Institut national de recherche en informatique et en automatique

Survey on advances for learning deep neural networks

Internship Report 2020

Author:
Cadmos Kahalé-Abdou

Supervisors:
M. Alain Celisse
M. Hemant Tyagi

A report submitted in the context of

Master 1 of Applied Mathematics at Université Paris-Saclay

August 30, 2020

Contents

My thanks to M. Alain Celisse and M. Hemant Tyagi who followed me through my internship at INRIA Lille, even though we were far away in those times of global pandemic, and never failed to answer my questions and guide me in my work. My thanks to Mme Christine Keribin who have been following me for a year now with benevolence and who put me on the track of this internship.

1 Introduction

One of the main objectives of this paper is to establish a method of regression in the context of a study on neural networks. This being said, our main object of focus will be Single-Index Models (SIM) and Multiple-Index models (MIM), for which we will apply some learning algorithms in order to perform a regression. The idea behind focusing on these objects is that a one-layered neural network is a particular case of the Multiple-Index model. Solving a Multiple-Index model with a good tractability, and a low error rate would enable, in the idea of a cascade (see subsection 1.3), to solve multiple-layered neural networks in eventual extensions. Therefore, those models, as well as neural networks will be defined, and the link between the two will be put forward in section 1. The idea behind this study is to begin by solving the simple-index model in order to generalize it to the multiple-index one. Solving the multiple-index model means solving the 1-layered neural network by choosing the correct parameters. The idea in which we doing this, would be that once the 1-layered neural network is solved, we expect it can be applied to a general form of neural networks. Our focus here will be solely 1-layered neural networks, but this is what we would hope for in the long run.

1.1 Defining the single-index model [1]

We assume that we have a set of observations (X_i, Y_i), with $i \in 1, \ldots, n$. Those pairs are generated by the following regression model:

$$Y_i = f(X_i) + \epsilon_i. \tag{1}$$

Here, we call Y_i the *response variable*, and the X_i are d-dimensional random variables, ϵ_i are random errors, at least centered most of the time. The function $f(\cdot)$ is an unknown function $f : \mathbb{R}^d \to \mathbb{R}$. The assumption over f is that $f(x)$ has a very specific structure:

$$f(x) = g(a^\top x). \tag{2}$$

The function $g(\cdot)$ is known as the *link function*, and is defined as $g : \mathbb{R} \to \mathbb{R}$ and a is an unknown *index vector*. Generally, both those equations are referred to as *single-index regression models.* Those models are often found in the field in econometrics as they find a middle-ground between fully parametric and fully non-parametric models (McCullagh and Nelder, 1989)[6]. In fact, they are thoroughly used in the projection pursuit regression (cf. Friedman and Stuetzle, 1981 and Hall, 1989)[7].

The main estimation problems revolving around the single-index model is on a first hand, the estimation of the unknown function f, on the second one, the estimation of the index vector a. Here, we are mainly focusing on the second one, discussing the estimation of the index vector within different assumptions on f and g.

1.2 Defining the multiple-index model [2]

To generalize the idea of single-index models, we can define the *multiple-index model* as follows: In the same setting than the single-index model both for the elements of the couples (Y_i, X_i), the definition of f, the noise ϵ_i, what changes is the specific structure of f. This time f can be written as

$$f(x) = g(Ax) \tag{3}$$

with g an unknown m-dimensional link function, and A is a linear orthonormal mapping from the high-dimensional space $\mathbb{R}^d$ onto $\mathbb{R}^m$, so an $m \times d$ matrix, with m smaller than d (less rows than columns) a priori. A satisfies the condition $AA^\top = \mathcal{I}_m$.

1.3 About neural networks [3]

1.3.1 Example of a 1-layer artificial neural network

To begin with let's place ourselves in $\mathbb{R}^2$. We take any point in this subset, and our objective is is to return, by a certain transformation, an output that defines two categories. Let's say that if we apply this transformation on a point of $\mathbb{R}^2$, the output will be either category A, or category B.

We can use a lot of different functions that could work in this simple setting, but neural networks specifically uses a simple non-linear function called *activation function.* There is a lot of different activation functions depending on each set-up, but a classic one would be the sigmoïd function:

$$\sigma(x) = \frac{1}{1+e^{-x}}.$$

This function can be seen as a smoothed-out step function, and this explains why we may call it a *neuron*: Firing (output equals to 1) if the input is large enough, and remaining inactive (output equals to 0) otherwise. In addition to that, the sigmoïd function is convenient because of its differentiability and its property of having its derivative defined as:

$$\sigma'(x) = \sigma(x)(1-\sigma(x)).$$

We can influence the steepness and the location of the transition by scaling and shifting the argument. In the language of Neural Networks, we say that we are adding weights to the input and biasing it. For z in $\mathbb{R}^d$, for example, the activation function is $\sigma : \mathbb{R}^d \to \mathbb{R}^d$, and is defined by applying the sigmoïd function component-wise, therefore:

$$(\sigma(z))_i = \sigma(z_i), \tag{4}$$

And this notation will enable us to introduce the idea of layers in neural networks. In each layer, every neuron outputs a single real number which is passed to every neurons on the next layer. At the next layer, each and every one of those neurons, applies its own weight and bias at their input, then applies the activation function to generate the output. Therefore, if the real numbers produced by the neurons in one layer are collected into a vector x, then the vector of outputs for the next layer is written:

$$\sigma(Wx+b), \tag{5}$$

where W is a matrix and b is a vector. The matrix W is said to contain the weights, whereas the vector b is called the biases. Obviously, for the expression to make sense, W needs to have as many columns than x has rows, so, the number of neurons that yielded x at the previous layer. The number of components in b matches the number of neurons at the current layer.

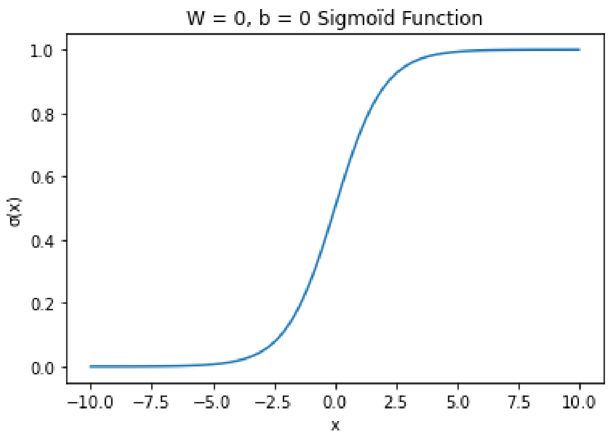


Figure 1: Example of an activation function: The sigmoïd

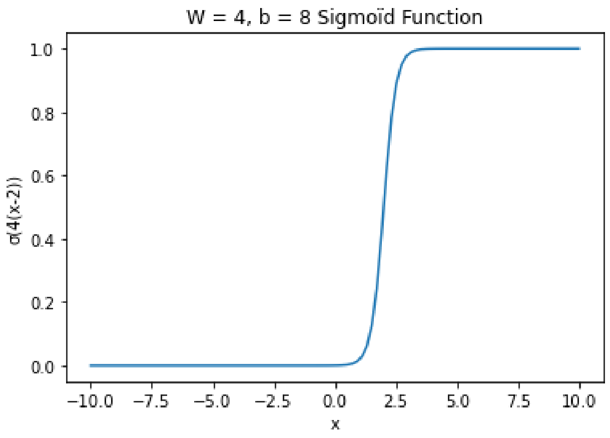


Figure 2: Example of a weighted and biased activation function

1.3.2 The relationship between multiple-index models and 1-layer neural networks

Now, by replacing the weight matrix W by an orthonormal mapping A, the activation function by a general m-variate function g, the equation (5) strongly reminds us of the multiple-index model set-up (3). In fact, since we can take and x in $\mathbb{R}^d$ from the form $x = x_0 + b_0$ with both x_0 and b_0 in $\mathbb{R}^d$, we can fully affirm that the output of a 1-layer neural network is exactly a special case of the multiple-index model as we defined it in (3).

1.3.3 Example of a multi-layered neural network

This being said, in order to complete the explanation on neural networks, we will extend a little bit our explanations on neural networks.

We are going to give an example of a multi-layered neural network, to at least have an idea of how it works. In (3), we have in the input layer two circles. This is because our input data has two components, we can see it as a point in $\mathbb{R}^2$. The second layer has three circles. This means that three neurons are being used. Since the input data is a point $x \in \mathbb{R}^2$, the weight is a matrix $W^{[2]} \in \mathbb{R}^{3\times 2}$, and the bias $b^{[2]} \in \mathbb{R}^3$ The output from layer 2 has the following form:

$$\sigma(W^{[2]}x + b^{[2]}) \in \mathbb{R}^3$$

Layer 3 has 5 neurons, each one receiving an input in $\mathbb{R}^3$. So the weights and biases for layer 3 may be represented by a $W^{[3]} \in \mathbb{R}^{5\times 3}$ and a vector in $b^{[3]} \in \mathbb{R}^5$. The output from layer 3 has the form:

$$\sigma(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}) \in \mathbb{R}^5$$

We apply the same ideas to the fourth layer and therefore obtain the expression for the whole network from the form:

$$\sigma(W^{[4]}\sigma(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}) + b^{[4]}) \in \mathbb{R}^2$$

Therefore, we can observe that we can move forward through a neural network by composing each time an activation function with a weight and bias parameters.

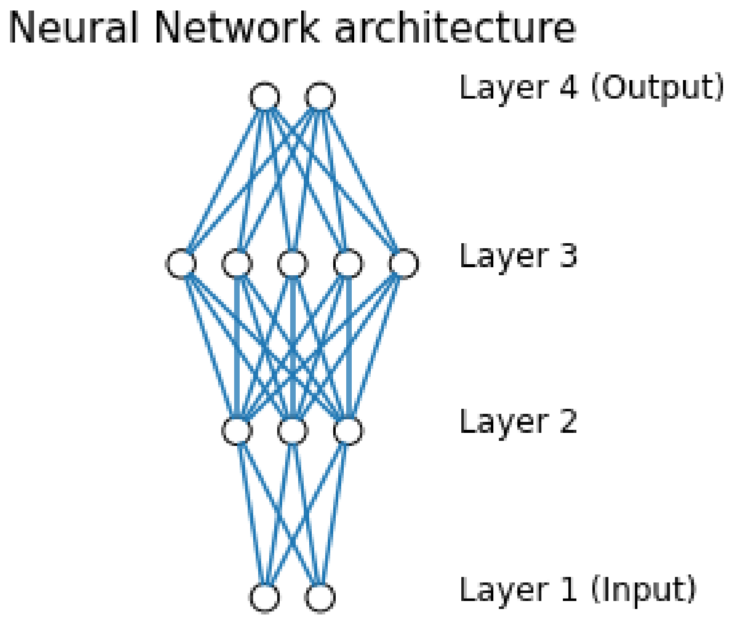


Figure 3: Neural Network of Four Layers (Figure template from [8])

1.3.4 The general set-up

The preceding example has given an idea of what is a neuron and how it works within the neural network with the activation function, and how it acts within layers. Generally speaking, at any layer, each neuron receives the same input: one real value from every neuron of the previous layer. One neuron of the current layer then produces a real value, distributed to all the neurons of the next layer. Two layers make the exceptions: the first layer called the input layer, since there is no previous layer, and the last one, the output layer, since there is no layer to pass information to, and therefore provides the overall output. The layers in between these are called "hidden layers". The idea behind this term is that those layers are performing intermediate calculations, but in no way, means that those calculations are always unknown. Deep learning is a term that morally implies that many hidden layers are being used.

