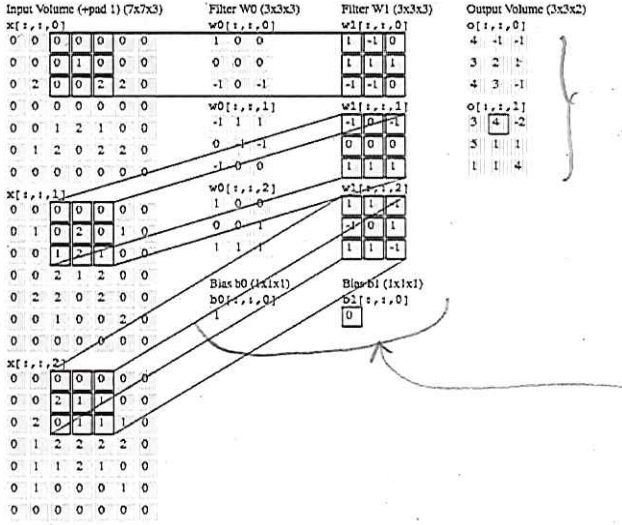

Figure 2.10 – Example of a convolution layer in a 3D input with parameters Initial input volume 3×3, $K=2$, $F=3$, $S=2$ and $P=0$

$D_2=2$ $F_H=3, F_W=3$ (kernels are of size 3×3, there are $D_1\times D_2$ $3\times2=6$ kernels, output is 2 maps of size $W_2 = \frac{7-3+0+2}{2}=3$

$D_1=3$
$W_1=7$
$H_1=7$

$H_2 = \frac{7-3+0+2}{2}=3$

- **Fully connected layer** is completely connected to all neurons in the previous layer, as in a regular neural network. It also makes networks grow lager and when used in CNN's they are commonly the last layer computing the final result for a type of classification. As with pooling layers we do not make use of it.

- **Activation function for CNN's** normally refers to a linear rectifier function ReLu 2.5. It is explicitly given as another layer in the network and allows for efficient gradient propagation. It is very popular because not only is its computation easier than other activation functions like sigmoid 2.3 or tahnh 2.4 but, it also decreases the probability for observing vanishing gradients.

3 Implementation methodology

As stated before we want to evaluate the performance of CNN's to reconstruct images from compressed measurements. In this chapter we will describe the tools and software frameworks used in order to build the set-up for our experiments. First, we introduce some of the most widely used libraries to build and train neural networks and the specific software tools for this thesis. Second, we briefly describe Graphics Processing Units (GPU's) and give reasons why they are an important tool when working with deep learning. Third, we will describe the datasets for training the networks and briefly justify its utilization. Then, we explain how the data is pre and post-processed. This is important since it allows to efficiently use the hardware resources and makes the training process numerically stable. Finally, we propose CNN architectures for the reconstruction process.

3.1 Theano

Theano is a library written in python that permits to define, optimize and compute mathematical expressions that deal with high-dimensional arrays in an efficient way. It is widly used among the deep learning community because it is optmized to make use of GPU routines that make training faster, efficient symbolic differentiation, stability optimizations and therefore making it realiable since it is constantly tested and debugged [49]. Nevertheless, Theano is not intended to be used only for neural natwork applications. Therefore, it is necessary to write routines or API's that specially handle the difficulties encountered for deep learning. Such applications are normally an extra abstraction layer that sits on top of Theano's implementation and specially allow to build and train neural networks. Examples publically avaible are Lasagne [22], Blocks [50] and Keras [19]. Other common frameworks in deep learning are Google's Tensorflow [4] that is written in C++, Torch [20] written in Lua and Caffe [36] also written in C++. Most of them offer the same characteristics and mostly differ in the language they are written in, speed and target application, for example Caffe is not suitable for audio and text applciations.

Figure 3.1 – NVIDIA graphics card GeForce GTX Titan X used in our experiments.

3.1.1 Sdeepy

Due to the fact that the previously mentioned implementations are constantly updated and changing, Sony decided to develop its own library to develop its projects. Sdeepy is an in-house Sony's deep learning library implementation that makes use of Theano and incorporates routines commonly used as building-blocks for training and testing neural networks. It has the advantage that many new features can be added at any time they become available while maintaining the stability. Furthermore, it is easier to adapt and modify according to one's needs. Because of all that, we will use Sdeepy for this thesis but all implementations might be easily translated into any available framework.

