Communication Systems
Semester Project
Bachelor Semester 5 - Fall 2022

Learning-based image coding for DNA storage

Student: Sophie Strebel

Supervised by: Davi Lazzarotto
Michela Testolina
Prof. Dr. Touradj Ebrahimi

January 13, 2023

Multimedia Signal Processing Group
EPFL

Abstract

Data is growing at an exponential rate: the amount of data generated in 2021 was roughly 79 zettabytes and is expected to reach 180 zettabytes by 2025 [1]. Data storage is expensive, but also requires a lot of energy and is said to be responsible for 2% of global carbon emissions [2]. Current storage systems are therefore reaching their limit and scientists are looking for new ways to store data. One solution is storing it into DNA, which is very long-lasting, energy friendly, astonishingly dense and will not become obsolete [3]. Since DNA is made up of 4 different nucleotides (ACGT), information stored into it must be encoded into a quaternary code that respects biochemical constraints due to the shape of the molecule [4]. Once the code is generated, DNA is synthesized, and later sequenced in order to reconstruct the data. Despite the several advantages mentioned above, there are multiple challenges such as random access and cost of sequencing and synthesis, which is expensive although expected to decrease.

Machine learning has become very common nowadays and is used in a variety of domains, for example social media, self-driving cars, and healthcare [5]. Learning-based image compression has been implemented using autoencoders and has given very good results [6].

The goal of this project is to study the state of the art in learning-based image compression and in DNA storage and coding, integrate a quaternary DNA coding algorithm into an existing autoencoder implementation, and evaluate the results by comparing them to an anchor method. The quaternary algorithm used was a ternary Huffman code followed by a Goldman code, implemented into CompressAI, an end-to-end image compression autoencoder. The results showed that the MS-SSIM and PSNR scores of the modified CompressAI implementation were better than those of the JPEG DNA Benchmark Codec, but the visual results were very different and thus difficult to compare.

Contents

1 Introduction

1.1 DNA based storage

DNA (Deoxyribonucleic acid) is a molecular structure made up of 4 types of nucleotides: Adenine, Citosine, Guanine and Thymine (abbreviated ACGT) [7] that makes up the genetic code of all living organisms [8].

Conventional data storage techniques are not suitable for long term storage [9], since their life expectancy is much lower than the time they will need to be stored for. For example, tape has a lifetime of 10-20 years, while 60% of data will need to be stored longer than 20 years [10].

Because of the exponential growth in digital data, scientists are looking for new solutions to store data, one of which is to store data in DNA.

This has several advantages, the most important of which is density: the whole world's storage needs could be stored in a single cubic meter of DNA [11]. Another advantage is its stability, as DNA can last for centuries with little damage [10]. Lastly, storing DNA is environmentally friendly as it requires very little energy and maintenance [12].

Despite all of these advantages, there are many challenges with DNA based storage. Synthetic DNA is divided into short strands of 150-300 nucleotides called oligos to increase robustness and stability [4]. These oligos have to respect the following biochemical constraints in order to avoid errors when sequencing them [4]:

1. **Homopolymers**: consecutive occurrences of the same nucleotides should be avoided.
2. **G, C content**: The percentage of G and C in the oligos should be lower or equal to the one of A and T.
3. **Pattern repetitions**: The codewords used to encode the oligos should not be repeated, forming the same pattern throughout the oligo length.

Another important challenge is the cost of sequencing and synthesis, which is expected to decrease in the future, making this technology more available [4]. Other challenges include the lack of random access as well as deletion, insertion and substitution errors during synthesis and sequencing [4].

1.2 The importance of machine learning and autoencoders

Machine learning is present almost everywhere nowadays to predict and analyse data [13]. Autoencoders are a type of machine learning model that generates data based on a given input [14].

The goal of an autoencoder is to compress the input to get a smaller representation [16], represented by "code" in Figure 1. From this representation, the original input can be reconstructed (with more or less loss depending on the input and the neural network models). Some use cases for autoencoders include dimensionality reduction and denoising data [16] [15], as well as image compression.

The application this project will focus on is end-to-end image compression. The input of such an autoencoder is an image, which is compressed and reconstructed to form the output. The autoencoder is trained to minimise the difference between the input image and the reconstructed image.

1.3 Objective

The objective of this project is study the state of the art in DNA storage and in autoencoders for end-to-end image compression, and to integrate a quaternary coding algorithm suitable for DNA storage into an existing end-to-end image compression autoencoder implementation.

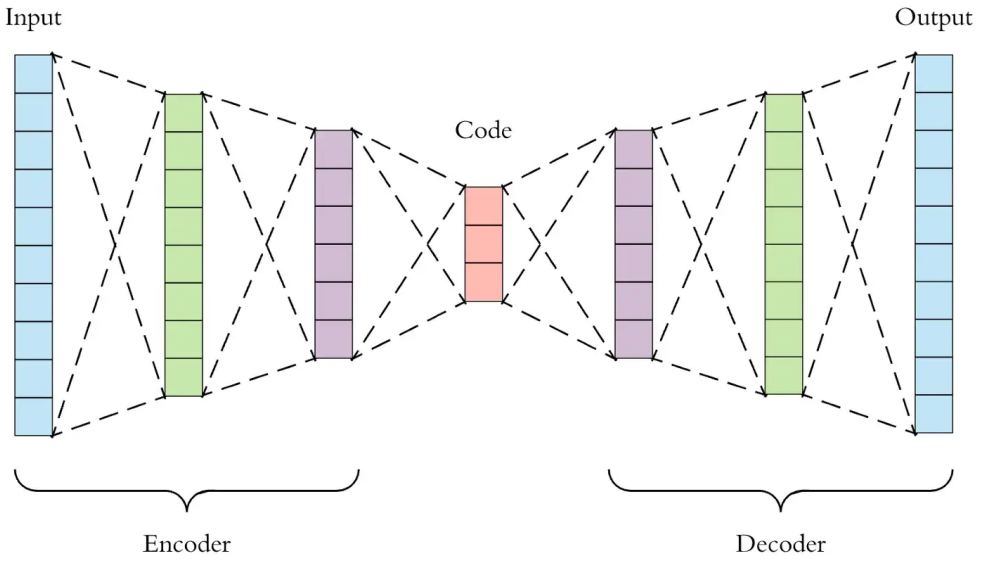


Figure 1: General shape of an autoencoder [15]

2 State of the art in DNA storage

2.1 End-to-end DNA storage architecture

Figure 2 shows the end-to-end architecture of DNA based media storage. Media is stored in binary, as shown on the image. To encode and decode binary data into DNA, the following steps are followed:

- **Encoding**: converts binary media into quaternary while respecting the biochemical constraints
- **DNA synthesis**: creates DNA from a sequence of letters (ACGT)
- **Encapsulation**: stores the synthesized DNA
- **Thermal damage simulation**: simulates damage that would affect DNA if kept over a long period of time
- **DNA release**: releases the encapsulated oligos
- **Sequencing**: reads the oligos and generates a sequence of bases
- **Decoding**: reconstructs binary data from the sequence of bases

Note that, in this project, we only convert data into a sequence of ACGT and restore the data from that sequence, corresponding to "Encoding" and "Decoding" on Figure 2, meaning there is no physical manipulation of DNA (synthesis, sequencing, encapsulation and so on).