1.4 Solving the multiple and single-index model

Since we have a sufficient amount of information on g, or the function is simply known, the equation (3) includes the hypothesis that the information on f is concentrated in the projected subset Ax, which will most likely be a subset of lower dimension. This means that it is by finding the subspace spanned by A that we find all the information we want on f, which is at the end of the day what we are looking for. Therefore, the problems related to this model are mainly finding the *effective dimension* m and the *index space* spanned by A. Therefore, the index space can be generated by A or a rotation of A. Indeed we can write $f(x) = g(A_0x)$

with $g(z) = (U_m z)$ and $A_0 = U_m^T A$, the model will still generate the same index space for U_m a rotation matrix. Moreover, once A is fixed, if g is still unknown, the link function g can be estimated by non-parametric methods. In the single-index model, since the index-vector is unique, estimating f is much more direct since the problem happens to be identifiable with no further considerations, as it will be shown in *Section 2*.

For the multiple-index model, we can find in the literature various methods of dimension reduction. Point-query methods, that will be studied in this report, has been developed in Fornasier et al. [9] to learn functions of few arbitrary linear parameters in high dimensions, under a set of assumptions on f and g, with which we can use the classical theory of *singular value decomposition* to estimate accurately the researched parameters. Under the assumption of normally distributed regressors, we can use the "sliced inverse regression" approach, proposed by Li [10]. Extensions of this method have been achieved, and Hristache et al. [2] wrote about an approach, relying on what is called average derivative estimations methods, or more recently Babichev et al. [5] presented the approach of "sliced average derivative estimation", which shows that score functions within sliced inverse regressions tends to have better performances than non-sliced estimations. One of the greatest issues with this method, is that most of the time, they are hardly computed because when d is large, the data in high dimensional space $\mathbb{R}^d$ is very sparse.

Those methods can most of the time be applied to single index models, and those ones won't have problems regarding dimension. The approach that will be followed here will be to begin by analysing single-index model as a foundation, then, with sufficient modifications and extensions of the single-index problem will be used to generalize the learning approaches to multiple-index problems.

1.5 Defining the problematic

Our final objective would be to establish a link between those regression methods and Deep Neural Networks because of the fact that the multiple-index model behaves similarly to a 1-layer neural network.

What are the requirements to be able to apply those regression methods and what are their limits? How can they enhance tractability? Error rates?

This report also has the ambition of introducing single and multiple-index models, under which assumptions those models are identifiable, and present examples of simulated learning methods within those models, in order to discuss about the difficulties encountered while computing those methods, their performances and accuracy.

2 Identifiability of the single-index model

In the single-index model setting, the basic properties we are going to inject to our very general definition of the single-index model using (1) and (2), are first of all a certain smoothness of the f function, and the properties of independence, zero mean and finite variance σ^2 for the noises. In Lin and Kulasekera [11], the assumption made to prove the identifiability of the model is the support of the function being a bounded convex set, with at least one interior point, but we will make the choice of changing this assumption into the following one, since it is a milder assumption that seems to be enough to achieve the proof:

Assumption 2.1. *The support S of $f(\cdot)$ is an Euclidean vector space included in $\mathbb{R}^d$.*

Therefore, $f(\cdot) \in L_2(S)$ is a d-variate function defined following (1) and (2).

Theorem 2.1. *Suppose that Assumption 1 holds, and $f(\cdot)$ is a non-constant continuous function on S. If*

$$f(x) = g(a^\top x) = h(b^\top x) \tag{6}$$

for some continuous functions g and h, and some $a, b \in D$, then $a = b$, and $g = h$ on $\{a^\top x | x \in S\}$, with $D = \{\alpha \in \mathbb{R}^d | \|\alpha\| = 1$ with first nonzero element positive$\}$

theorem from Lin and Kulasekera [11].

We note that there is no loss of generality restricting a and b to the D subspace since it is always possible to transform an $\mathbb{R}^d$ vector into a vector from the D subspace with a multiplicative factor, therefore the uni-dimensional space generated by $a^\top x$ is always the same.

Proof. This proof builds on the proof in Lin and Kulasekera [11], in a more explanatory and detailed way. It is clear that $g = h$ if $a = b$. Hence it suffices to show a = b.[11]

Indeed, since there is the hypothesis of continuity on both f and g, if $a = b$ the equality (6) yields $g = h$ on the $\{a^\top x | x \in S\}$ subspace.

Suppose $a \neq b$, Since $f(\cdot)$ is continuous, and non-constant on S, there exists a ball $B = B(x_0, r) \subset S$ for some x_0 such that $f(\cdot)$ is non-constant on B.

To be accurate vis-a-vis the topological considerations, it is important to note that restricting ourselves to a ball for the rest of the proof covers up for S not being a bounded convex. It is enough to have a subspace in which each interior point can be the center of a ball, the latter gives us the bounded convexity we need.

For the rest of the proof, we will follow the proof in Lin and Kulasekera [11] step by step.

By (6) and the fact that $a^\top a = 1$, we have for all $t \in (-r, r)$ that $x_0 + ta \in S$,

$$f(x_0 + ta) = g(a^\top x_0 + t) = g[a^\top (x_0 + ta)] = h[b^\top (x_0 + ta)] = h[b^\top x_0 + t(b^T a)] \tag{7}$$

The first and second equality rely on the fact that $a^\top a = 1$, the third equality uses (6). In the same way:

$$h(b^\top x_0 + t) = h[b^\top(x_0 + tb)] = g[a^\top(x_0 + tb)] = h[a^\top x_0 + t(b^\top a)]. \tag{8}$$

By the requirement that the first nonzero component of a and b are positive, we have $a \neq b$. Hence $|a^\top b| < 1$, since the case of equality of the scalar product to 1 is true only if $a = b$, and, by the continuity of g:

$$\begin{aligned} g(a^\top x_0 + t) &= h[b^\top x_0 + t(b^\top a)] = g[a^\top x_0 + t(b^\top a)^2] \\ &= \ldots = g[a^\top x_0 + t(b^\top a)^{2n}] = \ldots = g(a^\top x), \quad \forall t \in (-r, r) \end{aligned} \tag{9}$$

The set of equality (9) is retrieved by making a back and forth between (7) and (8), but it is understood that starting (9) by g or h is equivalent. For the sake of comprehension, we note that in (9), the g function will take all the even powers of $b^\top a$ while h will take the odd ones. Of course the role of g and h here is exchangeable. The last equality of (9) is obtained by the continuity of g that gives us the equality for $n \to \infty$, which implies $(b^\top a)^{2n} \to 0$.

Pick any $x \in B(x_0, r)$. Then $x = x_0 + \gamma$ for some unit vector γ and $t \in (-r, r)$. Hence, by (9), $f(x) = g(a^\top x) = g(a^\top x_0 + ta^\top \gamma) = g(a^\top x_0)$. This indicates that $f(\cdot)$ is constant on B, which is a contradiction. Thus $a = b$ and the proof is completed.

□

3 Stein's lemma for simple-index models

The object of this section is to estimate the parameter of the univariate link-function in the Single Index Model (SIM) setting, using Stein's Lemma.

Here, a is chosen laying on the unit sphere $\mathcal{S}^{d-1}$ to remain within the conditions of (2.1). Let Y be the response variable defined according to (1) and (2). Therefore, X is the covariate, $Y \in \mathbb{R}$ is the response and ϵ is the noise and is independent of X, and is centered. Moreover, we assume that the entries of X are independent and identically distributed (i.i.d.) random variables with density p_0.

3.1 The Gaussian case

In the Gaussian case, Stein's Lemma [12] is summarized as follow:

Proposition 3.1.1. *Let $X \sim \mathcal{N}(0,1)$ and $f : \mathbb{R} \to \mathbb{R}$ be a continuous function such that $\mathbb{E}|f'(X)| \leq \infty$. Then we have $\mathbb{E}[f(X)X] = \mathbb{E}[f'(X)]$.*

Which can be proved with the following standard proof from Yang et al. [4]:

Proof. Let $X \sim \mathcal{N}(0,1)$ and $f : \mathbb{R} \to \mathbb{R}$ be a continuous function such that $\mathbb{E}|f'(X)| \leq \infty$. We note $p(x)$ the standard normal distribution.

$$\begin{array}{rcl} \mathbb{E}[f'(X)] & = & \int_{\mathbb{R}} f'(x)p(x)dx \\ & = & -\int_{\mathbb{R}} f(x)p'(x)dx \\ & = & \int_{\mathbb{R}^d} f(x)xp(x)dx \\ & = & \mathbb{E}[f(X)X] \end{array}$$

$\square$

In the context of the SIM, we have two notable results that we are going to remind for clarity purposes:

1. $f(X) = g(aX)$, therefore $\mathbb{E}[f'(X)] = \mathbb{E}[g'(aX)]a$, and therefore $\mathbb{E}[f'(X)] \propto a$.

2. $\mathbb{E}[f(X)X] = \mathbb{E}[YX]$ which speaks for itself since ϵ is centered.

Using those remarks, we understand that $\mathbb{E}[YX] \propto a$, we have the following estimator based on the sample version of this proposition for X_i a Gaussian random variables and Y_i response variables in $\mathbb{R}$ [4]:

$$\hat{a}_{SL} = \arg\min_{\alpha \in \mathbb{R}} \frac{1}{n} \sum_{i=1}^{n} (Y_i X_i - \alpha)^2$$

Let $\hat{a}_{LS}$ be the least squares estimator defined as follows[4]:

$$\hat{a}_{LS} = \arg\min_{\alpha \in \mathbb{R}} \frac{1}{n} \sum_{i=1}^{n} (Y_i - \alpha X_i)^2 \qquad (10)$$

One of the main object of this section is to give an alternative justification for the results from Plan and Vershynin [13], which shows that under the assumption that X is standard Gaussian, that the least squares estimator is a good estimator for a, up to scaling. As $n \to \infty$, and since X is standard Gaussian, the law of large numbers ensures that both of those estimators are the same. This being said, as much as we would like this result to be easily extended in $\mathbb{R}^d$ with $d > 1$, it is not the case. This method only works when $a \in \mathbb{R}$. Thus, this part has achieved to give an alternative justification for the use of the least squares estimator in the Gaussian SIM setting when $\mathbf{d = 1}$ using Stein's Lemma. These observations will then leverage non-Gaussian versions of Stein's Lemma in order to deal with non-Gaussian covariates.

3.2 The unicity of the least squares estimator

According to Keribin [14] the model $\mathbb{E}(Y) = \alpha^\top X$ in regard of (10) is identifiable, if and only if one of the following equivalent properties are fulfilled:

- The columns of X are linearly independent

- X is full-rank

- X is injective

- $Ker(X) = 0$

Since $Y = \alpha^\top X + \epsilon$, if any element c of $Ker(X)$ exists such that $\mathbb{E}[Y] = \alpha^\top X = \alpha^T(X + c)$. The representation of $\mathbb{E}[Y]$ is not unique in α, and therefore α cannot be estimated.