3.2 Graphics Procesing Unit (GPU)

A Graphics Processing Unit (GPU) is a chip architecture mainly design for imaging and gaming. It differs from general-purpose CPU's in that they can handle larger amounts of data efficiently and faster in a parallel way, making them much more suitable for deep learning applications. It has extended its usage into machine learning area because companies like NVIDIA have develop GPU's specially optimized for neural networks. Not only that, but they have also written routines (NVIDIA Deep Learning SDK) that improve and speed up common operations like convolutions, better memory management and friendlier API's for easy learning. By distributing the workload of the training process the convergence happens much faster than by using conventional CPU's. See Figure 3.1 and Table 3.1 .

Table 3.1 – Technical details of the GPU

Architecture	Maxwell
CUDA cores	3072
Base clock speed	1000MHz
Boost clock speed	1075MHz
Memory type	GDDR5
Memory	12GB
Memory Bandwidth	7Gbps
Memory interface width	384-bit

Table 3.2 – Original features of each dataset.

Dataset name	Number of images	training	testing	Original image size	Grayscale
BioID	1521	1371	150	384x286	Yes
LFW	13233	11933	1300	250x250	No
LabelMe	50000	40000	10000	256x256	No

3.3 Datasets

It is also important to mention that along with better hardware and more ingenious algorithms, deep learning has achieved major success because in recent years more and more data is becoming available through the internet. For most of machine learning methods data is the most important asset to obtain successful results. In fact, choosing the right dataset is the first step in the deep learning work-flow cycle and it is, in general, much more important than the chosen learning algorithm itself. Since our goal is to reconstruct images, we make use of three different image datasets: BioID [29], Label faces in the wild (LFW) [33] and LabelMe [48]. The first two datasets served as an initial baseline for measuring the plausibility of our approach and the last one allows for a more robust justification as explained later. All of the datasets are known in the community for being applied to different computer vision tasks like face detection, emotion recognition and image classification. Here, we propose its usage for image reconstruction from compressed samples.

All datasets differ in terms of image size, color model (RGB or Grayscale) and file format (jpg, png, bmp, etc.). In order to facilitate implementation, all datasets had to be transformed into a more generic form that could meet our constrains. Namely, it is more convenient to work with grayscale images and once promising results are obtained, it can be easily extended to RGB images or even video. For that reason, LFW and LabelMe datasets were converted into grayscale images using rgb2gray MATLAB function. Furtheremore, having an image size that was divisible by 16 was an initial requirement and so each image was resize to 256x256 using imresize. The actual data being used is described in table 3.3.

Finally, in order to evaluate the performance of our trained network we introduce a set of 18 images. The images, we believe, are the most popular ones in the image processing community and serve as a baseline for many reconstruction algorithms and image processing tasks. They also help compare our results against the ones reported by other researchers. The evaluation set is shown in figure 3.2

Table 3.3 – Transformed features of each dataset.

Dataset name	Number of images	training	testing	Original image size	Grayscale
BioID	1521	1371	150	256x256	Yes
LFW	13233	11933	1300	256x256	Yes
LabelMe	50000	40000	10000	256x256	Yes

Figure 3.2 – From Top Left to Bottom Right: 'fingerprint', 'couple', 'boats', 'flintstones', 'peppers', 'woman', 'butterfly', 'bridge', 'houses', 'house', 'mandril', 'parrot', 'montage', 'foreman', 'man', 'barbara', 'lena', 'cameraman'

3.3.1 Training set

Consist of the largest set of images and is the one used for estimating optimum values of ω parameters during training. Because one can argue that fitting ω values to a particular dataset may lead to overfitting and therefore the network could perform poorly with unseen data, we have to test our model with different data. Column testing in table 3.3 refers to the number of training images for each dataset.

3.3.2 Testing set

Is a smaller part of the dataset that is not used for training. Since, the training images also come from the same distribution, correlations about the performance in in this data help decide whether the CNN is overfitting the data or how good the generalization is. Column testing in table 3.3 gives the number of testing images for each dataset.

3.3.3 Validation set

This a set of images we introduce in order to further test out CNN. It's not used or linked in anyway to the training process and serves more as a proof of the CNN's performance. It comes from the tradition of use certain famous images for showing results of an images processing algorithm.