2.2 JPEG DNA Benchmark Codec

Figure 3 describes the encoding process of the JPEG DNA Benchmark Codec (JPEG DNA BC) [18]. It takes as input a PNG image and an alpha value. The alpha value (typically a float between 0 and 3) determines the quantization table used. The lower the alpha value, the better the quality of the reconstructed image. The PNG image, originally in the RGB color space, is converted to the YCbCr color space. This is because the human eye is more sensitive to brightness (the luma component, Y) than to color (chroma components, Cb and Cr) and the YCbCr color space allows subsampling of the chroma components to reduce space [19]. The image is then

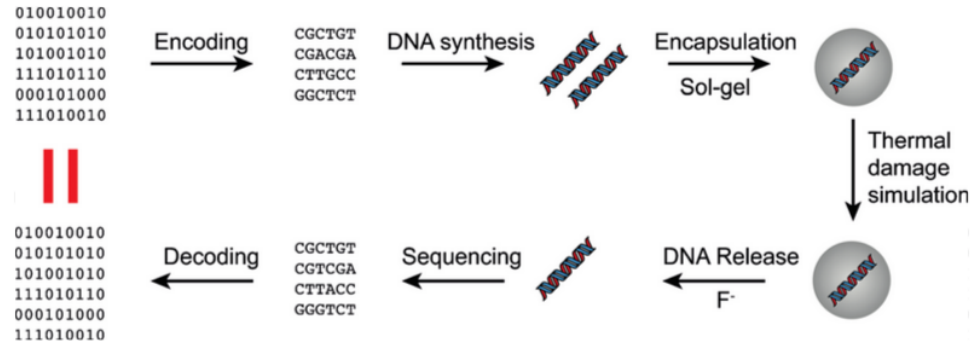


Figure 2: Coding and decoding steps of DNA based storage [17]

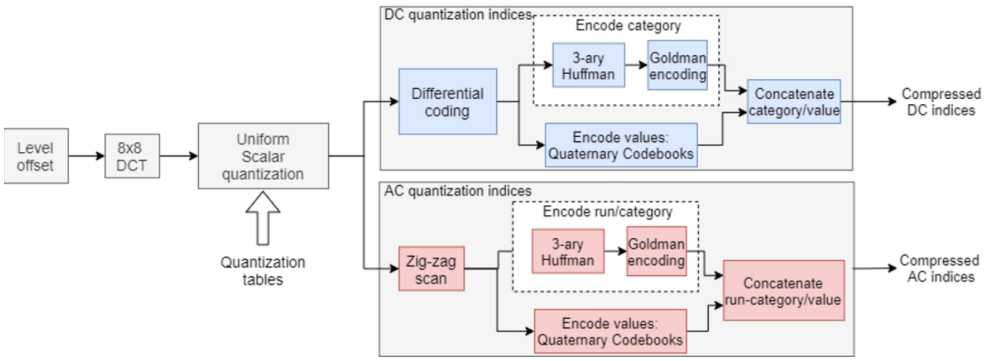


Figure 3: JPEG DNA BC workflow [4]

split up into 8 by 8 blocks, introducing padding if necessary and a discrete cosine transform (DCT) is performed on the blocks. This results in 1 DC coefficient and 63 AC coefficients for each block, which are quantized (divided by the quantization table). They are then separately encoded using AC and DC coefficient encoders, since the DC coefficients are encoded using a differential encoding, whereas a zig-zag transform is performed on the AC coefficients before encoding them. The encoding for both coefficients is based on a ternary Huffman code followed by a Goldman encoding, an encoding method proposed by Goldman et al. in 2013 [4], as shown by Figure 3.

The ternary Huffman code converts the binary sequence into a sequence of 0, 1, and 2 in an optimal way. These numbers are then encoded into DNA by excluding the previous base and associating the remaining 3 bases to 0, 1, and 2. This avoids homopolymers and is the first attempt of a DNA encoding algorithm that respects biochemical constraints [4].

The encoded concatenated DC and AC coefficients are then concatenated block by block. The Y, Cb and Cr encoded strings are also concatenated and yield the final encoded DNA string. Decoding runs the inverse operations (with some loss because of quantization) and reconstructs the image.

3 State of the art in learning-based image compression

3.1 End-to-end optimized image compression

Traditional compression methods make use of block partitioning, frequency transforms and other coding tools, which were all developed and optimized separately [20]. The goal of creating an end-to-end compression autoencoder is to optimize everything all at once for the sole goal of compressing (and decompressing) images [20], in order to better conserve the image structure.

For example, the DCT is best used on linear and gaussian data. However, the image data is in general not gaussian and the rate distortion optimal transform is not linear [20].

The goal of the autoencoder described in [21] is to minimize the rate and distortion. A lower rate means a more compressed image, whereas lower distortion means the reconstructed image is most similar to the original (and therefore little information is lost). Since there is a trade-off between the two, for simplicity, the method proposed in [21] aims to minimize $R + \lambda D$ (where R is the rate and D the distortion) for some constant λ that controls the trade-off.

While stochastic gradient descent is very popular, in the case of image compression where the latent space is quantized, the quantized latent space $\hat{y}$ as a function of the latent space y is a step function, meaning the derivatives are either 0 or ∞, neither of which are appropriate for gradient descent [20]. The way this has been solved is to replace quantization by an addition of uniform noise during training, yielding $\tilde{y}$, which allows a continuous relaxation of the probability distribution of y (then used for arithmetic coding).

This non linear transform coding is more flexible and allows lower rate and distortion sum $R + \lambda D$ (while not necessarily being optimal) [20].

Another advantage of this method is that it can be optimised for any quality metric just by changing the loss function [6]. The model can therefore yield better results (depending of the metric used to assess the results), for example when optimized for MS-SSIM rather than MSE (mean squared error). This leaves the question of what metric captures the visual loss best and should be used.

3.2 CompressAI implementation

In this project, the autoencoder implementation used is the one proposed in [22]. It is an implementation of the state of the art method described above. There are multiple trained networks according to the theory explained above, but the network used for this project was the `bmshj2018-factorized`. This model was chosen rather than a model with a hyperprior (`bmshj2018-hyperprior`) for simplicity, since a hyperprior would imply encoding multiple tensors into DNA. The network is given a quality level ranging from 1 to 8, and then used to encode an image, x, that is put through the network in a forward part g_a, which results in the latent space representation of the image, y. y is a tensor with a shape defined by the quality level and the image size: $1 \times n \times \lceil \frac{h}{16} \rceil \times \lceil \frac{w}{16} \rceil$, where h and w are respectively the height and width of the original image, and n is the number of channels, which depends on quality level. Quality levels 1 to 5 have 192 channels, whereas higher levels have 320 channels. The division of height and width by 16 emerges because of the 4 convolutions in the network, whereas taking the ceil results from padding, which is added to the image if its height and/or width is not divisible by 16.