4 Stein's lemma in the non-Gaussian case

We have just seen that when X follows a Gaussian distribution in the case of a SIM, then $\mathbb{E}[f(X)X] \propto a$. Therefore by approximating this expected value with i.i.d. observations, we can recover a up to a multiplicative factor. The limitations with this method being that this method only applies to normally distributed data and only applies to the single-index case. When the multiplicative factor is zero, then we simply cannot recover a (This will be shown and explained in this section). Therefore, to be able to retrieve the index vector or the index space in a broader set of cases, the method using Stein's Lemma in the Gaussian case needs to be extended. Let's note that in the Gaussian case is an special case of this one. Since the Gaussian case as we described it in the preceding section can only solve the model for $a \in \mathbb{R}$, here we can solve it for $a \in \mathbb{R}^d$ and elliptically distributed data.

4.1 Using the score function[4][5]

To enable milder assumptions on the f function and the distribution of X, a score function is required. Let $p : \mathbb{R}^d \to \mathbb{R}$ be a probability density function defined on $\mathbb{R}^d$. The score function $S_p : \mathbb{R}^d \to \mathbb{R}$ associated to p is defined as

$$S_p(x) = -\nabla_x[\log p(x)] = -\nabla_x p(x)/p(x)$$

In this definition of the score function, we bring up to the attention of the reader that it is indeed the derivative taken in respect of x that is used. We will omit the subscript x on the gradient and also the subscript p, when the underlying density p is clear from the context.

In a similar way to the preceding Gaussian process, we define a Stein's lemma that is applicable to non-Gaussian random variables and continuously differentiable link and density functions.

Lemma 4.1. *(Non-Gaussian Stein's Lemma, Stein et al. [15]) Let $f : \mathbb{R}^d \to \mathbb{R}$ be a continuously differentiable function and $X \in \mathbb{R}^d$ be a random vector with density $p : \mathbb{R}^d \to \mathbb{R}$, which is also continuously differentiable. Under the assumption that the expectations value $\mathbb{E}[f(X)S(X)]$ and $\mathbb{E}[\nabla f(X)]$ are both well-defined, we have the following generalized Stein's identity*

$$\mathbb{E}[f(X)S(X)] = -\int_{\mathbb{R}^d} f(x)\nabla p(x)dx = \int_{\mathbb{R}^d} \nabla f(x)p(x)dx = \mathbb{E}[\nabla f(X)]. \tag{11}$$

By setting the f function to be a constant function in the preceding identity, we have that $\mathbb{E}[S(X)] = 0$. Moreover, we have the following result by direct application of the lemma in the context of a SIM:

$$\mathbb{E}[YS(X)] = \mathbb{E}[g(a^\top X)S(X)] = \mathbb{E}[\nabla g(a^\top x)]a.$$

As $n \to \infty$, leveraging the $\hat{a}_{SL}$ estimator and using the law of large numbers on the following estimator:

$$\arg\min_{\alpha\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^{n}(Y_i S(X_i)-\alpha)^2$$

The expression of the problem we would want to minimize is:

$$\underset{\alpha\in\mathbb{R}^d}{\text{minimize}}\,\mathbb{E}[\alpha^\top\alpha - 2Y\alpha^\top S(X) + Y^2 S(X)^\top S(X)]$$

Which yields the following equivalent problem:

$$\underset{\alpha\in\mathbb{R}^d}{\text{minimize}}\{\alpha^\top\alpha - 2\mathbb{E}[Y\alpha^\top S(X)]\} \tag{12}$$

This problem has solution $\alpha = \mu a$, where $\mu = \mathbb{E}[g'(a^\top X)]$. Hence, this methods enables us to recover the unknown a as long as $\mu \neq 0$.

4.1.1 About the symmetric elliptical distribution case

When X follows a strictly positive density $p(x)$ which is differentiable with respect to the Lebesgue measure such that $p(x)\to 0$, when $\|x\|\to+\infty$, and when all the required conditions from the preceding section are fulfilled (differentiability of g and existence of the expected value), then the symmetric elliptical distribution case can be solved by using the score function, therefore the generalized version of Stein's Lemma. One of the remaining problems being that in many situation with symmetries, the proportionality constant is equal to zero and thus, the a cannot be recovered [5].

4.1.2 The limitations of this method

Here, the existence of the score function relies on the fact that the distribution of X has a strictly positive density which is differentiable in respect to the Lebesgue measure and such that $p(x)\to 0$, when $\|x\|\to+\infty$. Moreover, the score needs to be at least sub-Gaussian for consistent results. The strategy here is still to estimate the average derivative value (this is known as the Average Derivative Method (ADE)). The Gaussian assumption is not necessary anymore, and have been replaced the existence of a differentiable log-density. This being said, this method is still applicable only to recover the index vector, therefore in the single-index model case. In addition to that, the case where the proportionality constant equals zero, as said before also makes this method hit a wall.

We can overcome this limitation by using a higher order order of Stein's Lemma as it happens that we only use the first order one here, as proposed by Janzamin et al. [16]. Another method would be to combine Sliced Inverse Regression (SIR) associated with the score function, a method proposed by Babichev et al. [5], using the SIR method originally proposed by Li and Duan [17], Li [10], and Duan and Li [18].

The use of Stein's Lemma in the Gaussian and non-Gaussian case, assures that for sufficient smoothness, estimating the derivative of the link function is enough in order retrieve the value of the index-vector up to scaling. Moreover, the solution of the population version of the least square is, in both cases, Gaussian and non-Gaussian a good estimator for the index vector. Of course, we understand that it is a problem if the estimated value of the derivative of the link function is zero. This rules out the non-Gaussian phase retrieval (where $f(u) = u^2$) since we have $\mu = 0$ when X is centered, but this also remains true in the Gaussian and elliptical setting.

From an algorithmic point of view, this method enables us to recover the unknown parameter for an acceptable set of functions and distributions with some limitations for some cases in the non-Gaussian setting, where methods of truncation exist Yang et al. [4] to overcome the eventuality that the score function $S(X)$ or Y may be heavy tailed.

4.2 Second order Stein's lemma

The limitations of the "Stein's lemma" we just introduced is that for f a quadratic function, where $\mathbb{E}[f'(X)] = 0$, this version of Stein's lemma doesn't permit us retrieve the index-vector, moreover, it only enables us to solve single-index models, and not multiple-index ones. Another version of Stein's lemma using the second order derivative, or the Hessian in higher dimensions exists, and enables us in a first hand to solve the quadratic case, and on a second hand, to solve multiple-index models.

Proposition 4.2.1. *(Janzamin et al. [16]) Assume that the density of X is twice differentiable. In addition, we define the second-order score function $T : \mathbb{R} \to \mathbb{R}^{d\times d}$ as*

$$T(x) = \nabla^2 p(x)/p(x)$$

Then, for any twice differentiable function $g : \mathbb{R}^d \to \mathbb{R}$ such that $\mathbb{E}[\nabla^2 g(X)]$ exists, we have

$$\mathbb{E}[g(X)T(X)] = \mathbb{E}[g(X)(XX^\top - \mathcal{I}_d)] = \mathbb{E}[(a^\top X)^2(XX^\top - \mathcal{I}_d)] = 2a^\top a \tag{13}$$

We understand why the case where $X \sim \mathcal{N}(0, \mathcal{I}_d)$ doesn't extend easily to the $d = 1$ case, using the first order of Stein's lemma, since we need a second order score function $T(x) = xx^\top - \mathcal{I}_d, \forall x \in \mathbb{R}^d$. Then for the quadratic function for example, we take $g(a^\top x) = (a^\top x)^2$ and we obtain :

$$\mathbb{E}[g(X)T(X)] = \mathbb{E}[g(X)(XX^\top - \mathcal{I}_d)] = \mathbb{E}[(a^\top x)^2(XX^\top - \mathcal{I}_d)] = 2a^\top a \tag{14}$$

then retrieve the a vector by another optimization problem [19].

The idea behind the second order of Stein's lemma is the same as the first order one, but since we are looking for the second order differential, even if the first differential is zero, then the second one can be non-zero. The problem with MIMs using the first order Stein's lemma, to begin with, is that since f takes its values in $\mathbb{R}^d$ the first order differential is at best a gradient, so a vector, while we need to retrieve a matrix. Second order Stein's lemma will enable us to retrieve the Hessian. We understand that those are only dimensional concerns, but with that we can then define new minimization problems and perform principal components analysis assuming sparsity conditions which are theoritically detailed in Yang et al. [19] and that would enable us to retrieve the span of A.

The object of this section is especially to make the reader notice the validity and existence of a second order lemma, and to understand that the idea that motivates the methods using second order Stein's lemma are very similar to the first order ones.

4.3 Experiments for linear least squares

For X following a Gaussian distribution, it is enough for the link function to be continuously differentiable. Therefore, we can't assess of the computational validity of the method using ReLU link function ($g : x \to \max\{0, x\}$), or symmetric function like the square function or the absolute value function since the non-differentiability in 0 theoretically makes those functions not usable for our purpose.

We would expect the functions to work in the algorithmic process, in the meanwhile, since they are continuously differentiable almost everywhere from a Lebesgue point of view.

Whereas for non-Gaussian approaches, we require the link function to be at least continuously differentiable, and for the expectations value $\mathbb{E}[g(X)S(X)]$ and $\mathbb{E}[\nabla g(X)]$ are both well-defined. In this case we have ruled out the ReLU function and the absolute value function as their derivative is non-continuous, but can still assess of the applicability of the approach over sigmoïds and smoother functions.

In order to test the applicability of this method with certain activation functions that are not everywhere continuously differentiable we are to compute the Gaussian case with different link functions and assess the results

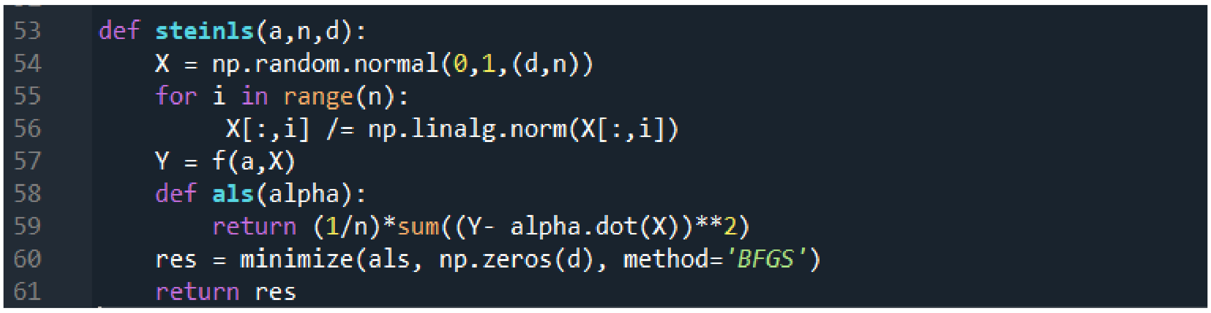
4.4 Testing the method on different link functions

We will be testing the Gaussian case for $d = 1$ as we are mainly interested by verifying if this method works with link functions that are not everywhere continuously differentiable.

We are first going to run the method on some polynomial, non-symmetric g to assess of the validity of the algorithm, and to confront our results, we are also going to make the experiment with functions that the theory rules out, typically the absolute function or the square function because of their symmetry and because the absolute function is not continuously differentiable everywhere.