3.4 Preprocessing of the images

Even though grayscale images can be easily interpreted as a $2D$ matrix, CNN's expect as an input a $3D$ tensor, chapter 2.2.2, complying with the convention depth, height and width. Consequently, extra preprocessing is carried out to generate the data set that is used as the final input of the network for training. Moreover, we make use of a technique called Block Compressed Sensing (BCS). BCS allows to divide the image into small blocks of size $B \times B$ with $B = 16$ and then compress them by using the same sensing matrix Φ. It has the main advantage

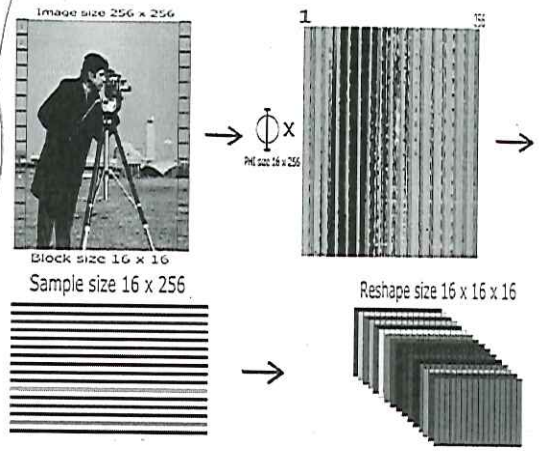

Figure 3.3 – Example of image preprocessing with $B = 16$, $\mathcal{C} = 16$ and $N = 256$.

of having a small sensing matrix that can be easily extended to process high-dimensional images and making the overall process faster [30, 28]. Another important parameter is the ~~subrate or subsampling size~~ $\mathcal{C}$. That is, how much information we would like to sense. For most of our experiments ~~the we choose~~ $\mathcal{C} = 16$, we also show results with different $\mathcal{C}$ values, ~~yielding a subrate of 1/16~~. For an image of size 256×256 that means there are 65536 possible samples (original pixel values), by using a ~~subrate of 1/16~~ we will only be taking 4096 measurements from which we will reconstruct the image.

To demonstrate the process, consider an image randomly taken from one of the datasets. It has a size of $N \times N$ with $N = 256$. First, it is converted into blocks of size B by using MALAB command im2col. Second, ~~the image~~ is multiplied with the sensing matrix Φ to obtain the measurements. ~~Since the size of Φ is $\mathcal{C} \times B^2$ and the image size is $B^2 \times N$, the product will have size $\mathcal{C} \times N$. At last, the measurements are reshaped into a 3D tensor of size $\mathcal{C} \times N \times N$.~~ A pictorial representation of the process can be seen in Figure 3.3. After preprocessing each image individually, the data used for training and testing is a 4D tensor and each dimension is interpreted as follow: (*Number of images, size of* $\mathcal{C}$ *size of* N/B *size of* N/B)

3.5 Network architectures for CS recovery

The architecture of the network refers to the order in which different layers are arranged and interconnected to achieve a specific purpose. Throughout the development of the thesis

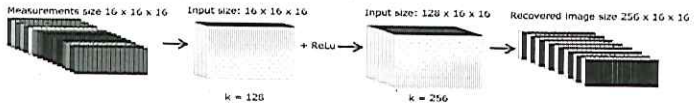

Figure 3.4 – Depiction of the alpha CNN architecture.

different architectures were heuristically tested and in the following we present the ones that produced better performance.

Alpha CNN network

This the first network we propose to recover images from compressed samples is in Figure 3.4. It is composed of an input layer and an output layer. The input layer receives measurements of size (*size of* $\mathcal{C}$ *size of* N/B *size of* N/B with a kernel size $k = 128$ and followed by ReLu. The output layer receives as input the output of the previous layer, that is (*size of k in previous layer, size of* N/B *size of* N/B and a kernel $k = 256$. The output layer is designed so that it matches the size of the original image.

Beta CNN network

During the development of this thesis several attemps for reconstructing images using deep learning were published [39, 44, 34, 35, 5]. Interestingly, Kulkarni *et al.* [39] also addressed the problem of CS recovery using a CNN. The other approaches are very similar because they are based on an auto-encoder. Even more, they do not compressed the images using iid Gaussian matrices. Rather, they learn the sensing matrix Φ in the first layer. Other interesting findings were made by Kulkarni *et al.* [39]. They introduced a new parameters that controls the redundancy of the network. Redundancy refers to how large the network will be in relation with the size of the original image. For example, a redundancy factor of 8, would create a reconstruction layer of size *redundancy* $\times N$. The value 8 turned out to yield better reconstruction performance according to their tests.