$\hat{y}$ is obtained by rounding the latent space tensor y and then adding the medians m, which are predefined values set during training. The quantized latent space is therefore $\hat{y} = \lfloor y \rceil + m$.

Finally, $\hat{y}$ is put back through the inverse transformations of the autoencoder g_s and finally yields $\hat{x}$, the reconstructed image vector.

The entire process can be tested in a jupyter notebook demo provided by CompressAI, which was modified to contain the additions made by this project.

4 State of the art in learning-based DNA coding

Learning-based image compression with a DNA encoding algorithm has been done in [23], where an autoencoder (visible on Figure 4) was trained to minimize rate and distortion while dealing with substitution errors in the encoded DNA string. Because the algorithm encoded values into a DNA strand of some fixed length, a substitution error in the code does not affect its neighbors, thus adding noise to the quantized latent space before encoding into DNA has the same effect as applying this noise at the nucleotide level [23]. The results of this research showed that optimizing

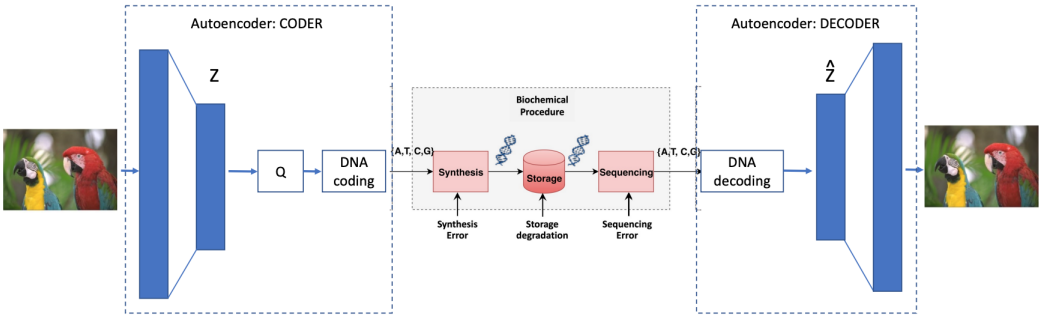


Figure 4: Shape of the autoencoder described in [23]

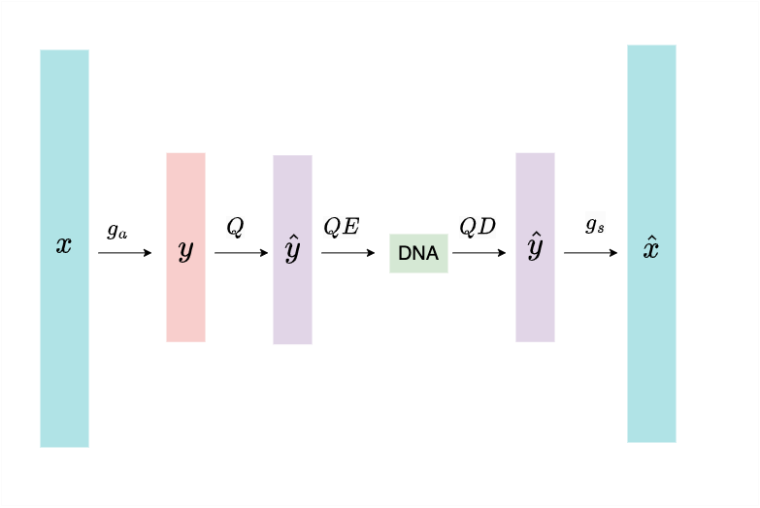


Figure 5: General workflow of this project

a model for noise gives better results when there is noise in the test phase, but underperforms when there is no noise. In contrast to this research, in this project, we use a previously trained end-to-end image compression autoencoder and integrate a quaternary algorithm into it, rather than training with DNA encoding and decoding. Additionally, we use a lossless quaternary encoding/decoding algorithm, thus do not simulate substitutions in the encoded DNA strand.

5 Learning-based DNA coding

The objective of the project is to incorporate a quaternary encoding algorithm into CompressAI, an end-to-end image compression autoencoder. We encode the quantized latent space image representation $\hat{y}$ into quaternary, decode it, and put it back through the autoencoder to get the reconstructed image $\hat{x}$. This method gives the same results as the original CompressAI implementation (without quaternary encoding and decoding) because the quaternary encoding algorithm is lossless. This workflow is shown on Figure 5, where Q stands for quantization, QE and QD for quaternary encoding and decoding respectively and g_a and g_s are the analysis and synthesis functions of the autoencoder.

5.1 Encoding the latent space into DNA

As explained above, the goal is to encode $\hat{y} = \lfloor y \rceil + m$ (see subsection 3.2) into DNA. Since m contains floats, $\hat{y}$ is a float tensor and cannot be encoded using the DNA quaternary entropy coding algorithm. However, since m is known, $\hat{y} - m$, which is $\lfloor y \rceil$ and only contains integer