4.5 The algorithm to test the least squares estimator using Stein's lemma based method

First we will code for the (10) estimator, computing samples for $X \sim \mathcal{N}_d(0, 1)$, then $Y = f(a, X)$, with $f = g(a^\top X)$, and g the function we will be changing. Of course we take care of normalizing the samples in X, because in $\mathbb{R}$ we can't know if we have retrieved the good vector up to a scaling, except if we retrieve the only α, such as $\alpha^\top a = 1$, which is specifically $\alpha = a$.

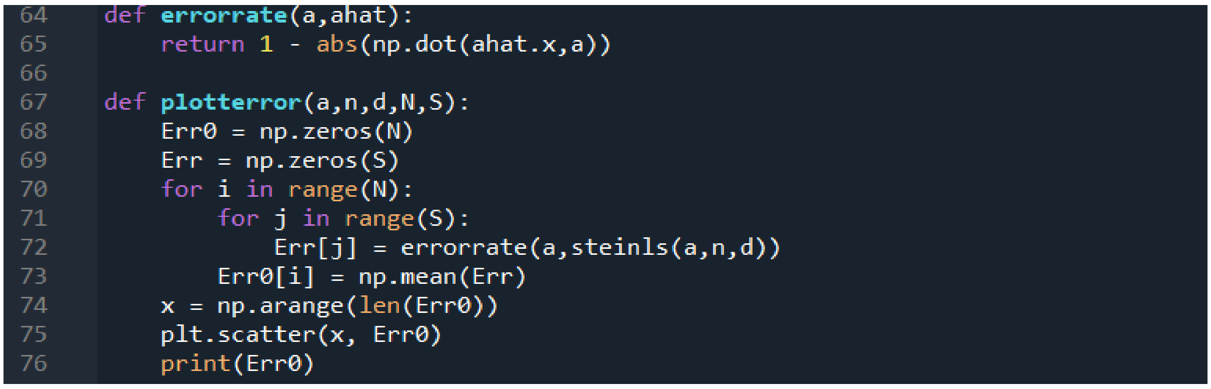


```
def steinls(a,n,d):
    X = np.random.normal(0,1,(d,n))
    for i in range(n):
         X[:,i] /= np.linalg.norm(X[:,i])
    Y = f(a,X)
    def als(alpha):
        return (1/n)*sum((Y- alpha.dot(X))**2)
    res = minimize(als, np.zeros(d), method='BFGS')
    return res
```

Figure 4: Function to solve the minimization problem of the Stein's lemma method in the Gaussian case

Then in order to test the validity of our retrieval we plot the error rate between our approximation and the a we are testing out for:

We compute S times an approximation of N samples, then for each set of approximation we take the mean error rate $1 - np.dot(ahat, a)$.



```python
def errorrate(a,ahat):
    return 1 - abs(np.dot(ahat.x,a))

def plotterror(a,n,d,N,S):
    Err0 = np.zeros(N)
    Err = np.zeros(S)
    for i in range(N):
        for j in range(S):
            Err[j] = errorrate(a,steinls(a,n,d))
        Err0[i] = np.mean(Err)
    x = np.arange(len(Err0))
    plt.scatter(x, Err0)
    print(Err0)
```

Figure 5: Retrieving the error rate

4.5.1 Polynomial non-symmetric functions

We see that for a polynomial, non symmetric g, the algorithm quickly reaches a very small value.

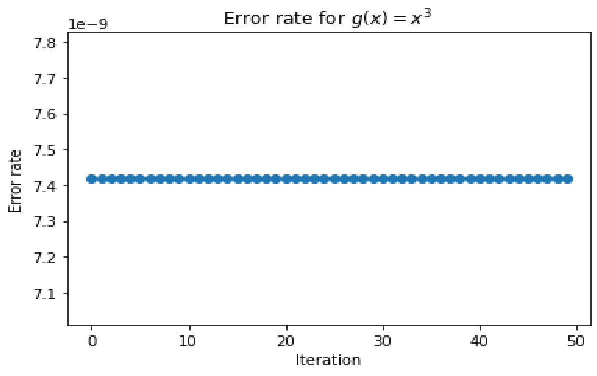


Figure 6: Error rates for a non-symmetric polynomial $d = 1$

For $g(x) = x^5$ we obtain the exact same graph. In order to test the validity of the set-up, we test the algorithm on $g(x) = x^2$ and $g(x) = |x|$ as well and we obtain the following results, organised in a histogram with 50 iterations using the mean of 20 approximations ($N = 50, S = 20$):

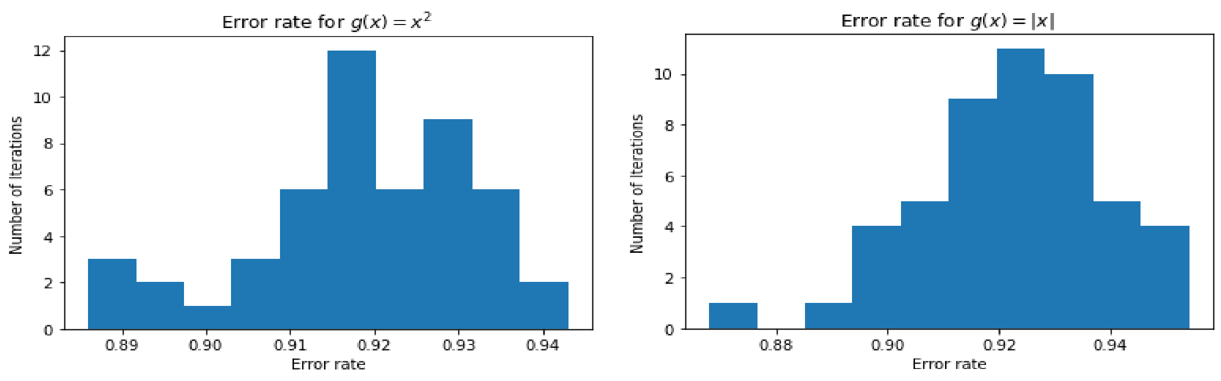


Figure 7: Applying the algorithm to functions ruled out by the method using Stein's lemma

Since we are unable to find good approximations of a in those conditions, we suspect our algorithm work as we expect it to in regard of those results, and the conditions g must meet in order for this method to work.

4.5.2 Testing activation functions with a Gaussian distribution and d=1

As the ReLU function is not continuously differentiable everywhere, we wouldn't expect it to work with this method but we could have positive expectations regarding the sigmoïd function since it is smooth enough. Testing the algorithm with those functions over 100 iterations of a mean of 15 approximations each time.

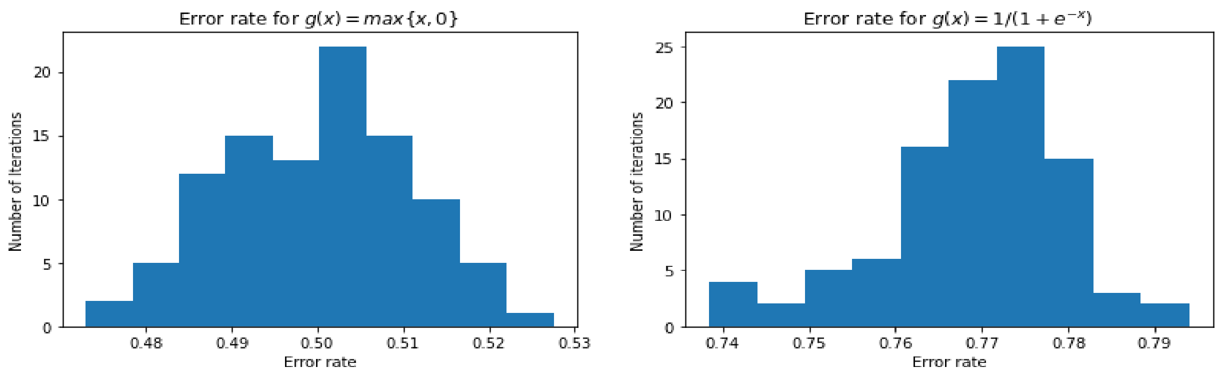
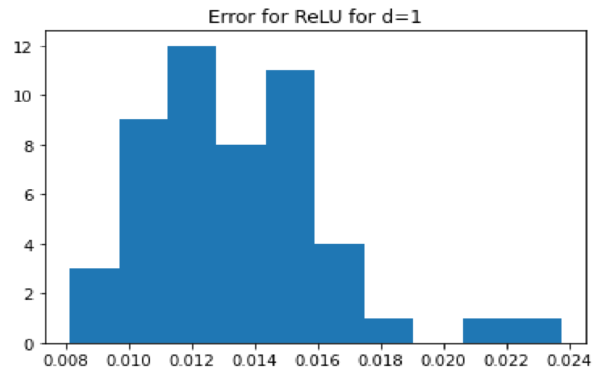


Figure 8: Error rates for ReLU function on the left and sigmoïd on the right

The fact that the ReLU function doesn't yield good results could have been theoretically expected but the fact the sigmoïd function has trouble in retrieving a, even in $\mathbb{R}$ is troubling.

As a control, we tried using the same method, in a "supervised learning" fashion, which means that we ran several times the algorithm and divided it by a in order to find the multiplicative factor, then tried to retrieve an unknown a, then by measuring our error by evaluating the euclidean distance between the approximation and the real value of a, the sigmoïd function still seems to perform poorly but the ReLU function gives us surprisingly good results. We also notice that when a is close to 0 in the ReLU case, the algorithm performs consistently better as we can see it on the right of the figure below. This being said, it is surprising since we could assume the the $g(x) = x$ and the ReLU function are similar on the $[0, 1]$ segment and would behave similarly, like (Figure 6). The error when measuring the euclidean distance after finding the multiplicative factor gives more positive results in the ReLU case but not the sigmoïd one.



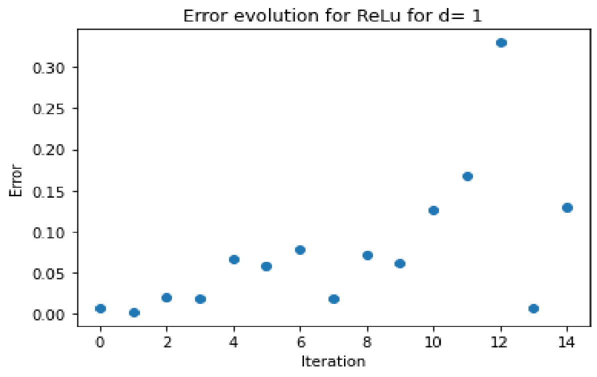


Figure 9: on the left we have the euclidean distance between the approximation and the real value, on the right the performances of the algorithm over [0,1] with a 1/15 interval between each iterations

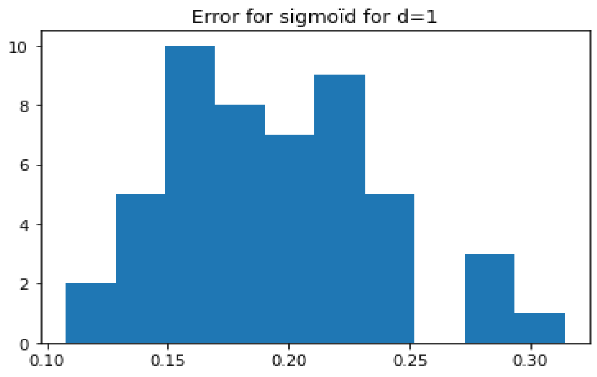


Figure 10: Performances for the sigmoïd function for the Euclidean norm

4.5.3 Testing Stein's lemma based method with a normal distribution for $d > 1$

When we change the value of d, this algorithm doesn't converge anymore. When $d > 1$ we automatically need to solve (12), and it isn't exactly the same problem anymore. This being said, we have assessed of the precision of the algorithm in the non-symmetric polynomial cases. To continue on this path, one should test non-Gaussian distributions in the $d = 1$ case, or test those same functions for $d > 1$ using the score function, and verify if activation functions can be used since the non-Gaussian Stein's lemma accepts when the f function is almost everywhere continuously differentiable.