Based on the previous concepts and proved capabilities of the alpha network we proposed another network sketched in Figure 3.5. Unlike alpha network, beta network does not receive compressed measurements as input. Rather, the original image interpreted as blocks is reshaped into a 3D tensor. Doing that, allows the network to learn the sensing matrix, in fact the first layer is interpreted as the compression step. The second layer and third layers are called reconstructions layers. The output layer reshapes the data back to the original image size and is the last step in the recovery process.

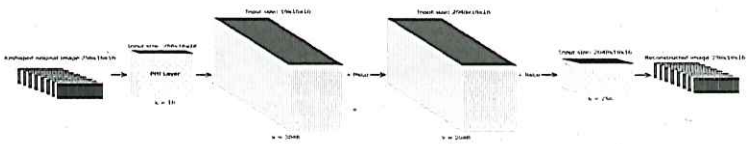

Figure 3.5 – Depiction of the beta CNN architecture. Notice the first layer is learning the sensing matrix Φ.

3.6 Training CNN's

Training is the process of learning the values of the parameters of the networks. During this step one is concerned with choosing an appropriate weight initialization, loss function, parameter update rule and batch size.

3.6.1 Weight initialization

Choosing the right initial values of the network is paramount to secure a sucessful convergence. Furthermore, it can avoid problems like vanishing gradients or ending up being stuck in local minima. Using independent Gaussian random numbers is a very common practice for simple networks but, as complexity increases smarter ways had been investigated [31]. Glorot and Bengio empirically validated a method to initialize the weights called normalized initialization

$$W \sim U[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}] \tag{3.1}$$

were n_j is the layer size of layer j. In this thesis we follow the same approach.

3.6.2 Loss function

Since we need a measure how good the reconstruction from compressed measurements is, we need to define a way to compare the difference (error) against the original image. Not only that, the loss function also serves as an objective that we would like to minimize as much as possible. The choice of the loss functions depends on the type of problem that one is facing, regression or classification, in order to achieve better results. We have chosen to use the mean square error as the loss function (MSE) because it is highly related to an evaluation metric (PSNR) that we will explain later. The MSE is mathematically defined as

$$L(w) = \frac{1}{N}\sum_{n=1}^{N} \|f(x_n, w) - y_n\|_2^2 \tag{3.2}$$

were N is the number of images in the dataset, $\hat{x} = f(x_n, w)$ represents the nonlinear mapping computed by the CNN and y_n is the original image. Using backpropagation, we train the CNN by minimizing the loss function defined in 3.2.

3.6.3 Parameter update rule

The easiest way to update the weights ω is by following the negative direction of the loss function using SGD as we explanied previously 2.2.3. As with initialization methods, the algorithms to find a minimum abound. For our training, we use Adam (Adaptive moment estimation) [37] as it converges faster compared to other algorithms.

It works by storing the weighted decay of the previous square gradient v_t and previous gradient m_t. Subsequently, v_t and m_t are corrected in order to eliminate initial bias. Those estimates are used along with SGD as a way to adapt learning rates for parameter ω. The full algorithm as we use it goes

Algorithm 2 Adam update

Require: $\alpha = 0.0005$: Stepsize, $c = 10^{-8}$
Require: $\beta_1 = 0.9$, $\beta_2 = 0.999$: Exponential decay rats for moment stimates
Require: $L(\omega)$: Loss function
Require: ω_0: Initial parameter values
$m_0 \leftarrow 0$ (Initialize 1^{st} moment vector)
$v_0 \leftarrow 0$ (Initialize 2^{nd} moment vector)
$t \leftarrow 0$ (Initialize timestep)
while ω_t not converged **do**
  $t \leftarrow t+1$
  $g_t \leftarrow \nabla_\omega L_t(\omega_{t-1})$ (Gradient of loss function at timestep t)
  $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1)\cdot g_t$ (Update biased first moment stimate)
  $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2)\cdot g_t^2$ (Update biased second moment stimate)
  $\hat{m}_t \leftarrow \frac{m_t}{(1-\beta_1^t)}$ (Compute bias-corrected first moment estimate)
  $\hat{v}_t \leftarrow \frac{m_t}{(1-\beta_2^t)}$ (Compute bias-corrected second moment estimate)
  $\omega_t \leftarrow \omega_{t-1} - \alpha \cdot \frac{\hat{m}_t}{(\sqrt{\hat{v}_t}+c)}$ (Update parameters)
end while
return ω_t (Resulting parameters)

3.6.4 Batch size training

For training our CNN's and results presented later we have used a batch size of 20.

