# Report

Kylian Ajavon

Summer 2020

# 1 Introduction

The goal of compressed sensing is to recover a signal from linear measurements. Mathematically, this is equivalent to solving the following underdetermined linear system, for $m < n$ :

$$y = Ax^* + e \tag{1}$$

where $y \in \mathbb{R}^m$ denotes the observed measurements, $A \in \mathbb{R}^{m \times n}$ is the measurement matrix, $x^* \in \mathbb{R}^n$ is the vector we wish to recover and $e \in \mathbb{R}^m$ is the noise vector.
It is impossible to solve such a system without making an assumption on the structure of $x^*$. In compressed sensing, that assumption is sparsity, i.e. most of the entries in $x^*$ are zero or negligible. However, in this paper [1], a different approach is proposed : instead of using sparsity, we consider a generative model $G : \mathbb{R}^k \to \mathbb{R}^n$, where $k \ll n$ and we assume that $x^*$ lies near the range of $G$ - namely, $x^*$ can be approximated up to a small error using $G$. The generative models used here are variational autoencoders (VAEs) [7] and generative adversarial networks (GANs) [4].

The main motivation of this summer research project was to apply generative models to compressed sensing. More specifically, we attempted to better understand from a theoretical point of view, how generative models could be used to effectively recover the signal $x^* \in \mathbb{R}^n$, and how to implement those theoretical results numerically.

# 2 How the algorithm works

The goal of the algorithm is to find an estimate for $x^*$ by minimizing the objective function

$$\|AG(z) - y\|^2 \tag{2}$$

with respect to z. For practical reasons, a regularization term $L(z)$ is added to the objective function. Due to certain properties of VAEs and GANs, this regularization term can be expressed as a multiple of $\|z\|^2$. Thus, in the algorithm, we minimize

$$\|AG(z) - y\|^2 + L(z) \tag{3}$$

where $L(z) = \lambda\|z\|^2$, and $\lambda$ measures the relative importance of the prior as compared to the measurement error [1].

The inputs of the algorithm are the following :

- $G : \mathbb{R}^k \to \mathbb{R}^n$ i.e. the generative model, given by the **model type**, the **size of the representation space** $k$ and the **length of the signal** $n$
- the **type of random measurement**, as well as the **number of measurements** $m$, which we use to obtain the measurement matrix $A \in \mathbb{R}^{m \times n}$

- the observed measurements $y \in \mathbb{R}^m$
- the mini-batch size
- the learning rate, used for gradient descent
- the parameter $\lambda$, which we use to obtain the regularization term $L(z)$ in the objective function (*L2 measurement weight loss, weight on z prior*)
- the optimizer type
- the maximum number of iterations
- the number of random restarts
- the momentum, which helps accelerate the stochastic gradient descent algorithm

As mentioned above, the algorithm should return a vector $\hat{x}$ which approximates $x^*$.

Procedure :

**Step 1** Draw $z \sim P_Z$. Since $G$ is known, we can then get $G(z)$.

**Step 2** Get a random matrix $A \in \mathbb{R}^{m \times n}$ from the measurement type, the number of measurements $m$ and the length of the signal $n$.

**Step 3** Get a random noise $e$ from its standard deviation.

**Step 4** Get the measurement vector as defined in the first equation, $y = Ax^* + e$.

**Step 5** From $A$ and $y$, which are given, define $loss(z) = \|AG(z) - y\|^2 + L(z)$. The regularization term is given by $L(z) = \lambda\|z\|^2$.

**Step 6** Get the gradient of $loss(z)$ with respect to $z$.

**Step 7** Update $\hat{z}$ by using the previous $z$, the gradient of $loss(z)$ evaluated at $z$ and the learning rate.

**Step 8** Update the the measurement error $\|AG(\hat{z}) - y\|^2$.

**Step 9** Evaluate the gradient of $loss(z)$ at $\hat{z}$. If this gradient is "close" to the zero vector, go to the next step. If not, then restart from step 7 and keep updating $\hat{z}$ and the measurement error. If after the maximum number of iterations, the gradient is still not "close" to the zero vector, choose the last updated values of $\hat{z}$ and the measurement error, and go to the next step. Steps 5 through 9 describe the gradient descent algorithm to minimize $loss(z)$ up to a certain accuracy.

**Step 10** The vector $\hat{x}$ we wished to obtain as a reconstruction of $x^*$ is now given by $G(\hat{z})$, which is the output of the algorithm.

# 3 More on Generative Models

Generative models are neural networks that are trained on a given dataset, and learn to create new data samples. Throughout this section, we will consider the case where the dataset of interest is a collection of images. A generative model takes as input a vector (often called *latent input vector*) sampled from a Gaussian distribution. It is then trained to output a new image that *resembles to* the images in the dataset.