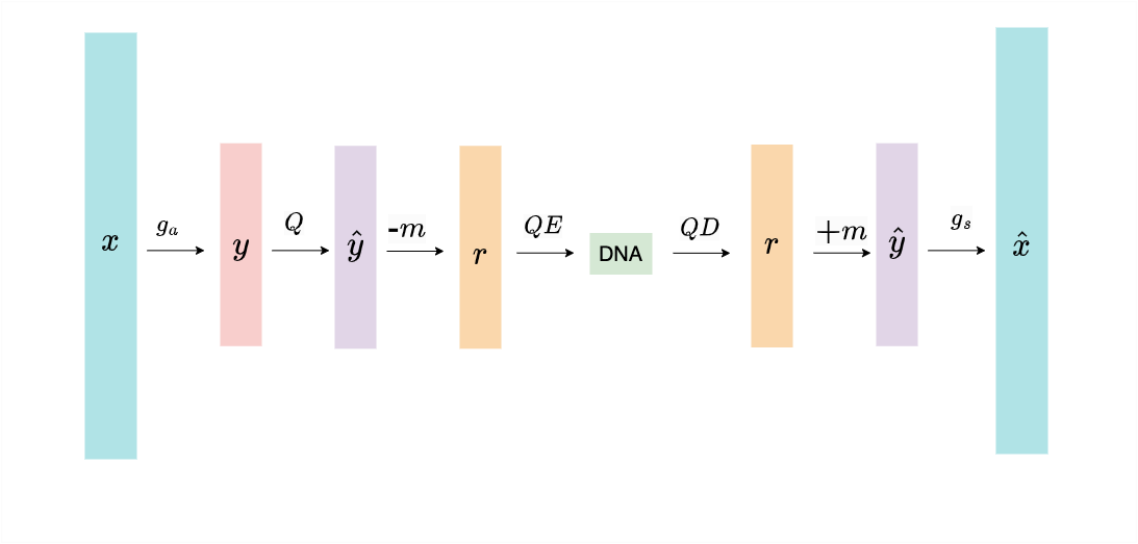


Figure 6: Updated workflow of this project

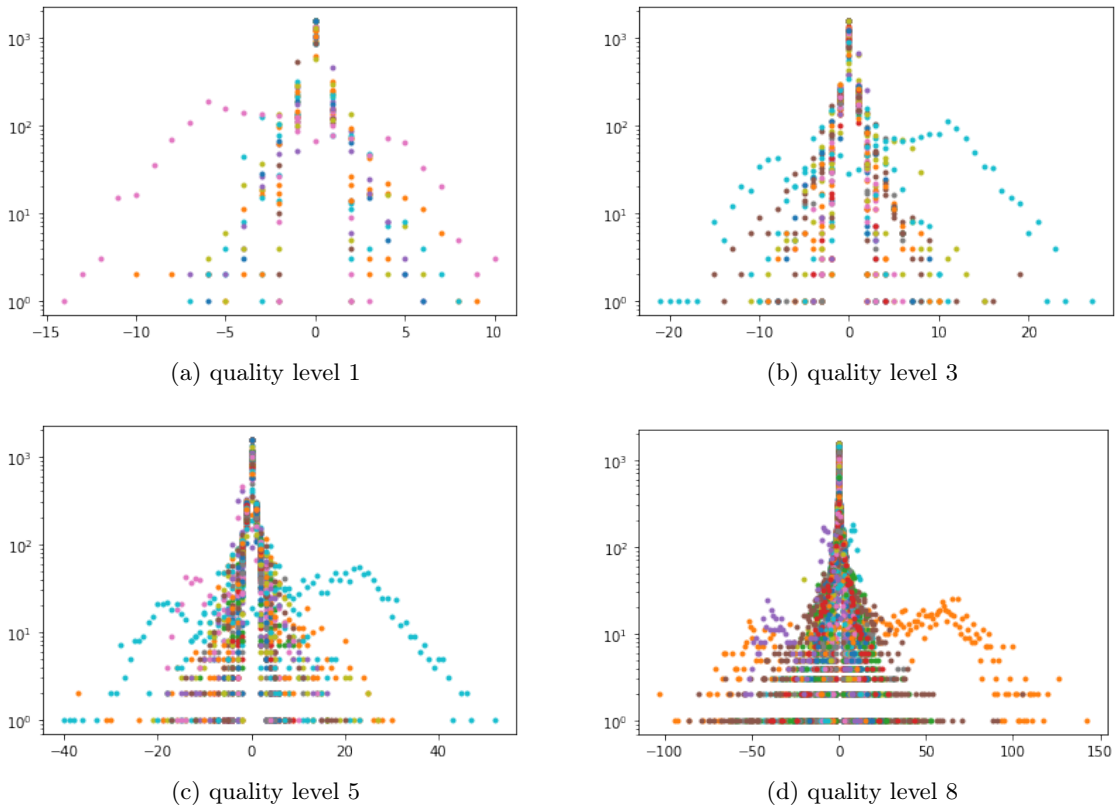
values can be encoded. $\hat{y} - m = \lfloor y \rceil$ is a rounded tensor, meaning it only contains integer values, so we call it r. The revised workflow looks like shown on Figure 6. Note that since entropy coding is lossless, decoding the encoded tensor r gives the exact same tensor r.

However, since many values of r are 0, it does not make sense to encode all of them. r is a tensor of shape $1 \times n \times \lceil \frac{h}{16} \rceil \times \lceil \frac{w}{16} \rceil$, where n is the number of channels defined by the quality level. Figure 7 shows the values contained in each of the n channels on the x-axis and the number of times this value appears in the channel on a logarithmic scale on the y-axis for the first image of the Kodak dataset encoded with quality levels 1, 3, 5, and 8. Logically, there are many more meaningful values for higher quality levels than lower quality levels. Note that all tested images were distributed similarly. Despite $n = 192$ for quality levels 1, 3, and 5, the number of channels containing values other than 0 is increasing with higher quality levels. Clearly, many channels contain only 0. We call these the trivial channels. Only non-trivial channels are relevant to encode. Note that higher quality levels not only have more non-trivial channels, but these channels contain a wider range of values. For example, for Kodak image 1, the values for quality level 1 range from about -15 to 10, whereas for quality level 8, they range from -100 to 150 (see Figure 7). The exact range of values depends on the encoded image, but all images were centered around 0 and showed an increase in range with increasing quality levels.

The quality levels each determine a certain number of non-zero channels:

- quality level 1: $n_1 = 25$ non-trivial channels
- quality level 2: $n_2 = 36$ non-trivial channels
- quality level 3: $n_3 = 53$ non-trivial channels
- quality level 4: $n_4 = 78$ non-trivial channels
- quality level 5: $n_5 = 107$ non-trivial channels
- quality level 6: $n_6 = 173$ non-trivial channels
- quality level 7: $n_7 = 222$ non-trivial channels
- quality level 8: $n_8 = 275$ non-trivial channels

Quality levels 1 through 5 have 192 channels total, while quality levels 6, 7 and 8 have 320.



(a) quality level 1

(b) quality level 3

(c) quality level 5

(d) quality level 8

Figure 7: Values in each channel of r for kodim01 at various quality levels

This means that, when encoding an image at a quality level q, only n_q channels need to be encoded.

In order to encode only the relevant channels of r, the relevant channels, which are the same for any image as they are defined by the network, are extracted, flattened, and encoded using the quaternary encoding algorithm which will be described below in subsection 5.2. To decode the encoded DNA string, the tensor r (of shape $1 \times n \times \lceil \frac{h}{16} \rceil \times \lceil \frac{w}{16} \rceil$) is reconstructed using the shape of the image and the list of non-trivial channels (given by the quality level). After that, m is added to reconstructed tensor r in order to get $\hat{y}$ which is then put back into the autoencoder for the inverse transformation g_s, yielding $\hat{x}$, the reconstructed image.

5.2 Quaternary (AGCT) encoding algorithm

The quaternary algorithm was extracted from the JPEG DNA BC. However, as explained in subsection 2.2, the JPEG DNA BC first performs a DCT on blocks of the image, and then encodes them using an AC and DC coeffient encoder respectively, both of which are based on a Huffman and Goldman code. Since, in this project, the values to encode are integer values of tensor r, which are not results of a DCT, it does not make sense to use the AC and DC coefficient encoders. Instead, the entropy coding was simplified to build a Huffman dictionary based on frequencies of values in r. More precisely, the dictionary is constructed from the values contained in the non-trivial channels of r and the number of times each value appears. From this information, the most common values are associated with shorter codewords whereas values that appear less are associated with longer codewords. Then, these integer values (contained in the non-trivial channels of r) are encoded into 0, 1, and 2, which is followed by a Goldman encoding, explained in subsection 2.2.

The Huffman and Goldman coders were taken from JPEG DNA BC, but needed to be adapted

slightly in order to work for the CompressAI implementation. The bug fixes are described in the Appendix A. Another issue was that the implementation of `encode` of `HuffmanCoder` was very slow. While is completely fine for JPEG DNA BC, as the actual encoding is done by the AC and DC coefficient coders, it was not viable to encode all values of the channels in CompressAI. `encode` was therefore parallelized, which allowed the total encoding of 275 channels to be done in about 40 seconds, compared to 15 minutes for 192 channels before parallelization (for an image of size 512×768).

