaluation of the algorithms with only 10 images, so the confidence interval of this results could be quite high.

7.3.1 Untargeted attack with L_∞

For each attack a maximum time of 5 minutes (300 seconds) has been given to each algorithm and a the value of ϵ has been set to 0.03. Results showed a strong performance of *Inexact ZSCG* which managed to converge under the maximum time in 9 cases out of 10 (against 6 out of 10 of *Classic ZSCG*). The worst performances has been given by *Frank-Wolfe* which had to converge in only 1 step since performing 2 step would have required almost 8 minutes, for this reason it managed to converge only in 1 case out of 10. Results are shown in Table 9 and Fig. 14.

Optimizer	ϵ	Success rate	Average time (s)
inexact	0.03	1.0	93.44
classic	0.03	0.6	137.45
frank-wolfe	0.03	0.1	270.48

Table 9: Result of untargeted attack with L_∞ norm against InceptionV3 fine-tuned on Cifar10

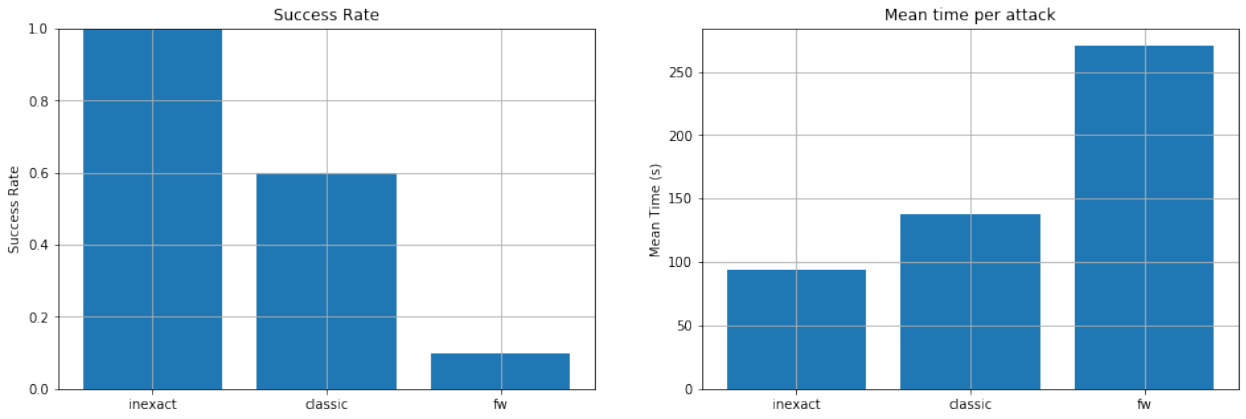

Figure 14: **Untargeted attack against InceptionV3 with L_∞ norm:** Success Rate and Average Time each algorithm.

7.3.2 Untargeted attack with L_2

When we switch to the L_2 norm, so that our feasible set could be defined as $\chi = \{x \in [0,1]^{268203} : \|x_0 - x\|_2 \leq \epsilon = 4\}$ things doesn't change a lot. *Frank-Wolfe* struggles to perform an attack since with the time given it can only perform one step, while *Inexact ZSCG* and *ZOO* with solver ADAM manage to reach good Success Rate (0.9 and 1.0 respectively) and with *Classic ZSCG* not being able to go above 0.4 Success Rate. Results are shown in Table 10 and Fig. 15.

Optimizer	Average L_2 distance	Success rate	Average time (s)
inexact	3.99	0.9	407.77
classic	1.24	0.4	313.99
frank-wolfe	3.99	0.1	534.50
zoo-adam	3.36	1.0	739.00

Table 10: Result of untargeted attack with L_2 norm against InceptionV3 fine-tuned on Cifar10

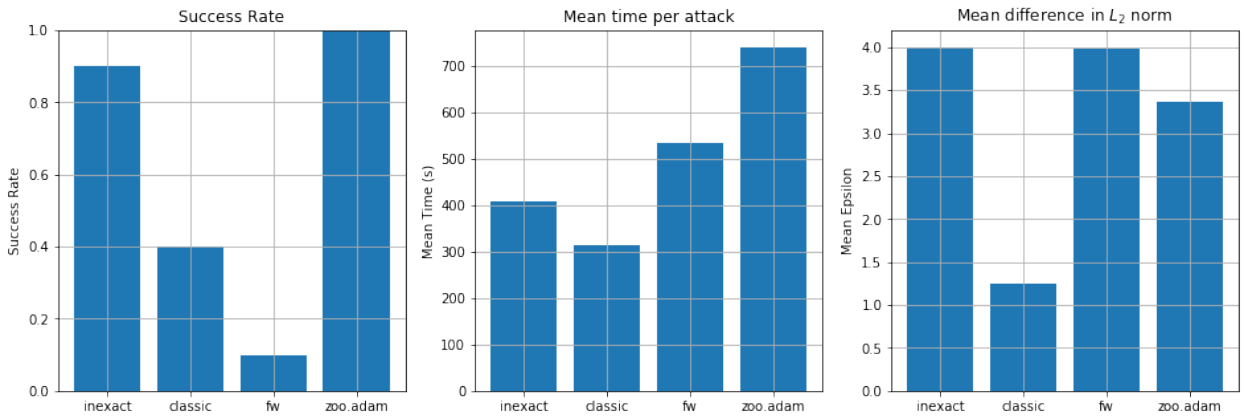

Figure 15: **Untargeted attack against InceptionV3 with L_2 norm:** Success Rate, Average Time and Average L_2 distance for each algorithm. We can see how *ZOO.adam* and *Inexact ZSCG* perform the best, but despite a low Succes Rate *Classic ZSCG* has a really low Average L_2 distance