## 3.1 Generative Adversarial Networks

Generative Adversarial Networks (GANs) [4] are comprised of two neural networks : a **generator** and a **discriminator**. The generator is a classic generative model, which is trained on a given dataset and tries to recreate new data samples that resemble to those in the dataset. The discriminator is trained to classify the images into two categories : the real images (from the dataset) and the generated images. The two neural networks are trained simultaneously and compete with each other. Namely, the generator tries to trick the discriminator into classifying a generated image as a real image, whereas the discriminator's goal is to always be able to distinguish the real images from the generated ones. We usually say that the *training has converged* when the discriminator can't tell the difference between the real and fake images.

To translate this into mathematical terms, we first define a generator $G$ and a discriminator $D$, where $G$ and $D$ are differentiable functions representing *multi-layer perceptrons*. Let the random vector $z$ be the input for the generator. Let $x$ represent the data.Then, $G$ and $D$ play the following two-player minimax game with value function $V(G, D)$ [4] :

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))] \tag{4}$$

The discriminator $D$ should output a scalar in the interval [0,1], representing the probability that its input is from the dataset. Namely, $D(x)$ represents the probability that $x$ came from the data distribution $p_{data}$ and $D(G(z))$ denotes the probability that $G(z)$ (*the image generated from* $z$) is also from the data distribution. Thus, $D$ is trained to maximize the entire value function, whereas $G$ is trained to minimize the second term.

## 3.2 Deep Convolutional Generative Adversarial Networks

There exists a class of convolutional neural networks called Deep Convolutional Generative Adversarial Networks (DCGANs) [8], that present a number of advantages compared to

GANs, in particular their stability under training. The generators of DCGANs also have interesting vector arithmetic properties [8] (*feature arithmetic*).

# 4 Implementing GANs with Python

As mentioned in the introduction, one of the main objectives of this project was to "uncover new things about generative models, and their connection to compressed sensing". As such, it felt natural to explore further how generative models work... Particularly, for a generative model $G : \mathbb{R}^k \to \mathbb{R}^n$, how does the latent space $\mathbb{R}^k$ and its subspaces relate to different components of the generated image (shape, orientation, etc) ? For any point $z$ on a segment between two points $z_1$ and $z_2$ in $\mathbb{R}^k$, what happens to $G(z)$ ? What if the noise vector (also in $\mathbb{R}^k$) used to train the generative model is sparse ?

## 4.1 Tensorflow tutorial and first modifications

Tensorflow provides a clear and simple implementation of GANs [5], in a notebook accessible online through Google Colab. This provided us with a template from which we could run our own experiments.

Both the generator and the discriminator are defined and trained on the MNIST dataset of handwritten digits from 0 to 9. Some parts of this code are of particular interest to us. The generative model is defined as follows :

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False,
     input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256)

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),
     padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),
```

```
     padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2),
     padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

As we can see, the initial layer is a densely-connected layer which takes a 100-dimensional seed (random noise) as input. This seed is then upsampled in a series of transposed convolutional layers, until we obtain a 28x28 output.