5 The point-query algorithm

What we understood from applying Stein's lemma on some particular cases is that the gradient of f carries key information in the recovery of the index vector in the single-index model. In this section, keeping this in mind, we apply Taylor's expansion to our f function and notice various good outcomes. As we observe the first equality of equation (15), we can see that the directional derivative of f is equal to $[ag'(a^\top x)]^\top \varphi$. First of all, φ is known, simply because it is the direction in which we are performing the derivative, and we are arbitrarily choosing it. Then, in the brackets we notice that a, the index vector, has popped out of the g function. Of course, $g'(a^\top x)$ is a real number, and multiplies a, but in the idea, if we restrict ourselves to a unit sphere, and normalize our vector, if we can find $[ag'(a^\top x)]$, the multiplicative factor isn't an issue since we can still obtain the span of a. Then, since there is the scalar product between $[ag'(a^\top x)]$ and φ, the information we obtain is whether a is sufficiently close to φ or not. Approximating the directional derivative of f at a sufficient amount of directions is therefore necessary to recover the span of a.

What we have explained, in the idea, is that once we have obtained a sufficient amount of directional derivatives of f we can obtain enough information on a to recover it. The algorithms in Fornasier et al. [9] are about approximating the directional derivative in order to recover a. In order to do so, we need to focus on the second equality of (15). In the first member $\frac{f(x+\eta\varphi)-f(x)}{\eta}$, we see that we need a value of f at $x + \eta\varphi$, which requires a random sample direction φ, a random sample point x and a "step-size" eta. Of course, since we want to compute this value which is the directional derivative, we are just going to take the smallest η possible just to be close the actual value of the derivative. This being said, this is what "point-querying" is about. In our case, for a set of sample direction and a set of sample vectors, we test the proximity of our samples to the vector we are trying to recover, which is given by the value of the directional derivative of f, and then by solving an optimization problem we are able to find the vector that is the closest to what we looking for.

5.1 Retrieving the index vector: the algorithm

For this algorithm we are under the assumption that in (2), f is a ridge function, and therefore, a is vector in the $\mathbb{R}^d$ sphere, in $\mathbb{R}^d$ and g is a function that maps values from the euclidean ball $B_{\mathbb{R}^d}(1+\bar{\eta})$ towards $\mathbb{R}$, with $\bar{\eta} \in \mathbb{R}_+$.

What is required here is for the index vector a to be in the unit sphere, for us to be able to recover it. We can logically understand that we need to fix our sampling subset, somehow, to be able to have an euclidean and bounded subset, first for the scalar product to make sense i.e. to measure our results we assess the value of the scalar product between our approximation $\hat{a}$ and our index vector a, then, normalizing all our vectors will enable us to test only for the span of a, rather than the span and the norm. We know that the only vector that verifies $\hat{a}^\top a = 1$ is a itself. Therefore, we can have an idea of how close we are to a, since our approximations will lay in the unit ball, we can have an idea of whether our approximation is close or not in the least squares sense. Therefore, our sample points also needs to lay in the unit ball, but since we are also evaluating f in $f(x+\eta\varphi)$, we are going to slightly extend the support of f to $B_{\mathbb{R}^d}(1+\bar{\eta})$ where $\epsilon r < \bar{\eta}$, with r such as for all $\varphi \in B_{\mathbb{R}^d}(r), r \in \mathbb{R}_+$, for f to theoretically be able to evaluate every sample point we give it.

Now for clarity's purpose we are going to recapitulate where our different variables lay : $x \in B_{\mathbb{R}^d}(1+\bar{\eta})$, $\varphi \in B_{\mathbb{R}^d}(r), r \in \mathbb{R}_+$, such as $r\eta < \bar{\eta}$, and $\epsilon \in \mathbb{R}_+$.

Now, as long as $f \in \mathcal{C}^2(B_{\mathbb{R}^d}(1+\bar{\eta}))$, we can write the following Taylor expansion:

$$[ag'(a^\top x)]^\top \varphi = \frac{\partial f}{\partial \varphi}(x) = \frac{f(x+\eta\varphi) - f(x)}{\eta} - \frac{\eta}{2}\left[\varphi^\top \nabla^2 f(\zeta)\varphi\right] \tag{15}$$

for a suitable $\zeta(x,\varphi) \in B_{\mathbb{R}^d}(1+\bar{\eta})$.

As for $\varphi^\top \nabla^2 f(\zeta)\varphi$ which constitutes the remaining, it doesn't affect our reasoning, but we want this term to be uniformly bounded such as there is no interference with our train of thoughts. The part 2 of Fornasier et al. [9] assures that this term doesn't theoretically interfere with our results as it is uniformly bounded, and as it constitutes the remaining of the expansion, from a computational point of view, this term is negligible. Moreover, we can see that in both algorithms that appear in Fornasier et al. [9], this term is indeed negligible. We have all the ingredients to build an algorithm that will enable us to approximate the index vector.

What is interesting is that, this also works in the case of the multiple index model. Nothing in our reasoning requires a to specifically be a vector rather than a matrix. If we have the following set-up:

$$f(x) = g(Ax),$$

in a setting similar to 3, everything we have just said remains true, except of course the support of f and the extensions vis-à-vis dimension we need to do for our set-up to remain well defined. We can therefore write in the same way :

$$[\nabla g(Ax)^\top A]^\top \varphi = \frac{f(x+\eta\varphi) - f(x)}{\eta} - \frac{\eta}{2}\left[\varphi^\top \nabla^2 f(\zeta)\varphi\right]. \tag{16}$$

As we look into more detail the leftest member $[\nabla g(Ax)^\top A]^\top \varphi$, the term in brackets is a product between $\nabla g(Ax)$ an A, so this product remains in the span of A. Then by the same process, we can gauge the proximity of our sample direction vector with an element of the span of A. Our reasoning is that by computing enough points and directions, we want to have an idea of the subspace A spans. Then, the rest of the reasoning is similar to the single-index case. We can even say, that is A is an $m \times d$ matrix, the single index-case is a special case of that one with $m = 1$.

5.1.1 The algorithm

Now we can begin to build an algorithm that would enable us to approximate the span of A and equivalently the span of a, since it is a special case of the multiple-index case as we just said.

First of all, we need a set of sample points. Those will be:

$$\mathbf{X} = \{x_j \in \mathbb{S}^{d-1} : j = 1, \ldots, m_X\} \tag{17}$$

We can see them as the X_i of (1) as we are pulling those sample points out of a defined distribution (normal distribution for our test case) . Then we are going to evaluate f in those points, and near those points to be apple to approximate the directional derivative, for which we also need a set of sample directions:

$$\Phi = \left\{ \varphi_i \in B_{\mathbb{R}^d}(\sqrt{d}/\sqrt{m_\phi} : \varphi_{i\ell} = \frac{\chi}{\sqrt{m_\phi}} \right\} \tag{18}$$

with

$$\chi = \begin{cases} 1, & \text{with probability } 1/2 \\ -1, & \text{with probability } 1/2 \end{cases} .$$

The point of view which is adopted by Fornasier et al. [9] is the one of the matricial point of view. In a first place Φ is identified with the $m_\phi \times d$ matrix whose rows are the φ_i vectors. In order to write the equation (15) as an equality of $m_X \times m_\phi$ matrices, in a concise way, we collect the directional derivatives $g'(a^\top x_j)a$, with $j = 1, \dots, m_\mathcal{X}$ as columns in the $d \times m_\mathcal{X}$ matrix X:

$$X = \left(A^\top \nabla g(Ax_1), \dots, A^\top \nabla g(Ax_{m_X}) \right)$$

Let's precise that we do not have those values yet.We are still only defining each and every matrix in order to re-write (16).

Then, the $m_\phi \times m_X$ matrices Y and $\mathcal{E}$ are defined entry-wise by:

$$y_{ij} = \frac{f(x_j + \eta\varphi_i) - f(x_j)}{\eta} \tag{19}$$

and,

$$\eta_{ij} = \frac{\eta}{2} \left[\varphi_i^\top \nabla^2 f(\zeta_{ij}) \varphi_i \right]$$

The columns of Y are denoted by y_j and the columns of $\mathcal{E}$ are denoted η_j, with $j = 1, \dots, m_X$. Those matrices enables us to write (16) as the following factorization:

$$\Phi X = Y - \mathcal{E}$$

which will then be written as:

$$Y = \Phi X + \mathcal{E} \tag{20}$$

Let's precise that (20) is only but a re-writing of (16). For now there is no loss of information, we are just setting-up the matricial equation we are trying to solve practically for clarity purposes. Here, the elements we will be able to approximate will be in the Y matrix. This is the matrix that contains the bulk of the directional derivative's values. This equals to ΦX, the matricial product between the sample directions and the matrix that contains the gradient of g. Here ΦX is exactly the leftest member of the equation (16). The thing we are therefore computing are the values of f situated in the Y matrix.

The important point here, is the fact the if X is full rank, then we can apply a singular value decomposition for $X = USV^\top$, for which its column span will coïncide with the row span of A (i.e. $A^\top A = VV^\top$). Moreover, $V^\top$ gives us an alternative representation of f as follows:

$$f(x) = g(Ax) = g(AA^\top Ax = g(AVV^\top x) =: \tilde{g}(V^\top x),$$

where $\tilde{g}(y) := g(AVy) = f(Vy)$. If $\hat{X}$ is a good approximation of X, then we can expect the first m right singular vectors of $\hat{X}$ to have the same span of X, which means the same span of A too.

Let's note that this is also applicable in the case where $m = 1$, therefore in the singe-index case.

Fornasier et al. [9] writes the following algorithm:

5.1.2 Results and limits of the algorithm

What is done here is the estimation of the index vector for one set of sample of size $m_X m_\phi + m_X$ in order to have a fully sampled query. Indeed, looking at

$$y_{ij} = \frac{f(x_j + \eta\varphi_i) - f(x_j)}{\eta}$$

we see that in $f(x_j + \eta\varphi_i)$, for each of the m_ϕ sample direction we need m_X sample points, so $m_X m_\phi$ samples in total. Then for $f(x_j)$ we need m_X sample points.

What should be done, in order to have the best possible estimator according to the least squares, would be to obtain an empirical mean of the index vector's direction using singular value decomposition. The intuition behind this is that, since the derivative of the link function gives the direction of the index vector, querying the function in multiple points to be able to retrieve the gradient direction only gives us an estimation of this direction. Therefore, to be able to obtain the best possible estimator according to the least-squares, the SVD gives the main direction in which those vectors are pointing.