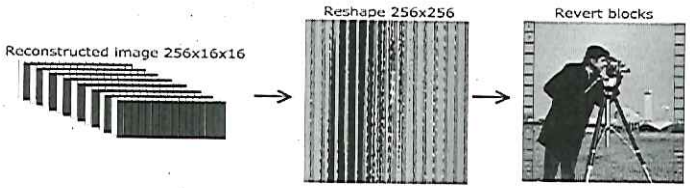

Figure 3.6 – Postprocessing illustration to visualize the reconstructed image.

3.7 Postprocessing of reconstructed images

As it can be inferred from Figure 3.5 that the output of the CNN is not an image. Rather, it is a 3D tensor from which, with minor extra postprocessing, we recover the image. This process aims to invert the prepocessing step explained in section 3.4.

First the image is reshaped back to ist original size $N \times N$. Then using col2im MATLAB commnad get the reconstructed image ready to be visualized. Image 3.6 shows the procedure. Optionally, we also make use of the very common image denoiser BM3D [21]. That is because we would like to get rid of the block ~~nature~~ artifacts previously explained.

3.8 Evaluation metrics

Whenever images are undergoing any type of processing, they are bound to suffer a deprecation of the visual quality. Namely, for applications that are ultimately perceived by human eye, there is a need to precisely evaluate the degradation rate (noise) in the image that a certain algorithm adds. Even though agreeing on a metric that is capable of fairly measureing the visual impact is somewhat subjective, there are two methods widely used for that purpose Peak signal-noise-to ratio (PSNR) and Structural similariry index (SSIM) [52].

3.8.1 PSNR

The most common and straightforward way to compute measurement metric is the mean square error (MSE) 3.2. It is obtained by averaging the difference between the original pixel values of an image and the pixel values of a reconstructed image. By using the MSE of the original image I and reconstructed RI, the PSNR expressed decibels (dB) is calculated as

$$MSE = \frac{1}{NN}\sum_{i=1}^{N}\sum_{j=1}^{N}[I(i,j) - RI(i,j)]^2 \tag{3.3}$$

$$PSNR = 10 \cdot log_{10}\left(\frac{MAX_I^2}{MSE}\right) \tag{3.4}$$

Here, MAX_I^2 is the maximum pixel value of the image. For grayscale image such value is 255. PSNR has the advantages of being simple and mathematically suited from the optimization point of view but, it also has two major disadvantages: it fails to keep the structure of the image which means that two images with the same MSE may have a considerable visual impact difference and it does not consider the human perception system. Desired values for PSNR should be above 30dB.

3.8.2 SSIM

SSIM is a method measuring the similarity between two images. Unlike PSRN, it uses a more complex approach to computing the difference in accordance with the human visual perception. It takes into consideration three features of an image: luminance (l), contrast (c) and structure (s). For two images I and RI, each term is evaluated as

$$l(I,RI) = \frac{2\mu_I\mu_{RI} + C_1}{\mu_I^2 + \mu_{RI}^2 + C_1} \tag{3.5}$$

$$c(I,RI) = \frac{2\sigma_I\sigma_{RI} + C_2}{\sigma_I^2 + \sigma_{RI}^2 + C_2} \tag{3.6}$$

$$s(I,RI) = \frac{\sigma_{IRI} + C_3}{\sigma_I\sigma_{RI} + C_3} \tag{3.7}$$

variables μ_I, μ_{RI}, σ_I, σ_{RI} and σ_{IRI} represent local means, standard deviations and cross-covariance of images I and RI. Arguments C_1, C_2 and C_3 are predefined constants. The final index is weighted a multiplication of the terms

$$SSIM(I,RI) = [l(I,RI)^\alpha \cdot c(I,RI)^\beta \cdot s(I,RI)^\gamma] \tag{3.8}$$

In practice $\alpha = \beta = \gamma = 1$, and $C_3 = \frac{C_2}{2}$ simplying to

$$SSIM(I,RI) = \frac{(2\mu_I\mu_{RI} + C_1)(2\sigma_{IRI} + C_2)}{(\mu_I^2 + \mu_{RI}^2 + C_1)(\sigma_I^2\sigma_{RI}^2 + C_2)} \tag{3.9}$$

This index can have values ssim $\in [0,1]$ with 0 meaning that the images are not similar at all and with 1 meaning that images are exactly the same. Ssim values above 0.7 are considered sufficiently good for any reconstruction algorithm.