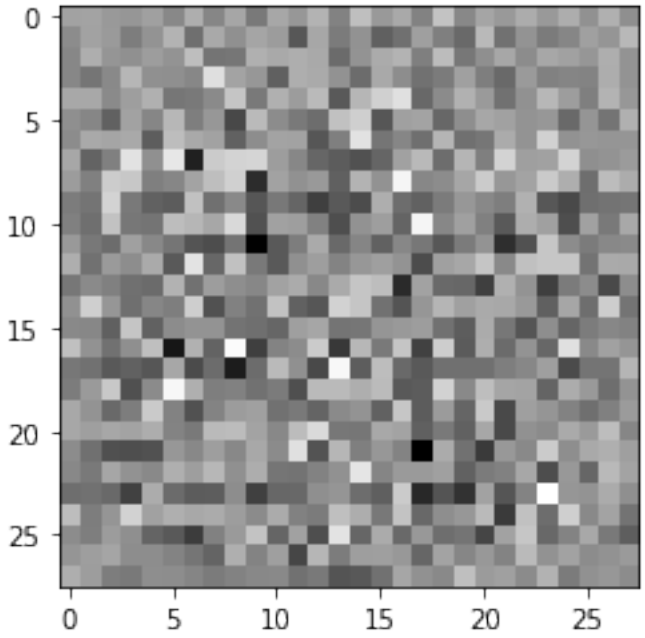
In the following lines of code, the untrained generator takes as input some random noise to create an image :

```
generator = make_generator_model()

noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)

plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

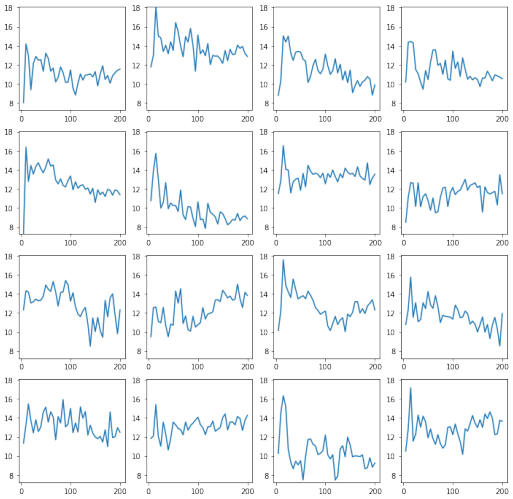
Figure 1: Image created by an untrained generator



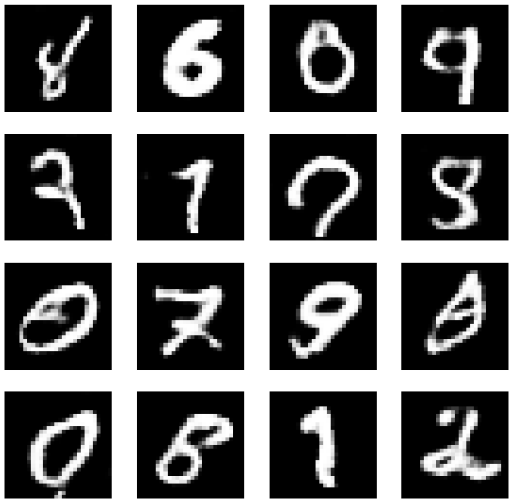

The generated image obviously does not look like a handwritten digit. However, after 50 epochs, the generator is able to create images closely resembling the MNIST digits. (include image from gif)

Consider the test set for the MNIST dataset, which we will denote by $S$. One way to measure the performance of our generator could be to take the distance between each of the generated images and the set $S$, and see the evolution throughout the training process. In other words, for the $i$th generated image, we would compute $d_i = \min_S \|G(z_i) - s\|$ every 5 epochs (as opposed to every epoch, for computational reasons), where each $s \in S$ is an image in the test set.

We set the number of generated images to 16, and the numbers of epochs to 200. As shown in the figures below, for most generated images, the distance $d_i$ tends to decrease but not smoothly. However, we can also see that for some of the images, the distance $d_i$ at epoch 200 is not that much lower than its value at epoch 1.



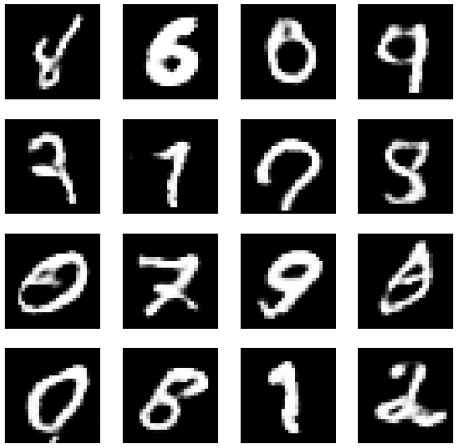
(a) Convergence of distance of the generated images from the test set



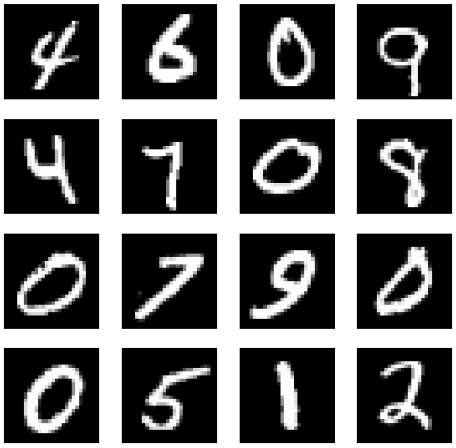
(b) Generated images at epoch 200

For a few examples, the distance $d_i$ does show a downward trend over 200 epochs. But as we can see from some of the subplots above, that trend is not clear for some of the examples. Some of them even seem to have a higher distance $d_i$ from the test set at epoch 200 than at the beginning of the training process. One thing that seems to be consistent, however, is that $d_i$ tends to decrease more clearly for "simpler looking" digits, like "1".

As an additional illustration, the figures below show each $G(z_i)$ (each generated image) at epoch 200, and the corresponding $s \in S$ which minimizes $\|G(z_i) - s\|$ i.e. the closest image in the test set to $G(z_i)$.



(a) Generated images at epoch 200



(b) Closest image in the test set from each generated image

Obviously, this method for calculating the generator's efficiency is not perfect, as it might give a false sense of which digit each $G(z_i)$ represents, as we can see in the figures above. Given the way $d_i$ is computed, the distance between two pictures that *should* represent the same digit, might be larger than intended due to a difference in their inclination, for example. From our numerous experiments, we would sometimes find that a generated picture that resembled a "1" or an "8" would be closest to a "7" or a "9" in the test set. It should be noted, however, that most of the time we got an accurate / a more realistic "mapping" from the generated images to the test set.

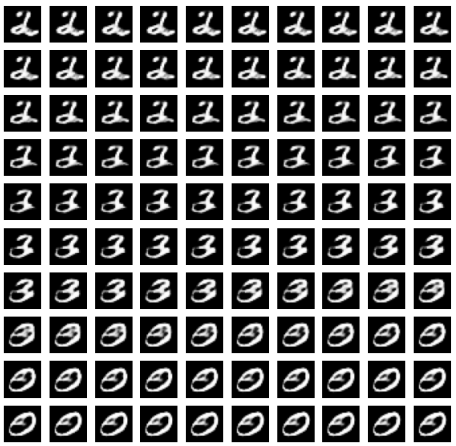
### 4.2 What happens on a segment between two points in $\mathbb{R}^k$

In our context, we know that $G$ takes any vector $z \in \mathbb{R}^k$ and maps it into a subspace of $\mathbb{R}^n$ representing a set of possible handwritten digits from 0 to 9. Consider two random points $z_1$ and $z_2$ in the domain $\mathbb{R}^k$ of the generative model $G$. Let $z$ be a convex combination of $z_1$ and $z_2$, i.e. any point lying on the segment between $z_1$ and $z_2$. Namely, $z = (1-\lambda)z_1 + \lambda z_2$ where $0 \leq \lambda \leq 1$.

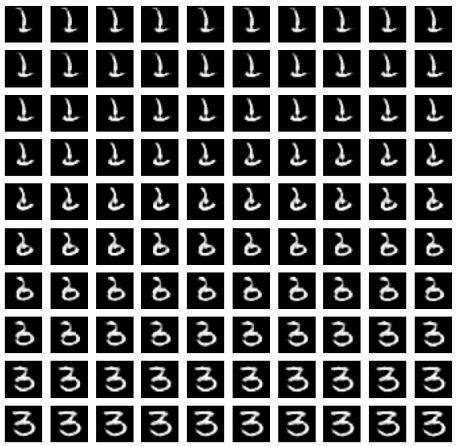