5.3 Implementation

5.3.1 Quaternary Entropy Coding

The quaternary coding algorithm python code contains `AbstractCoder`, `HuffmanCoder` and `GoldmanCoder`, all three of which were taken from JPEG DNA BC (with the modifications mentioned in subsection 5.2 for `HuffmanCoder`). Additionally, it contains `Coder`, which encapsulates the whole quaternary entropy coding process added to CompressAI. `Coder` contains an `encode` and a `decode` method, which encode r and decode the DNA string and reconstruct r respectively. Recall that r is a tensor of shape $1 \times n \times \lceil \frac{h}{16} \rceil \times \lceil \frac{w}{16} \rceil$ defined as $r = \hat{y} - m$, as seen on Figure 6 and in subsection 5.1.

Note that `Coder` also contains a list of non-trivial channels for each quality level.

- `encode` takes r and a quality level, filters r to only keep the channels to encode based on the quality level and takes their integer part (since the tensor is a float tensor but the values are integers for example "2.000" instead of "2"). It then constructs a Huffman coder based on the elements contained in the non-trivial channels of r and their frequencies. Since the Huffman coder encodes strings, we convert the integers to strings and encode the flattened tensor into ternary using the Huffman coder and then into quaternary using the Goldman Coder. It returns the encoded DNA string.

- `decode` takes the encoded DNA string, the quality level and the shape of the image tensor x. It decodes the encoded DNA string with the Huffman and Goldman coders. In order to reconstruct the tensor r, the decoded values must be converted to floats (since r is a float tensor containing integers) and reshaped into $n_q \times \lceil \frac{h}{16} \rceil \times \lceil \frac{w}{16} \rceil$ where n_q is the number of non-trivial channels at the given quality level q. h and w are given by the image shape parameter. A `for` loop then reconstructs r by adding the trivial channels (in the correct order) and returns it.

5.3.2 CompressAI modifications

Most of the code modified from CompressAI is `google.py`, which computes $g_a(x)$ and $g_s(\hat{y})$. The `forward` method of `bmshj2018-factorized` directly returns $\hat{x}$. Since we need to encode $r = \hat{y} - m$, we not only need $\hat{y}$, but m. Then, the inverse transformations need to be run. Two methods were added:

- `get_y_hat_and_medians` takes x and returns $\hat{y}$ and m. From this, we can reconstruct r to encode and decode (into/from DNA), as well as reconstruct $\hat{y}$ from the decoded version of r.

- `get_x_hat` takes $\hat{y}$ and runs it through the network to return $\hat{x} = g_s(\hat{y})$.

`get_y_hat_and_medians` has to return m. Since the calculations with m are done in `forward` in `EntropyBottleneck` (`forward` returns $\hat{y}$), `forward` was modified as well to return m. The code in `google.py` was adapted accordingly.

5.3.3 Jupyter Notebook Demo

This is where most of the code for this project is. All the added functions are listed and somewhat explained here, but there is more documentation in the jupyter notebook directly.

The functions added to form, evaluate the modified autoencoder and compare the results to the JPEG DNA BC images are:

- `compute_x_hat` takes an image tensor x and a quality level and runs x through the modified autoencoder. This means it calculates $g_a(x)$, encodes and decodes the result into DNA and back to a tensor r, calculates $g_s(\hat{y}) = \hat{x}$ and returns the result, as well as the length of the encoded DNA string. This will be used to compute the rate in order to compare to the JPEG DNA BC results. Additionally, this function can return r, which is used to plot the channel values like in Figure 7.

- `plot_channels` plots the channels like in Figure 7.

- `get_non_trivial_channels` was used before to extract the non-trivial channels. Now that they are known (since they only depend on the quality level) and saved in `Coder`, this function is no longer used.

- `get_image` extracts the image tensor x from a given location, as well as the size of the image. This is a general function used later on both to extract the images encoded with the JPEG DNA BC and the images encoded/to encode with the modified autoencoder. The size of the image is used to compute the rate (nucleotides per pixel) when calculating and plotting the quality metrics.

- `get_dna_compressai_x_hats` runs an image through the modified autoencoder for all 8 quality levels and returns the result. This function, while used in other functions, is no longer used overall but was left for completeness. It was used in order to compute all images of the datasets through the modified autoencoder and to obtain the results presented in section 6.

- `get_plot_values_compressai_dna` returns the rates, MS-SSIM and PSNR values of a CompressAI encoded image in order to plot the results shown in section 6 (for example Figure 10).

- `get_jpeg_images` gets all the reconstructed image tensors $\hat{x}$ that were encoded and decoded with the JPEG DNA BC with different alpha values.

- In order to calculate the rate, the length of the encoded DNA strands and the size of the image for each alpha value were saved to a text file for each image in the JPEG DNA BC. `get_values_from_file_jpeg` extracts these values in order to use them to compare the results to the modified autoencoder.

- `get_plot_values_jpeg` returns the rates, MS-SSIM and PSNR values of the JPEG DNA BC encoded images in order to plot the results shown in section 6 (for example Figure 10).

- `plot_jpeg_dnaCompressAI` uses the functions defined above `get_plot_values_jpeg` and `get_plot_values_compressai_dna` to plot the results like in Figure 10.

- `calculate_and_plot_image` uses all the previous functions to run an image of a dataset through the modified autoencoder for all quality levels as well as extracts the images encoded and decoded using JPEG DNA BC for all alpha values, calculate and plot the quality metrics.

- `calculate_reconstructed_image` does the same as `calculate_and_plot_image` but without plotting the result. This was used to run the code for all images of datasets, but is no longer relevant.

- `add_to_lists` adds rates, PSNR and MS-SSIM values and $\hat{x}$ of the JPEG DNA BC and of the modified autoencoder of an image to given lists. The lists contained all the information now stored in pickle variables in a compact way. This is also no longer relevant, since all the information is extracted from the pickle variables.

- `get_rates_xhats_quality_levels` created the lists mentioned above for a certain dataset. The results of these lists were saved to pickle files, most of which are on the git repository of the project. The rest of the files should be added to the `compressai/examples` folder of the project in order for the code to run properly on the jupyter notebook. Note that this is only necessary to plot or show already encoded images, not to encode a new image.

- `get_average` extracts the average for a given list. This function was used to get the average of all MS-SSIM, PSNR or rates averages over all images of a dataset in order to get plots like on Figure 10.

- `get_averages` gets the averages of the rates and PSNR and MS-SSIM values of the lists containing all the information of the encoded images of a dataset. These are the lists stored in pickle variables.

- `get_ci` calculates the 95% confidence interval of the PSNR/MS-SSIM values for a dataset encoded with CompressAI or JPEG DNA BC.

- `get_all_ci_s` calculates the 95% confidence interval of the PSNR and MS-SSIM values for a dataset encoded with CompressAI and JPEG DNA BC (using `get_ci`).

- `get_pickle_variable` gets the variable saved in a pickle file. This function is used to extract the values previously stored in the pickle files.