8 Conclusions

After several attacks of different types have been performed against three different models (Ad-Hoc, VGG16, InceptionV3) some conclusion on the efficiency of the algorithms can be made. First of all we have seen a superiority of *Inexact ZSCG* against its classic version *Classic ZSCG* in the attacks against *Ad-Hoc* and *InceptionV3* models, but failing in the case of targeted attacks against *VGG16*. Secondly, good performances have been given by *Frank-Wolfe* in case of attacks against model with low dimension input (784, 3072), but

also giving bad performances in the case of high dimensions (*InceptionV3* with input images having 268203 dimension), in fact in both the attacks (L_∞ and L_2) it managed to perform successfully only one attack in ten. Finally *ZOO* method, evaluated only in the L_2 case, managed to reach always almost perfect Success Rate (from 0.98 to 1.00), but sometimes with bad results in terms of *time taken* and *Average* L_2 *distance*.

In the end, simulations, showed that a clear winner doesn't exist, but this final points can be made: if an high number of attacks must be performed or the model attacked is very complex, ZSCG algorithms, particularly *Inexact ZSCG*, seems to be the best since the major factor to take in consideration is the time; but if the model to attack is fairly simple or the number of attacks to perform is low, switching to *ZOO* or *Frank-Wolfe* could be the best choice, since the proofed to be the most accurate in terms of Success Rate if enough time is given. One of the reason of ZOO's high attack time is that it suffers from a poor query complexity since it requires to estimate the gradients of all the coordinates (pixels) of the image.

Further work, could be focus on performing a full *Grid Seach* on the parameters and to evaluate the models with a larger number of examples to have more reliable results.

References

[1] N. Akhtar and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," *IEEE Access*, vol. 6, pp. 14 410–14 430, 2018.

[2] P.-Y. Chen, H. Zhang, Y. Sharma, J. Yi, and C.-J. Hsieh, "Zoo," *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security - AISec '17*, 2017. [Online]. Available: http://dx.doi.org/10.1145/3128572.3140448

[3] J. Chen, D. Zhou, J. Yi, and Q. Gu, "A frank-wolfe framework for efficient and effective adversarial attacks," 2018.

[4] K. Balasubramanian and S. Ghadimi, "Zeroth-order nonconvex stochastic optimization: Handling constraints, high-dimensionality and saddle-points," 2018.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[6] S. I. J. S. Christian Szegedy, Vincent Vanhoucke and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015.

[7] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," 2016.

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

[9] S. Z. T. H. Lea Schonherr, Katharina Kohls and D. Kolossa, "Adversarial attacks against automatic speech recognition systems via psychoacoustic hiding," 10 2018.

[10] Y. Nesterov and V. Spokoiny, "Random gradient-free minimization of convex functions," *Foundations of Computational Mathematics*, vol. 17, 11 2015.

[11] [Online]. Available: http://yann.lecun.com/exdb/mnist/

[12] [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html

Appendices

Appendix dedicated to provide a few information on the four Optimizer Objects used in this report. Particularly, for each optimizer, information about the arguments of *constructor* and *run* method will be given.

A.1. Zero order Stochastic Conditional Gradient (Classic ZSGD)

The following tables will serve as a sort of API for the Class *ClassicZSGD* which can be found on *scripts/zeroOptim.py*.
A jupyter demo can also be found on main folder.

ClassicZSGD.__init__		
Name	Type	Description
model	torch.nn.Module	Neural network against which adversarial attack must be carried out
loss	loss.Loss	Loss function used to compare current and target output
device	torch.Device	Device on which optimizer must be run, where either neural network weights are stored. Default is *cuda*

Table 11: Constructor arguments for Classsic ZSCG

ClassicZSGD.run arguments			
Name	Type	Suggested	Description
x	(torch.tensor)		The variable of our optimization problem. Should be a 3D tensor
v	(float)	0.001	The gaussian smoothing
n_gradient	(list)	2000	Number of normal vector to generate at every step
ak	(list)	0.2	Momentum every step
epsilon	(float)		The upper bound of norm
L_type	(int)		Either -1 for L_∞ or x for L_x. Default is -1
batch_size	(int)		Maximum parallelization during the gradient estimation. Default is -1 (=n_grad)
C	(tuple)	[0,1]	The boundaires of the pixel. Default is (0, 1)
max_steps	(int)	100	The maximum number of steps. Default is 100
verbose	(int)		Display information or not. Default is 0
additional_out	(bool)		Return also all the x. Default is False
tqdm_disable	(bool)		Disable the tqdm bar. Default is False

Table 12: Constructor arguments for Classsic ZSCG

A.2 Zero order Stochastic Conditional Gradient with Inexact updates (Inexact ZSCG)

The following tables will serve as a sort of API for the Class *InexactZSGD* which can be found on *scripts/zeroOptim.py*.
A jupyter demo can also be found on main folder.