Given that $G(z_1)$ represents a handwritten digit, and $G(z_2)$ represents another handwritten digit (or the same digit, but with a slightly different shape/orientation), it would be interesting to understand, or at least "see", the behavior of $G(z)$ as $z$ goes from one end of the segment to the other.

We made the experiment taking two random vectors from the seed [5]. We defined $z$ as a convex combination of those two vectors, with a variable $\lambda$ starting at $\lambda = 0$ and increasing by a small step to arrive at $\lambda = 1$. We took a step of 0.01, thus creating a 10x10

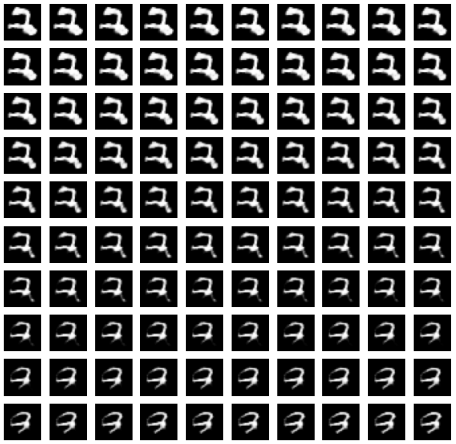
grid of 100 subplots, showing the evolution of $G(z)$ as $z$ goes from one end of the segment to the other.



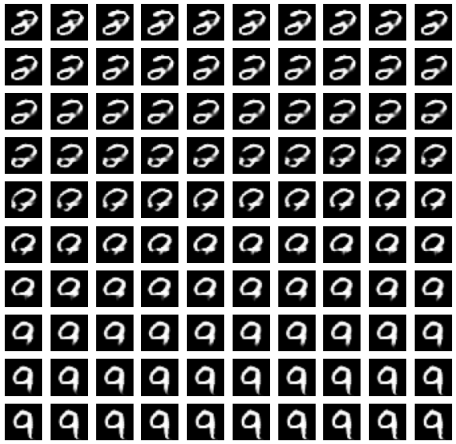
(a) Example 1



(b) Example 2



(c) Example 3



(d) Example 4

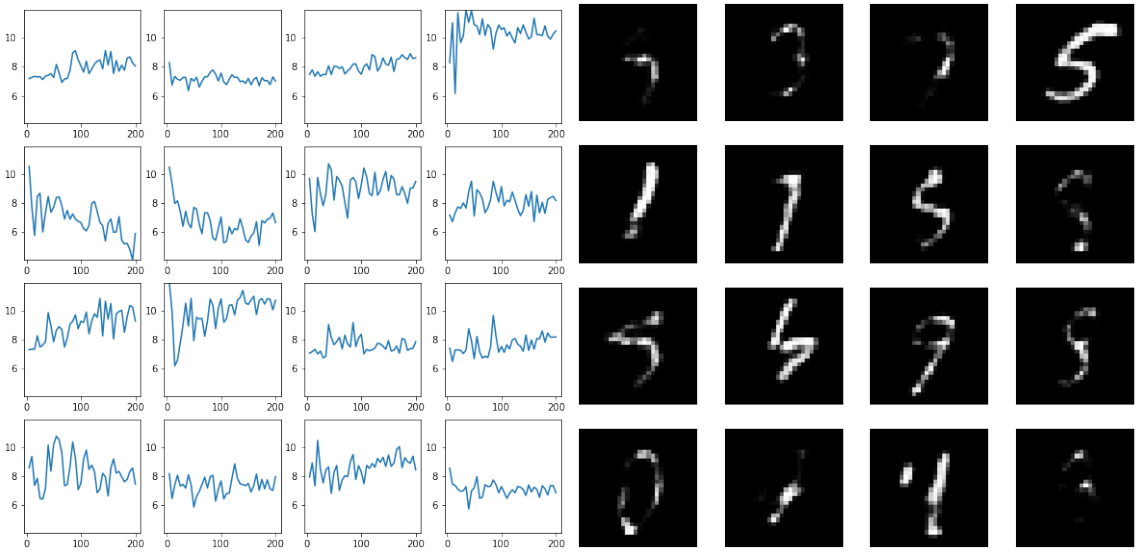
Between $z_1$ and $z_2$, the generator behaves in a way one would expect. As $z$ approaches $z_2$, $G(z)$ gradually looks more and more like $G(z_2)$.

There are several other ways one could further investigate how $G$ behaves on a particular subspace of $\mathbb{R}^k$. Instead of considering a line segment between two points $z_1$ and $z_2$, we could find a more chaotic/elaborate path defined by more than two points, or we could consider a subset of a hyperplane defined by a linear combination of some points

$z_1, z_2, ..., z_n \in \mathbb{R}^k$. Obviously, this would be more challenging to implement and it would be harder to clearly visualize how $G$ behaves, but it could potentially yield some interesting results.

### 4.3 Generative model trained with a sparse vector

We also trained another generator with a sparse seed. For simplicity, in this subsection, the generator trained in our original experiments will be called Generator A, and the generator trained with a sparse seed will be called Generator B. The parameters are the same as in the original implementation : the number of epochs is set to 200, the number of examples to generate is set to 16, and the noise dimension is set to 100. We also replicated the same plots/experiments, i.e. the convergence plot for the 16 generated examples, the subplots showing the $G(z)$ for a convex combination $z$ of two vectors $z_1, z_2 \in \mathbb{R}^k$, etc.



(a) Convergence of distance of the generated images from the test set

(b) Generated images at epoch 200

The most noticeable difference compared to Generator A, is that most of the digits are not clearly visible. Furthermore, if one accepts the distance $d_i$ from the test set to be a measure of how well the generator performs, then by this measure Generator B overall seems to perform as well (if not better) than Generator A, because in most of the examples from Generator B we can see that (at epoch 200) $d_i$ is between 6 and 8, whereas in the examples from Generator A, $d_i$ is between 8 and 10. This seems counterintuitive, as the digits generated by Generator A "feel" more "true digits". However, as mentioned earlier, this method for measuring the performance of a generator is not perfect, and this is another

one of its limits.

Another difference is that, even though the nonzero entries of the sparse seed are exactly the same as the equivalent entries of the seed used to train Generator A, some of the digits generated by Generator B appear to have changed.

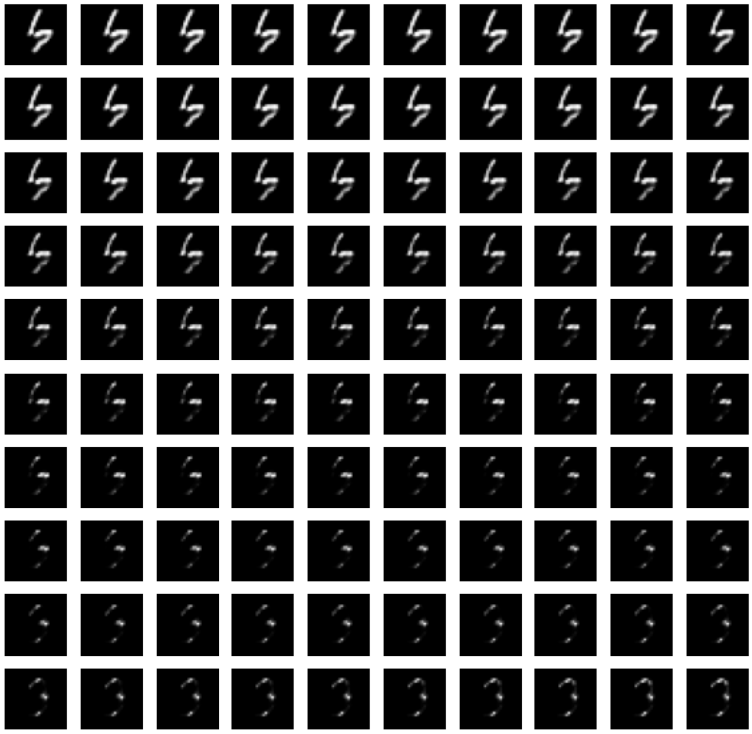


Figure 6: example

The figure above shows the behavior of Generator B on a segment between two pints $z_1, z_2 \in \mathbb{R}^k$. We observe a similar behavior as in the examples from Generator A. The only noticeable difference is that the generated digits are not clearly visible, as already mentioned earlier.