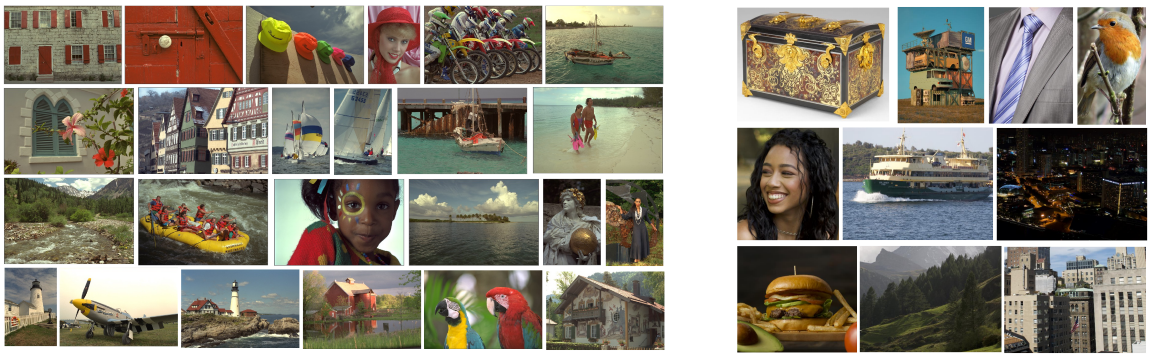
- `get_xhats_from_pickle_file` gets the reconstructed image tensors $\hat{x}$ from the pickle files for the JPEG AIC dataset. This is because the files were so big that they were separated into 2: $\hat{x}$ for the first 4 quality levels are in one file, the rest are in another (for each JPEG DNA BC and the modified autoencoder results). This functions gets all reconstructed $\hat{x}$ and stores them into variables. Concretely, it gets all $\hat{x}$ for all 8 quality level for each original image x of the JPEG AIC dataset, encoded with the modified autoencoder and stores them, and does the same with the images encoded and decoded with the JPEG DNA BC.

- `plot_single_dna_im` plots the quality metric results for a single image of the JPEG AIC dataset.

- `plot_single_kodim` plots the quality metric results for a single image of the Kodak dataset.

- `plot_single_img` plots the quality metric results for a single image of the JPEG AIC or the Kodak dataset depending on given parameter. The plot resembles Figure 11.

- `plot_crop_images` plots a crop of an image reconstructed with JPEG DNA BC and with the modified autoencoder, as well as the original.



(a) Kodak dataset

(b) JPEG AIC dataset

Figure 8: Datasets used for evaluation

- `plot_image` plots a crop of an image of either the JPEG AIC dataset or the Kodak dataset at given quality levels by calling `plot_crop_images`. The quality levels (quality level for CompressAI and 1-based index of the alpha value, both between 1 and 8) provided should be manually chosen so that the rates are as similar as possible. Figures like Figure 12 give an idea of the 1-based index to chose. Note that plotting lower nucleotide rates gives a better idea of the difference between JPEG DNA BC and the autoencoder.

6 Results and Discussion

6.1 Datasets

Two datasets were used to assess the results: the Kodak dataset, which consists of 24 color images of size 768×512 (both vertical and horizontal) and the JPEG AIC dataset, which consists of 10 color images of various sizes (ranging from 560×888 to 2592×1946). See both datasets on Figure 8.

Note that for the images with sizes not divisible by 16 (some images from the JPEG AIC dataset), the shape of x is not the same as that of $\hat{x}$ in CompressAI, as can be seen on Figure 9. The padding is always added on the right and bottom of the images. In order to compare the images with MS-SSIM and PSNR, the reconstructed image $\hat{x}$ was cropped, removing the right and bottom padding so that the images had the same size.

6.2 Comparison to the JPEG DNA Benchmark Codec

The quality and rate of the reconstructed image in JPEG is determined by an alpha value. In order to compare the MS-SSIM and PSNR metrics to the 8 quality levels of CompressAI implementations, 8 alpha values were chosen so that the rates of both methods were in a similar range. The chosen alpha values were $[0.2, 0.37, 0.5, 0.75, 1., 1.5, 2., 3.]$.

6.2.1 Quality metrics

Figure 10 shows an average across both datasets of the PSNR and MS-SSIM quality metrics at each rate. The rate is equal to the length of the DNA strand of the encoded image divided by the number of pixels of the image, and is expressed in nucleotides per pixel. Since the rates depend on the length of the DNA strand, which is not solely based on the quality level, different images at the same quality level do not necessarily have the same rate. Because of this, the rates on Figure 10 were also averages across all images of the respective datasets for each quality level/alpha value.

Figure 9: Original and reconstructed JPEG AIC image

We see that the Kodak dataset 95% confidence intervals are much smaller, especially for the MS-SSIM values. This makes sense because there are more images in the Kodak dataset than in the JPEG AIC dataset. Additionally, while JPEG never surpasses CompressAI for the Kodak dataset, it is slightly better in both MS-SSIM and PSNR for the JPEG AIC dataset for some rates, and the confidence intervals of the JPEG AIC dataset for PSNR have a big overlap for both encoding methods. We also see that for both datasets, the confidence intervals are much smaller for MS-SSIM than for PSNR.

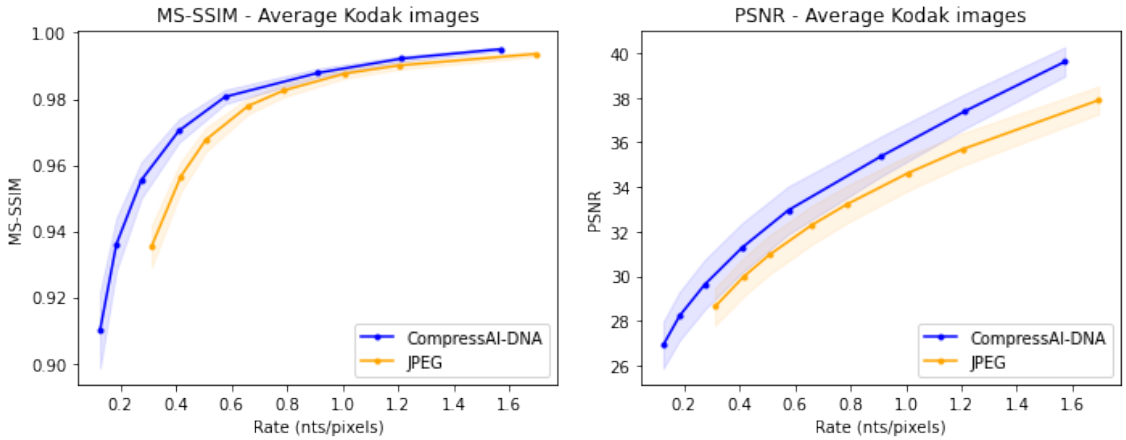
The second image of the JPEG AIC dataset (see Figure 13 because Figure 8 does not have the images in the correct order) has particularly different results compared to the other images (Figure 11). The JPEG DNA BC MS-SSIM and PSNR values are much higher than the CompressAI metrics. Additionally, it is worth noticing that this image has higher PSNR than most other images for both CompressAI and JPEG, as JPEG is at 41.9 and CompressAI is at 41.4, whereas the average is at 39.9 for CompressAI and 39.0 for JPEG.