InexactZSCG.__init__		
Name	Type	Description
model	torch.nn.Module	Neural network against which adversarial attack must be carried out
loss	loss.Loss	Loss function used to compare current and target output
device	torch.Device	Device on which optimizer must be run, where either neural network weights are stored. Default is *cuda*

Table 13: Constructor arguments for Inexact ZSCG

InexactZSCG.run arguments			
Name	Type	Suggested	Description
x	(torch.tensor)		The variable of our optimization problem. Should be a 3D tensor
v	(float)	0.001	The gaussian smoothing
n_gradient	(list)	2000	Number of normal vector to generate at every step
gamma_k	(list)	1	Momentum at every step inside ICG
mu_k	(list)	0.0025	Stopping criterion at every step k inside ICG
max_t	(int)	100	The maximum number of iteration inside of ICG.
epsilon	(float)		The upper bound of norm
L_type	(int)		Either -1 for L_∞ or x for L_x. Default is -1
batch_size	(int)		Maximum parallelization during the gradient estimation. Default is -1 (=n_grad)
C	(tuple)	[0,1]	The boundaries of the pixel. Default is (0, 1)
max_steps	(int)	100	The maximum number of steps. Default is 100
verbose	(int)		Display information or not. Default is 0
additional_out	(bool)		Return also all the x. Default is False
tqdm_disable	(bool)		Disable the tqdm bar. Default is False

Table 14: Arguments for running an attack on Inexact ZSCG

A.3 Zero Order Optimization with Adam/Newton (ZOO)

The following tables will serve as a sort of API for the Class *ZOOptim* which can be found on *scripts/ZOOptim.py*.
A jupyter demo can also be found on main folder.

ZOOptim.__init__		
Name	Type	Description
model	torch.nn.Module	Neural network against which adversarial attack must be carried out
loss	loss.Loss	Loss function used to compare current and target output
device	torch.Device	Device on which optimizer must be run, where either neural network weights are stored. Default is *cuda*

Table 15: Constructor arguments for ZOO

ZOOptim.run arguments			
Name	Type	Suggested	Description
x	(torch.tensor)		The variable of our optimization problem. Should be a 3D tensor
c:	(float)	0.5	confidence
learning_rate:	(float)	0.01	Learning rate
n_gradient	(int)	128	Number of normal vector to generate at every step
beta_1:	(float)	0.9	ADAM hyper-parameter
beta_2:	(float)	0.09	ADAM hyper-parameter
solver:	(str)	adam	Either "ADAM" or "Newton"
batch_size	(int)		Maximum parallelization during the gradient estimation. Default is -1 (=n_grad)
C	(tuple)	[0,1]	The boundaries of the pixel. Default is (0, 1)
max_steps	(int)	10000	The maximum number of steps. Default is 100
verbose	(int)		Display information or not. Default is 0
additional_out	(bool)		Return also all the x. Default is False
tqdm_disable	(bool)		Disable the tqdm bar. Default is False

Table 16: Arguments for running an attack on ZOO

A.4 Black Box Frank-Wolfe (Frank-Wolfe)

The following tables will serve as a sort of API for the Class *FrankWolfe* which can be found on *scripts/FWOptim.py*.
A jupyter demo can also be found on main folder.

FrankWolfe.__init__		
Name	Type	Description
model	torch.nn.Module	Neural network against which adversarial attack must be carried out
loss	loss.Loss	Loss function used to compare current and target output
device	torch.Device	Device on which optimizer must be run, where either neural network weights are stored. Default is *cuda*

Table 17: Constructor arguments for FrankWolfe

FrankWolfe.run arguments			
Name	Suggested	Type	Description
x	(torch.Tensor)		Input tensor (3d image)
m_weight	(float)	8	Momentum weight, must be in [0-1] interval (beta)
step_size	(float)	0.2	Learning rate (gamma)
num_epochs	(int)	100	The maximum number of steps
l_bound	(float)		Upper bound of l- norm used (epsilon)
l_type	(int/str)		Type of l- norm which must be used
grad_num_iter	(int)	500	Number of gradient estimation steps (b)
grad_smooth	(float)	0.001	Gradient smoothing coefficient (delta)
grad_how	(str)	gauss	How to estimate gradient ('gauss' or 'sphere')
grad_batch_size	(int)		Batch size of gradient computations
clip	(tuple)		Clip pixel values in this interval
ret_out	(bool)		Wether to return all the computed images
verbose	(bool)		Whether to print out verbose log or not

Table 18: Arguments for running an attack on FrankWolfe