## 5 A brief overview of the theoretical results

The Set-Restricted Eigenvalue Condition (S-REC), as well as a few other conditions, imply that the reconstruction error $\|\hat{x} - x^*\|$ is bounded above with high probability, which in turn

means that a robust recovery of $x^*$ is guaranteed (for a certain number of measurements $m$).

**Definition 1** *For $x \to \infty$, $f(x) = \Omega(g(x))$ iff $\limsup_{x\to\infty} |\frac{f(x)}{g(x)}| > 0$.*
*$f(n) = \Omega(g(n))$ iff there exists a constant $C > 0$ for which for all $n_0$, there is an $n > n_0$ where $|f(n)| \geq Cg(n)$*

**Definition 2 (Gaussian matrices)** *$A \in \mathbb{R}^{m\times n}$ is a Gaussian matrix if the entries of $A$ are independent identically distributed Gaussian random variables, with expectation 0 and variance 1/m. [3]*

**Definition 3 (Random Partial Fourier matrices)** *A random partial Fourier matrix $A \in \mathbb{C}^{m\times n}$ is derived from the discrete Fourier matrix $F \in \mathbb{C}^{n\times n}$ with entries*

$$F_{j,k} = \frac{1}{\sqrt{n}} e^{2\pi jk/n},$$

*by selecting m rows uniformly at random among all n rows. [3]*

**Definition 4 (Set-Restricted Eigenvalue Condition)** *A matrix $A \in \mathbb{R}^{m\times n}$ satisfies S-REC(S, $\gamma$) if for every $x_1, x_2 \in S$, $\|A(x_1 - x_2)\| \leq \gamma\|x_1 - x_2\|$.*

### 5.1 Case when $G$ is L-Lipschitz

From Lemma 4.1 [1], we can see that for $m = \Omega(\frac{k}{\alpha^2} \log \frac{Lr}{\delta})$ ($m$ being the number of measurements), the Gaussian matrix $A \in \mathbb{R}^{m\times n}$ satisfies the S-REC. We can also see that $m$ depends on the dimension of the domain of $G$, the radius of an $L_2$-norm ball in that domain (or any compact set), and the constant L which is related to the L-Lipschitz condition for the generative model.

### 5.2 Case when $G$ is a d-layer neural network using ReLu activations

Similarly, from Lemma 4.2 [1] we see that for $\alpha < 1$ and $m = \Omega(\frac{1}{\alpha^2} kd \log c)$, the Gaussian matrix $A \in \mathbb{R}^{m\times n}$ satisfies the S-REC. Here, the number of measurements $m$ depends on the dimension of the domain $\mathbb{R}^k$ of $G$, the number of layers in the neural network and the number of nodes per layer.

### 5.3 An overview of the lemmas

The result from Lemma 4.3 [1] implies that an estimate of the real signal $x^*$, sapproximately minimizing $\|y - Ax\|$, is close to the real signal $x^*$ with high probability. More formally, the lemma states that for $A \in \mathbb{R}^{m\times n}$ satisfying S-REC$(S, \gamma, \delta)$ with probability $1 - p$

and $\|Ax\| \leq 2\|x\| \forall x \in \mathbb{R}^n$ with probability $1-p$, and for $\hat{x}$ such that $\|y - A\hat{x}\| \leq \min_{x \in S}\|y - Ax\| + \epsilon$, we have

$$\|\hat{x} - x^*\| \leq (1 + \frac{4}{\gamma}) \min_{x \in S}\|x^* - x\| + \frac{1}{\gamma}(2\|e\| + \epsilon + \delta)$$

with probability $1-2p$. In other words, the reconstructed signal $\hat{x}$ is close to the real signal, up to the representation error, the noise level and the optimization error. The lecture notes from [6] provide a clear and concise proof of that lemma. The main steps of that proof are shown below :

First, we define $\bar{x} = \arg\min_{x \in S}\|x^* - x\|$. Recall that we want an upper bound for $\|\hat{x} - x^*\|$. By the Triangle inequality, $\|\hat{x} - x^*\| \leq \|x^* - \bar{x}\| + \|\bar{x} - \hat{x}\|$. Since $A$ satisfies $S - REC(S, \gamma)$ and $\bar{x}, \hat{x} \in S$, we have $\|\bar{x} - \hat{x}\| \leq \frac{\|A(\bar{x} - \hat{x})\|}{\gamma}$
$\implies \|\hat{x} - x^*\| \leq \|x^* - \bar{x}\| + \frac{\|A(\bar{x} - \hat{x})\|}{\gamma}$.
By the Triangle inequality, $\|A(\bar{x} - \hat{x})\| \leq \|A\bar{x} - y\| + \|A\hat{x} - y\|$. Thus, $\|\hat{x} - x^*\| \leq \|x^* - \bar{x}\| + \frac{\|A\bar{x} - y\| + \|A\hat{x} - y\|}{\gamma}$.