Contrastingly, the 12th image of the Kodak dataset is the image with the most deviation compared to the others of that dataset. The plot of its quality metrics is shown on Figure 12. For most other Kodak images, the CompressAI metrics are better than the JPEG DNA BC results with little to no overlap, as shown with the values and the confidence intervals on Figure 10. The deviation for Kodak image 12 is much smaller than the ones of the JPEG AIC dataset (in particular JPEG AIC image 2, see Figure 11).

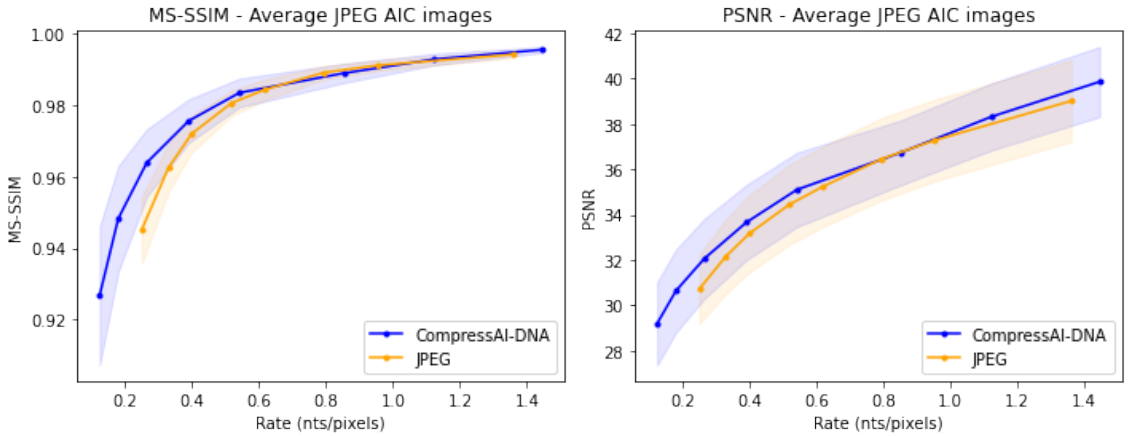
6.2.2 Example images and crops

Generally, CompressAI has better results than JPEG in both MS-SSIM and PSNR. However, these methods are very different which makes them difficult to compare visually. Note that most of the examples shown here are compared at low rates because that allows the difference between JPEG DNA BC and CompressAI to be clearer.

A first example on image 2 of the JPEG AIC dataset. As visible on Figure 11, the lowest rate of JPEG and second lowest of CompressAI are very close. The images and a few crops are visible on Figure 13. The image encoded using CompressAI (middle column) is much smoother than the JPEG DNA BC image (left column), on which the pixels are much more visible. The top left crop is very interesting because due to the blur, it is very smooth in the original image (right column). The CompressAI reconstructed image crop has slightly different colors (shades



(a) Average MS-SSIM and PSNR values for the Kodak dataset



(b) Average MS-SSIM and PSNR values for the JPEG AIC dataset

Figure 10: Average MS-SSIM and PSNR values for the Kodak and the JPEG AIC datasets with a 95% confidence interval

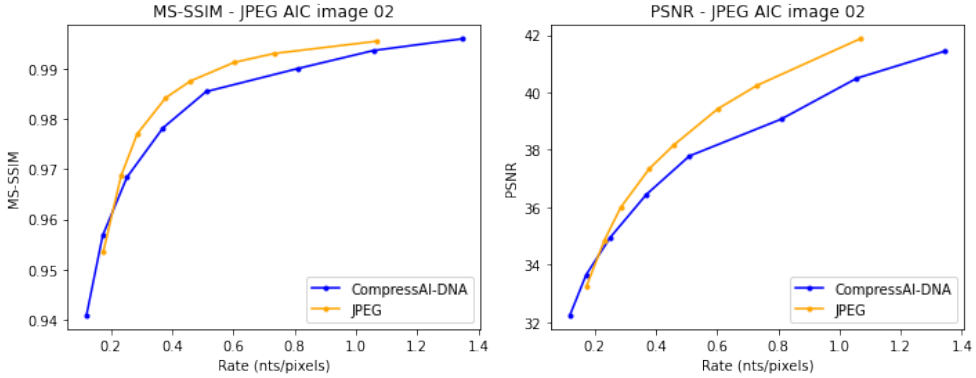


Figure 11: MS-SSIM and PSNR values for JPEG AIC image 2

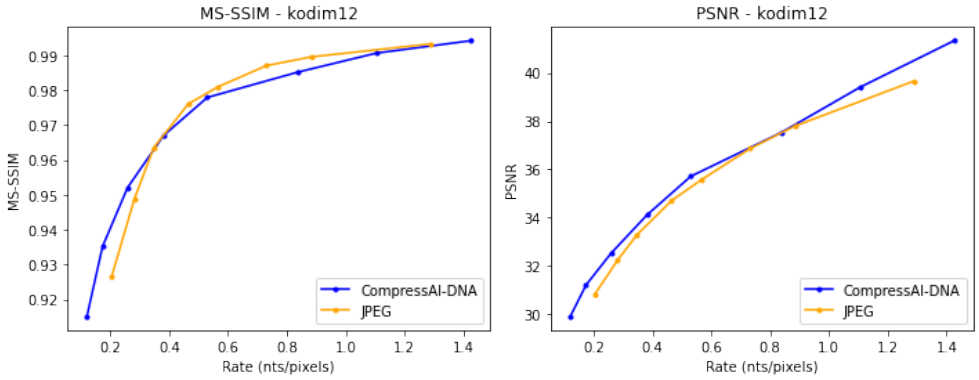


Figure 12: MS-SSIM and PSNR values for Kodak image 12

Figure 13: JPEG AIC image 2 at rate of ≈ 0.173 nts/pixel

of red in the middle), but still has a very natural look to it. Contrastingly, the JPEG DNA BC reconstructed image is more pixelated and has stricter boundaries between different colors. However, Figure 13 shows that, at this rate, CompressAI has better MS-SSIM and PSNR scores, so despite the rates being similar, it is expected that the CompressAI image looks more like the original. This makes the second crop more surprising: while the eye is strange (too smooth and not smooth enough respectively) in both CompressAI and the JPEG DNA BC, the hair in CompressAI is very smooth and therefore seems less realistic. In contrast, the hair in the JPEG DNA BC seems to have more texture and is more realistic.

Figure 14 shows crops of the 12th image of the Kodak dataset. As seen on Figure 12, the closest rates are around 0.27, which are the second alpha value of JPEG and third quality level for CompressAI. Note that CompressAI yields better MS-SSIM and PSNR scores.