On one hand this algorithm enables us to solve multiple-index models up to a rotation, and on the other hand it enables a "better" solving of the single-index case since for several approximations of the span of a, we are retrieving the mean, in the least square sense, of all of our approximations.

On a first hand, the error between X and $\hat{X}$ directly depends of the number of inquiries m_Φ. Then, on another hand, $V^\top$ will be stable only if the singular values of X are separated from zero, which also depends on the number of samples but this time for **X**.

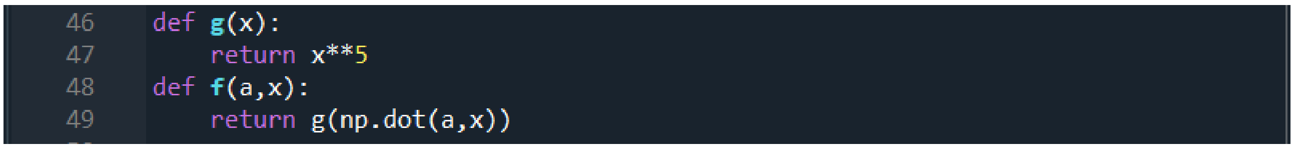
6 Setting up the algorithm

6.1 Retrieving the Index Vector for the SIM

In order to confront the algorithm developed in Fornasier et al. [9] to retrieve an index vector or a index space, we have coded it with python and measured the quality of the estimation under different sets of parameters

6.1.1 The code

In a first place, we define the functions that will enable us to set ourselves in the context of SIMs according to the (2) set-up:

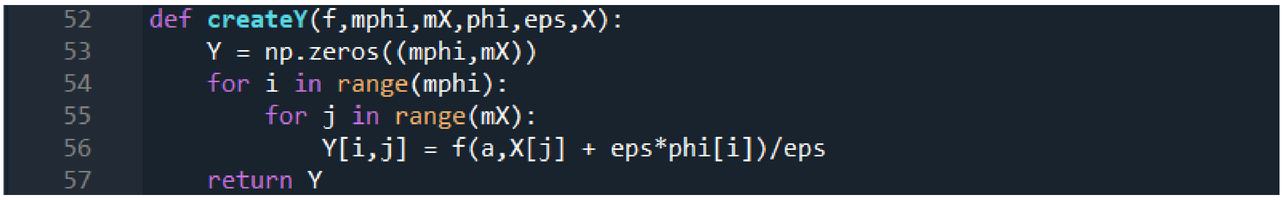


```python
def g(x):
    return x**5
def f(a,x):
    return g(np.dot(a,x))
```

Figure 11: Function definition in the context of SIM

Therefore, a polynomial function was chosen for g, because of its smoothness properties, and we chose $g(x) = x^5$ arbitrarily because of its lack of symmetry in $g(x) = 0$.

Then we create the matrices that will enable us to set up the algorithm:



```python
def createY(f,mphi,mX,phi,eps,X):
    Y = np.zeros((mphi,mX))
    for i in range(mphi):
        for j in range(mX):
            Y[i,j] = f(a,X[j] + eps*phi[i])/eps
    return Y
```

Figure 12: Creating the Y matrix

This function takes as arguments, the f function defined in 11, the size m_ϕ of the Φ set, the size m_X of the $\mathbf{X}$ set, the Φ matrix, the step size *eps* that is our η in the algorithm set-up, and the X matrix. With all those information we can create the Y matrix entry-wise as defined in (19). We haven't defined neither Φ or X yet, but since they are matrices defined with random components we are going to centralise their creation into one function. The size of the sets, them, are arbitrarily chosen at the beginning of the code so we can consider them as being some known and chosen parameters.

The last function we are creating, is the one that will enable us to find

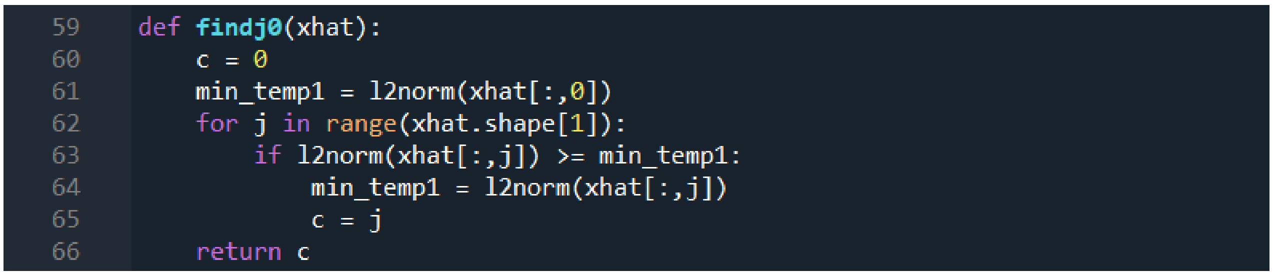
$$j_0 = \underset{j=1,\ldots,m_{\mathcal{X}}}{\arg\max} \|\hat{x}_j\|_{\ell_2^d}$$

and will be written as follows (13).

Here the "l2norm" function has simply been defined as the ℓ_2^d norm is defined. We note

$$\|x\|_{\ell_p^n} := \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

.



```python
def findj0(xhat):
    c = 0
    min_temp1 = l2norm(xhat[:,0])
    for j in range(xhat.shape[1]):
        if l2norm(xhat[:,j]) >= min_temp1:
            min_temp1 = l2norm(xhat[:,j])
            c = j
    return c
```

Figure 13: Function for finding j_0

Besides this, it is simply a function that enables us to find the j_0 that maximizes the ℓ_2^d norm for a given $\hat{x}_j$.

Now, everything is in place to write the algorithm that will give us the sought approximation. We can therefore write (Figure 14).

In the (Figure 14) function, d stands for the dimension of a while s stands for the number of times we want to approximate the a vector. All the approximations of a will then be placed in columns in an A matrix. Here we have chosen $eps = 10^{-7}$ in the screen-shot, as it is what suited best for the given function, this can eventually change. For bigger step sizes we have observed a lack of precision in the approximation which can randomly have a great impact on the quality of the algorithm. For a step size of 10^{-7} results we observed to be fairly consistent.

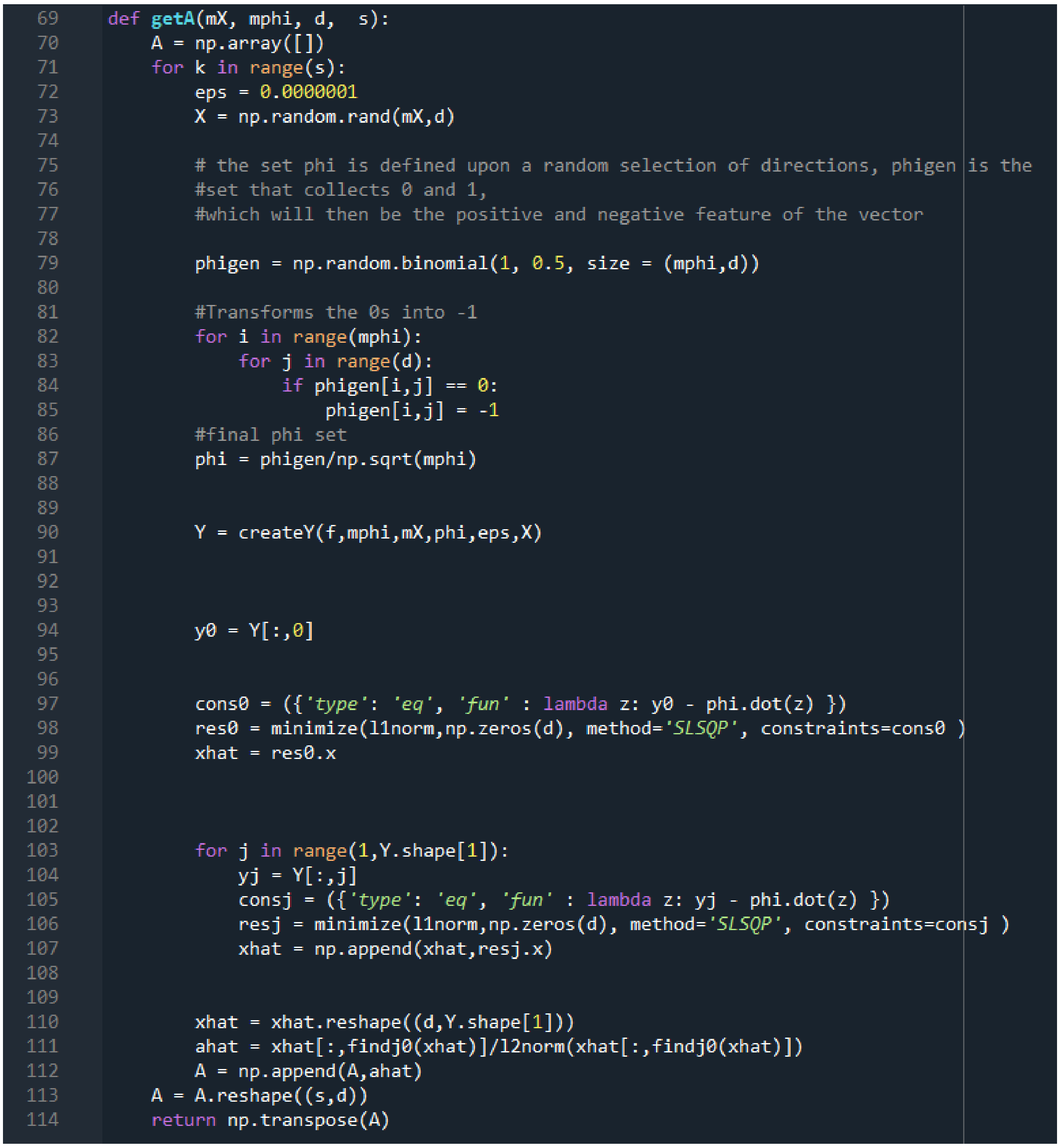
We define X as being a matrix with random points in the $\mathbb{R}^d$ sphere concatenated. Then to be able to code the Φ matrix we are creating random vectors following a binomial distribution of parameter 0.5. Then all the zeros will be transformed into -1s to suit (18).

Now we have the Φ and X matrix to be able to create Y. In the next step, we use the package *scipy.optimize.minimize* with the Sequential Least Square Programming (SLSQP) method, in order to solve the minimization problem

$$\underset{y_j = \Phi z}{\arg\min} \|z\|_{\ell_1^d}$$

We can see that the constraints have been defined accordingly. In this minimization sequence, the first step has been done outside of the loop as a control sequence to be able to operate on the code outside of the loop, but it could have been integrated in it. Then once the 3 first steps of **Algorithm 2** have been done, we set $\hat{a} = \hat{x}_{j_0}/\|\hat{x}_{j_0}\|_{\ell_2^d}$, and place the s, $\hat{a}$ approximations, in the A matrix in columns.