Since $\hat{x}$ is an approximate minimizer of $x^*$, we have $\|A\hat{x} - y\| \leq \|A\bar{x} - y\| + \epsilon$
$\implies \|\hat{x} - x^*\| \leq \|x^* - \bar{x}\| + \frac{2\|A\bar{x} - y\| + \epsilon}{\gamma}$
Once again, by the Triangle inequality, $\|A\bar{x} - y\| \leq \|A\bar{x} - Ax^*\| + \|Ax^* - y\|$
$\implies \|\hat{x} - x^*\| \leq \|x^* - \bar{x}\| + \frac{2\|A\bar{x} - Ax^*\| + 2\|Ax^* - y\| + \epsilon}{\gamma}$
$\implies \|\hat{x} - x^*\| \leq \|x^* - \bar{x}\| + \frac{2\|A\bar{x} - Ax^*\| + 2\|e\| + \epsilon}{\gamma}$.
Finally, since $\forall x \in S, \|Ax\| \leq 2\|x\|$, we have $\|A(\bar{x} - x^*)\| \leq 2\|\bar{x} - x^*\|$
$\implies \|\hat{x} - x^*\| \leq \|x^* - \bar{x}\| + \frac{4\|\bar{x} - x^*\| + 2\|e\| + \epsilon}{\gamma}$
$\implies \|\hat{x} - x^*\| \leq (1 + \frac{4}{\gamma})\|x^* - \bar{x}\| + \frac{2\|e\| + \epsilon}{\gamma}$, which finishes the proof.

All there is left to show is that if a matrix $A$ is a random matrix, then it satisfies the S-REC. More formally :
For $G : \mathbb{R}^k \to \mathbb{R}^n$ with $d$ layers (linear transformation followed by a ReLU) and at most $n$ nodes per layer, if $m = \Omega(\frac{kd \log n}{\alpha^2})$ then $A \in \mathbb{R}^{m \times n} \sim N(0, 1)$ satisfies $S - REC(G(\mathbb{R}^k), 1 - \alpha)$ with probability $1 - e^{(-\Omega(\alpha^2 m))}$. A proof is also provided in [6]. The main steps of this proof are detailed below :

The proof uses the two following theorems.

**Theorem 1** *If $A \in \mathbb{R}^{m \times n} \sim N(0, 1)$, then $\forall t \geq 0, \sqrt{m} - \sqrt{k} - t \leq \sigma_{min}(A) \leq \sigma_{max}(A) \leq \sqrt{m} + \sqrt{k} + t$ with probability $1 - 2e^{-\frac{t^2}{2}}$*

**Theorem 2** *If $\mathbb{R}^k$ is partitioned by $C$ hyperplanes, then the number of partition pieces is $O(C^k)$.*

The first layer of $G$ has at most $n^k$ different linear pieces. Since $G$ is a $d$-layer neural network, $G$ has at most $n^{kd}$ different linear pieces.
Then, by Theorem 2, $G(\mathbb{R}^k)$ lives in a union of $n^{kd}$ subspaces of dimension $k$

$\implies \{x_1 - x_2 \mid x_1, x_2 \in G(\mathbb{R}^k)\}$ lives in a union of $n^{2kd}$ subspaces of dimension (at most) $2k$.

$A$ satisfies the the $S - REC$ on each of the subspaces with probability $1 - e^{\Omega(\alpha^2 m)}$ if $m = \Omega(\frac{k}{\alpha^2})$. Then, the probability $p$ that $A$ satisfies the $S - REC$ on **all** the subspaces is

$p \geq 1 - n^{2kd} e^{-\Omega(\alpha^2 m)} \implies p \geq 1 - e^{-\Omega(\alpha^2 m)}$ if $m = \Omega(\frac{kd \log n}{\alpha^2})$, which is what we wanted to show.

Hence, from these two lemmas, we can conclude that for $x \in \mathbb{R}^n$ and for a random matrix $A$ satisfying $\|Ax\| \leq 2\|x\|$ and $\|y - A\hat{x}\| \leq \min_{x \in S} \|y - Ax\| + \epsilon$, $\|\hat{x} - x^*\|$ is bounded above, thus the signal $x^*$ can be reconstructed up to a few sources of error.

# 6 Experiment using generative models for compressed sensing

In this first experiment [2], we compare the algorithm with Lasso by reconstructing 10 images from the MNIST dataset. As described in section 5.1. of the paper [1], the generative model used is a VAE, where the input is a vectorized binary image of input dimension $n = 784$ (since each image is of size $28 \times 28$), and we set the size of the representation space $k = 20$. Lasso is also used to reconstruct the 10 input images, in order to compare its performance with the algorithm for the same number of measurements $m = 100$. The shrinkage parameter is set to be 0.1 for all the experiments. The following figure shows the values for all the input parameters described in section 2, as well as some parameters more relevant to the implementation on a computer (ex : the paths to access the dataset and the pretrained model).

Figure 7: Input parameters

```
batch_size = 10
checkpoint_iter = 1
dataset = mnist
decay_lr = False
dloss1_weight = 0.0
dloss2_weight = 0.0
gif = False
gif_dir =
gif_iter = 1
image_matrix = 1
image_shape = (28, 28, 1)
inpaint_size = 1
input_path_pattern = ./data/celebAtest/*.jpg
input_type = full-input
lasso_solver = sklearn
learning_rate = 0.01
lmbd = 0.1
max_update_iter = 1000
measurement_type = gaussian
mloss1_weight = 0.0
mloss2_weight = 1.0
model_types = ['lasso', 'vae']
momentum = 0.9
n_input = 784
noise_std = 0.1
not_lazy = True
num_input_images = 10
num_measurements = 100
num_random_restarts = 10
optimizer_type = adam
pretrained_model_dir = ./mnist_vae/models/mnist-vae/
print_stats = True
save_images = False
save_stats = False
sparsity = 1
superres_factor = 2
zprior_weight = 0.1
```

As expected [1], for 100 measurements, the reconstruction error (per pixel) is around 0.09 for Lasso, and approximately 0.01 for the algorithm (using VAE as the generative model). The figure below shows the measurement loss and L2 loss outputs displayed when the algorithm stops.

Figure 8: Reconstruction errors outputs