In order to measure the quality of our approximations we write (Figure 15), which will run k times the "getA" function, and will each time perform an SVD and will select the singular vector associated with the largest singular value. Each time, this will give us an approximation of the the index vector, and the error rate between this approximation and the sought vector. This will be measured by calculating the magnitude of the scalar product between a, (that we already have and are trying to retrieve) and the approximated vector then subtract it to 1. If the scalar product equals 1 then this means that the approximation is equal to a as it is a vector of the $\mathbb{R}^d$ sphere. The closer we are to 1, the better our approximation is. We take the magnitude, since we are able to retrieve the sought vector up to the sign, but it isn't a problem

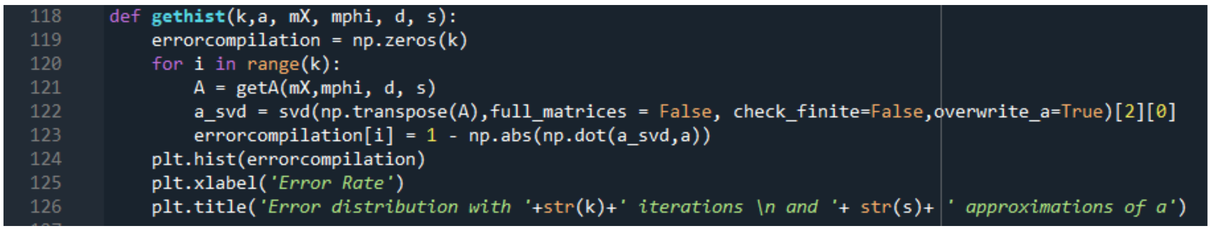


```python
def getA(mX, mphi, d,  s):
    A = np.array([])
    for k in range(s):
        eps = 0.0000001
        X = np.random.rand(mX,d)

        # the set phi is defined upon a random selection of directions, phigen is the
        #set that collects 0 and 1,
        #which will then be the positive and negative feature of the vector

        phigen = np.random.binomial(1, 0.5, size = (mphi,d))

        #Transforms the 0s into -1
        for i in range(mphi):
            for j in range(d):
                if phigen[i,j] == 0:
                    phigen[i,j] = -1
        #final phi set
        phi = phigen/np.sqrt(mphi)


        Y = createY(f,mphi,mX,phi,eps,X)



        y0 = Y[:,0]


        cons0 = ({'type': 'eq', 'fun' : lambda z: y0 - phi.dot(z) })
        res0 = minimize(l1norm,np.zeros(d), method='SLSQP', constraints=cons0 )
        xhat = res0.x



        for j in range(1,Y.shape[1]):
            yj = Y[:,j]
            consj = ({'type': 'eq', 'fun' : lambda z: yj - phi.dot(z) })
            resj = minimize(l1norm,np.zeros(d), method='SLSQP', constraints=consj )
            xhat = np.append(xhat,resj.x)


        xhat = xhat.reshape((d,Y.shape[1]))
        ahat = xhat[:,findj0(xhat)]/l2norm(xhat[:,findj0(xhat)])
        A = np.append(A,ahat)
    A = A.reshape((s,d))
    return np.transpose(A)
```

Figure 14: Retrieving the A matrix

since both still span the same space. Then we subtract this value to 1 in order to have an error rate rather than a retrieval rate. The error rates will compose an histogram. If this histogram is centered around a value close to zero, we can conclude that the algorithm will give us a good approximation most of the time.

It is very important to note that at each iteration, we randomly choose an a depending on the value of d, but most of the time this a is kept as long as we stay in the same dimension. The random process is situated in the computation of X and Φ, and for each iterations it is a new set of sample points and sample directions. The results do change depending on a. As we have seen it for example in the Stein's lemma computation, there are certain values for a in $\mathbb{R}$ that are easier to retrieve than others, but for the sake of observing how the parameters themselves impact the results. Therefore in those experiments, a is fixed and not randomised.



```python
def gethist(k,a, mX, mphi, d, s):
    errorcompilation = np.zeros(k)
    for i in range(k):
        A = getA(mX,mphi, d, s)
        a_svd = svd(np.transpose(A),full_matrices = False, check_finite=False,overwrite_a=True)[2][0]
        errorcompilation[i] = 1 - np.abs(np.dot(a_svd,a))
    plt.hist(errorcompilation)
    plt.xlabel('Error Rate')
    plt.title('Error distribution with '+str(k)+' iterations \n and '+ str(s)+ ' approximations of a')
```

Figure 15: Precision of the Algorithm

6.1.2 Results: No noise and d sparsity of the index vector - Challenging the number of approximations of a

According to Fornasier et al. [9], m_Φ is chosen by $m_\Phi = Ck\log(d)$, where C is a real number between 0 and 1 and k is the sparsity of the index vector. As we are choosing $k = d$ for now, we will be choosing $m_\Phi = E(\log(d))$ with E a function that gives the closest integer. This being said, at the beginning of the tests we chose some integers close enough to $E(\log(d))$ that would still make the SVD converge, and the minimization problem solvable. We understand that if m_Φ is too large, those are the two main problems that will occur and will prevent the program from yielding results.

At first, we gave a test run for program. We tried to figure out if the number of approximations of a really had a strong impact on the precision of the algorithm. The choice of parameter was: $d = 15, \quad mX = d, \quad mphi = 5, \quad g(x) = x^3, \quad \epsilon = 0.01, \quad s = 30$ and
$a = np.array([0.55184914, 0.04072427, 0.23083022, 0.04330443, 0.23910299, 0.21977374, 0.14437058, 0.35024734, 0.0434586, 0.1469475, 0.00365231, 0.5326643, 0.07174279, 0.08422273, 0.26493789])$
as a needs to be in the $\mathbb{R}^d$ sphere.

The program performed poorly over 50 iterations of the algorithm, but since we were trying to find out the impact of the size of s in the program, we continued in those parameters by changing the value of s.

We can see that the histogram is more or less center around $23 \sim 24\%$ of error rate which is pretty bad for a first shot. As we continue, we lowered the value of s and expected to have even worse results to come.

Even in (2) as we lowered s by 10, the quality of the approximations didn't seem to change. The histogram is still centered around 24% and the lowest error rate doesn't seem to go under 20%. Even though we notice that over 50 iterations, for $s = 20$ we are going to obtain a slightly higher maximum in the error rates, the histogram still seems to be the same.

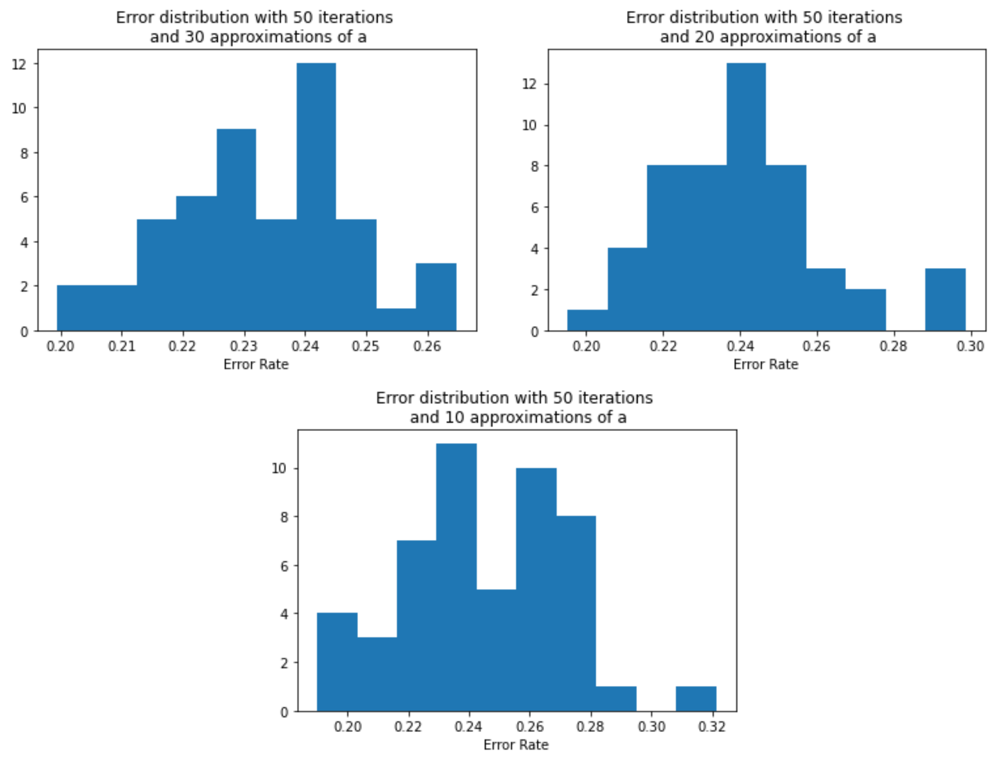


Figure 16

Finally for $s = 10$, the histogram still doesn't change much. With these results we can conclude that the number of approximation of a doesn't influence on the quality of the approximation, at least when the the results are poor to begin with, which we will see to be due to the size of ϵ, the size of d; the size of $mphi$ and sometimes the choice of the g function.

6.1.3 Results: No noise and d sparsity of the index vector - Challenging the dimension of the index vector

As the results for $d = 15$ were quite poor, we tried change the dimension of the index vector we are trying to retrieve. We tried lower dimension cases that we kept along the rest of the testings since the algorithm seemed to work well in this disposition.

6.1.4 Results: No noise and d sparsity of the index vector - Challenging the g function and the step size ϵ

As we have seen an improvement in the quality of results, we have chose to continue to test our parameters in lower dimension $d = 6$ to keep the calculation time acceptable. We changed the function $g(x) = x^3$ to $g(x) = x^5$ which gives us steeper variations, but those functions basically behave in the same manner. These changes already gave us a massive improvement in the results.

In (17) the parameters were: $d = 6, \quad mX = d, \quad s = 20, \quad mphi = 2, \quad \epsilon = 1e - 3$
$a = np.array([0.54487574, 0.53896174, 0.42187025, 0.27576556, 0.32196201, 0.23441412])$.

To challenge furthermore the improvement of quality of the results in answer to narrowing down the step size, we have lowered ϵ to the smallest size of acceptance of the SVD, therefore $\epsilon = 1e - 7$. Under this value the SVD wasn't able to converge anymore. Furthermore, the number of iterations of the algorithm went up in order to give us a better idea of what error

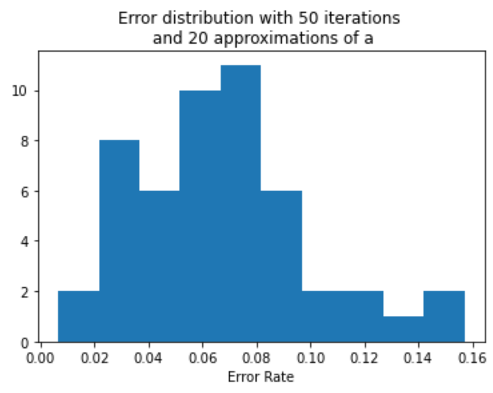


Figure 17: Error rates for d = 6 and $g(x) = x^5$

rate the algorithm was truly centered around for those parameters. This gave us (Figure 18) then with more iterations (Figure 19).

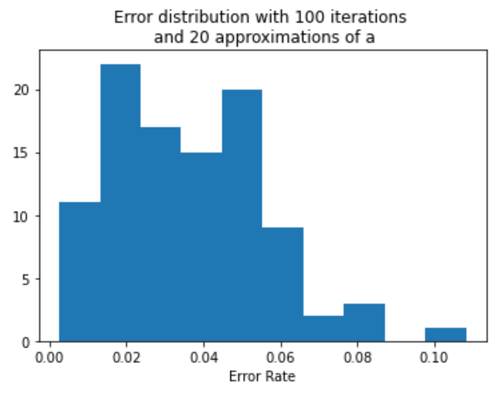


Figure 18: Results for $d = 6$ and $\epsilon = 1e - 7$

We chose $mphi = 2$ as $E(log(6)) = 2.$ and we can see that under those conditions the quality of the approximation is massively improved. The error rates are centered around 2.5% and very few approximations make a greater error than 7.5%, with the maximum error rate being a little bit more than 15%, with errors above 10% occurring only 4 times out of 500 in (Figure 19).