```
lasso
mean measurement loss = 19.0091068152
mean l2 loss = 0.0860554898165
vae
mean measurement loss = 5.5532747702
mean l2 loss = 0.00891904100937
```

# 7 Conclusion

As mentioned in the introduction, the initial goal of this summer research project was to better understand the theory behind how generative models could be used in the context of compressed sensing, and implement those theoretical results numerically. Our main interrogations slowly shifted more towards generative models. More particularly, we wanted to understand more about the structure of the domain of a generative model.

Thus, the numerical side of the project naturally became more prevalent. Most of the work was done using Python, trying to investigate the behavior of generative models by finding patterns between the latent space and the image space. The main question we attempted to answer was : can we reduce the domain dimension to a few key components that capture most of the important features in the image space ? Of course, this question along with several others, still remain open.

A potential route to get closer to the answer could be to use a different kind of generative model (for example, VAEs as their different architecture could be more appropriate for the task), or to find a better way to visualize the patterns between the latent space and the image space, allowing us to draw better conclusions.

## References


[1] Ashish Bora et al. "Compressed sensing using generative models". In: *arXiv preprint arXiv:1703.03208* (2017).

[2] Bora, Ashish. *GitHub repository for Compressed Sensing using Generative Models*. `https://github.com/AshishBora/csgm`. 2017.

[3] Massimo Fornasier and Holger Rauhut. "Compressive Sensing." In: *Handbook of mathematical methods in imaging* 1 (2015), pp. 187–229.

[4] Ian Goodfellow et al. "Generative adversarial nets". In: *Advances in neural information processing systems*. 2014, pp. 2672–2680.

[5] Google. *Deep Convolutional Generative Adversarial Network - Tutorial*. `https://www.tensorflow.org/tutorials/generative/dcgan`.

[6] Hand, Paul. *Notes on Compressed Sensing using Generative Models*. `https://khoury.northeastern.edu/home/hand/index.html`.

[7] Diederik P Kingma and Max Welling. "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114* (2013).

[8] Alec Radford, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks". In: *arXiv preprint arXiv:1511.06434* (2015).