6.1.5 Results: No noise and d sparsity of the index vector - Challenging the number of approximations of a in the context of low error rates

As we have achieved set of parameters that gave us low error rates, the idea of applying an SVD rather than a single approximation of a needs to be challenged. Are we really getting improved results over the number of approximations of a within an iteration of the algorithm ? (The parameter that gauges the number of approximations on the index vector within an iteration is s as we have called it in the program.)Therefore, to challenge the impact of this parameter on the end result we tried to iterate 500 times the algorithm for s = 10 and s = 30 compared to the s= 20 case in (Figure 19) .

6.1.6 Rounding up the experiments with polynomial functions

The preceding sections have illustrated our train of thought while testing out the performances of the algorithm. In fact our main issue was the step-size for our gradient. Once this issue was fixed, as long as we kept $mphi = E(log(d))$, $mX = d$, results were fairly consistent for

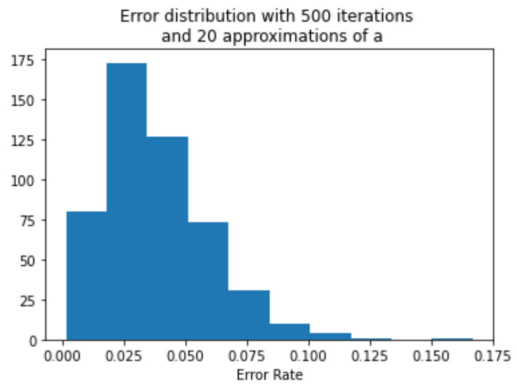


Figure 19: Results for $d = 6$ and $\epsilon = 1e - 7$

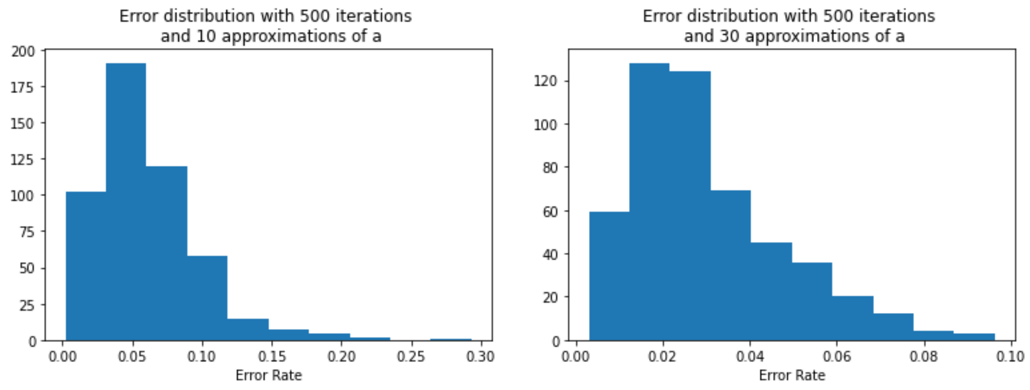


Figure 20: Results for $s = 10$, $s = 30$ and the same other parameters

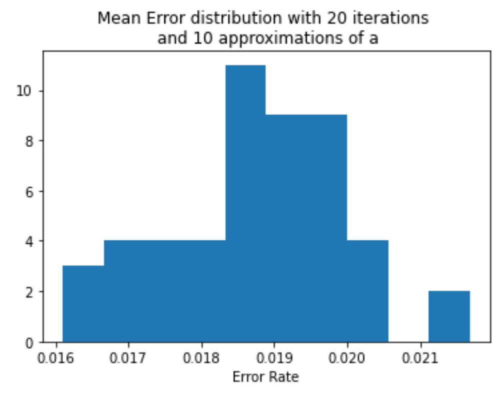
any d. At the moment when $mX > d$, precision in the retrieval would massively decrease, and the same thing would happen if $mphi > E(log(d))$. We assumed that those was the best parameters empirically for now, and decided to keep them for the sake of being able to change the g function from a polynomial to an activation function and confront the results in the same conditions.

6.1.7 Testing activation functions

Now that we have found a set of parameters that seems to yield good results, what is the most interesting for us is to know whether or not this method works well with activation functions such as ReLU or sigmoïds, or functions like the absolute function which is not continuously differentiable, and this method seems to yield some good results in those conditions. For those functions, we kept the number of iterations low, but we have computed each time the mean error. It is fairly the same process as before, but we can visualize better where the error rate is situated in those conditions. We have kept the same parameters than section 6.1.4 for consistency's sake.

We can see that over 50 times, the mean error over 20 iterations of the algorithm using 10 approximations of a, the error rate is between $1 \sim 2\%$ for both activation functions. Depending on a, the error rate can go up to $\sim 15\%$, but for each a the algorithm will have more or less the same mean precision over the iterations of the approximation. The same goes for the absolute function that will behave fairly the same in regard of the algorithm.

Regarding activation functions, we can safely say that Fornasier et al. [9]'s algorithm per-



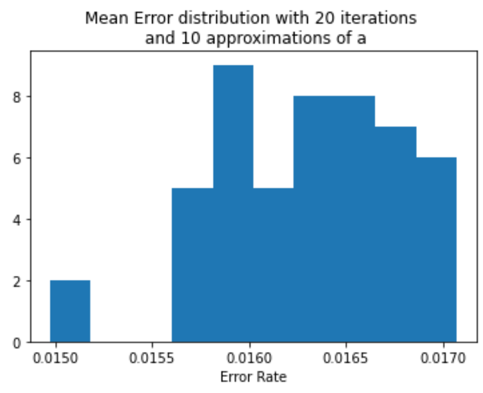


Figure 21: On the left, distribution of the mean errors for the ReLU function for 50 runs, on the right for the sigmoïd

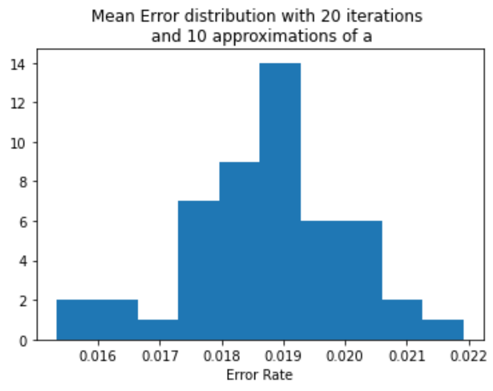


Figure 22: Distribution of the mean errors for the absolute function for 50 runs

forms much better than the method using the least squares estimator for the Gaussian case, even with $d > 1$.

6.2 Conclusion

The main focus in this study was retrieving index vectors in the single-index case, scouting which techniques could be used in order to solve this problem, under which condition and which parameter. Stein's lemma seemed to put us on the right track as we understood that most of the information on a was found in the gradient of f, but the conditions f must meet in order for those methods to work are not always optimal, especially when we want to make the link between SIMs and neural networks as activation functions are not always continuously differentiable. Another method we have focused on is the point-query method where we tried to approximate directional differentials of f, and leveraged Taylor's expansion to be able to retrieve, by solving a minimization problem, the index vector. This algorithm gave consistent results under a certain set of parameters. What could be done to improve our study on this point-query method would be to change the parameters more broadly and see what is the maximum value d can take while doing consistent results, and see at what value of s the improvements of the approximations doesn't match the running time anymore. Indeed, when d is too large, the waiting time if the algorithm is coded in python including some loops, the waiting time becomes absurdly high. In the idea, if such methods are applied to DNNs and retrieving a single vector take too much time, then retrieving information for a large set of hidden layers becomes even harder. This being said, by simply applying **Algorithm 2** to a MIM, we would be able to retrieve enough information on a 1-layer neural network, in the expectation of being able to make it on deeper neural networks in a fairly decent amount of time, and with a good choice of parameters.

References

[1] Marian Hristache, Anatoli Juditsky, and Vladimir Spokoiny. Direct estimation of the index coefficient in a single-index model. *Annals of Statistics*, pages 595–623, 2001.

[2] Marian Hristache, Anatoli Juditsky, Jörg Polzehl, Vladimir Spokoiny, et al. Structure adaptive approach for dimension reduction. *The Annals of Statistics*, 29(6):1537–1566, 2001.

[3] Catherine F Higham and Desmond J Higham. Deep learning: An introduction for applied mathematicians. *SIAM Review*, 61(4):860–891, 2019.

[4] Zhuoran Yang, Krishnakumar Balasubramanian, and Han Liu. High-dimensional non-gaussian single index models via thresholded score function estimation. In *International Conference on Machine Learning*, pages 3851–3860, 2017.

[5] Dmitry Babichev, Francis Bach, et al. Slice inverse regression with score functions. *Electronic Journal of Statistics*, 12(1):1507–1543, 2018.

[6] P McCullagh. Nelder. ja (1989), generalized linear models. *CRC Monographs on Statistics & Applied Probability, Springer Verlag, New York*, 81.

[7] Peter Hall et al. On projection pursuit regression. *The Annals of Statistics*, 17(2):573–588, 1989.

[8] Oliver Wilken. How to visualize a neural network. *https://stackoverflow.com/questions/29888233/how-to-visualize-a-neural-network.*

[9] Massimo Fornasier, Karin Schnass, and Jan Vybiral. Learning functions of few arbitrary linear parameters in high dimensions. *Foundations of Computational Mathematics*, 12(2): 229–262, 2012.

[10] Ker-Chau Li. Sliced inverse regression for dimension reduction. *Journal of the American Statistical Association*, 86(414):316–327, 1991.

[11] Wei Lin and KB Kulasekera. Identifiability of single-index models and additive-index models. *Biometrika*, 94(2):496–501, 2007.

[12] Charles M Stein. Estimation of the mean of a multivariate normal distribution. *The annals of Statistics*, pages 1135–1151, 1981.

[13] Yaniv Plan and Roman Vershynin. The generalized lasso with non-linear observations. *IEEE Transactions on information theory*, 62(3):1528–1537, 2016.

[14] Christine Keribin. Modélisation statistique (sta201) cours.

[15] Charles Stein, Persi Diaconis, Susan Holmes, Gesine Reinert, et al. Use of exchangeable pairs in the analysis of simulations. In *Stein's Method*, pages 1–25. Institute of Mathematical Statistics, 2004.

[16] Majid Janzamin, Hanie Sedghi, and Anima Anandkumar. Score function features for discriminative learning: Matrix and tensor framework. *arXiv preprint arXiv:1412.2863*, 2014.

[17] Ker-Chau Li and Naihua Duan. Regression analysis under link violation. *The Annals of Statistics*, pages 1009–1052, 1989.

[18] Naihua Duan and Ker-Chau Li. Slicing regression: a link-free regression method. *The Annals of Statistics*, pages 505–530, 1991.

[19] Zhuoran Yang, Krishnakumar Balasubramanian, and Han Liu. On stein's identity and near-optimal estimation in high-dimensional index models. *arXiv preprint arXiv:1709.08795*, 2017.