Again, on the top left crop, we notice some starcasing on the JPEG DNA BC sky and water, whereas the CompressAI image is much smoother, and therefore looks natural but loses a lot of detail. The second crop is more interesting, as CompressAI smoothens the water, which seems natural in the first crop because the water is distant and still, but is less realistic on the second

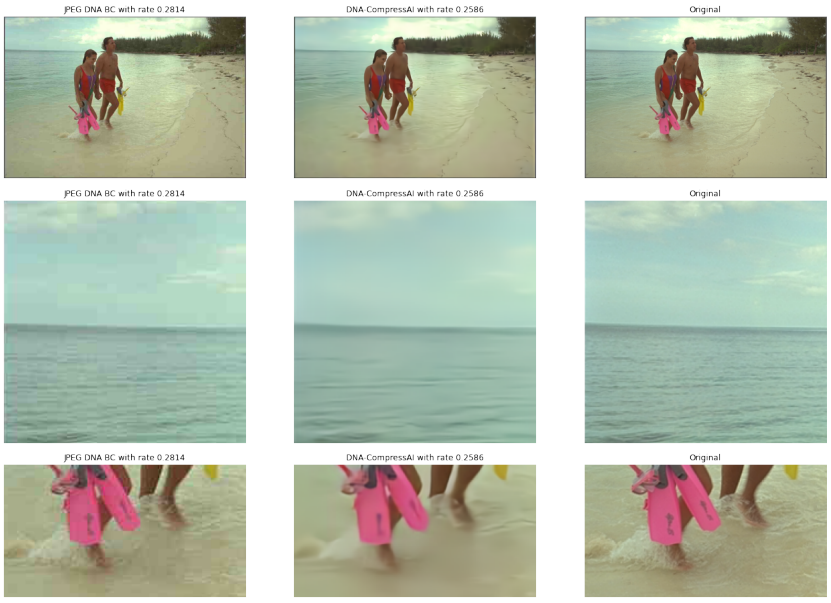


Figure 14: Kodak image 12 at a rate of ≈ 0.27 nts/pixel

crop. In contrast, JPEG DNA BC has more visible pixels on the second crop, but the splash of the water is more natural.

Some other results are included in figures 15, 16, 17 and 18. The PSNR and MS-SSIM scores and rates for each image are shown in the four tables below for both CompressAI and JPEG.

JPEG AIC image 9 lower rate	MS-SSIM	PSNR	Rate
CompressAI	0.9509	34.53	0.2388
JPEG DNA BC	0.9470	34.62	0.2249

JPEG AIC image 9 higher rate	MS-SSIM	PSNR	Rate
CompressAI	0.9768	37.44	0.4934
JPEG DNA BC	0.9793	37.73	0.4850

JPEG AIC image 4	MS-SSIM	PSNR	Rate
CompressAI	0.9679	32.65	0.3878
JPEG DNA BC	0.9588	31.73	0.3987

Kodim 19	MS-SSIM	PSNR	Rate
CompressAI	0.9501	29.65	0.2687
JPEG DNA BC	0.9316	28.78	0.2876

The crop on Figure 15 shows that the mountains and trees are much smoother and blurry for CompressAI than the JPEG counterpart image, which is much more pixelated. Note that this is less visible on the whole image since the it is very big (2048×1536). Contrastingly, the crop on Figure 16 shows the same image compressed at a higher rate. As seen on the table above, the JPEC DNA BC has better results than CompressAI in both PSNR and MS-SSIM for the higher rate. Visually, the crop of the CompressAI encoded image is still very blurry, albeit less than the same crop on Figure 15, but the JPEG DNA BC crop is much closer to the original and much less pixelated than with a lower rate.

Figure 17 shows staircasing in the sky for JPEG, whereas the sky for CompressAI is much more similar to the original. The writing does not seem very affected by the compression of

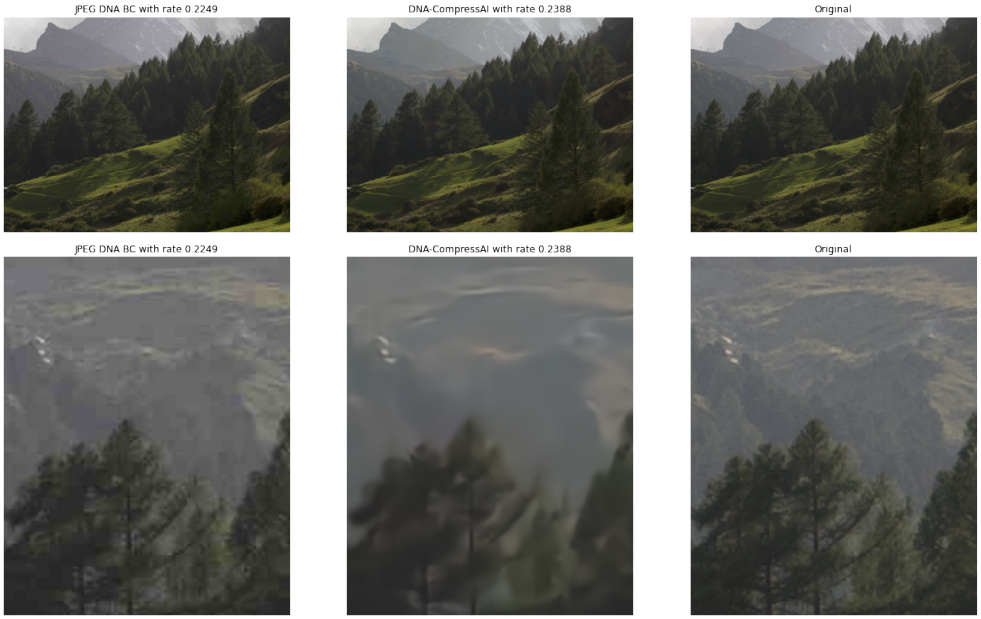


Figure 15: JPEG AIC image 9 at a rate of ≈ 0.23 nts/pixel

either JPEG or CompressAI and remains perfectly readable, which is probably due to its size (the original image is 2000×2496 and the crop is 500×600).

On the non-cropped image on Figure 18, the clouds and sky show the same behavior as the trees and clouds on Figure 15. The crop of Kodak image 19 shows the same trend. The post, which resembles a face in the original image, looks very blurred on the CompressAI image, the smile is barely visible. Additionally, the fence pickets on the left of the post are less separated. On the JPEG DNA BC image, the image has very visible blocks but does not at all look blurred. The boundaries between objects is much clearer.

To conclude, the results show that the CompressAI implementation has a "blur" effect on the original image so that the boundaries between different objects are less strict and the objects lose texture because they are smoother. The JPEG DNA BC shows a lot of staircasing and blocks, as well as more pixelated results in general.

7 Conclusion

This project studied the state of the art in DNA storage and coding as well as learning-based image compression, in particular the JPEG DNA BC and CompressAI. Then a quaternary DNA entropy coding was implemented into the existing CompressAI autoencoder. The results of performance were assessed using the Kodak and the JPEG AIC dataset, then compared to the JPEG DNA BC results using MS-SSIM and PSNR.

The results showed that CompressAI tends to have better PSNR and MS-SSIM scores than the JPEG DNA BC. The visual results of the two methods were very different, with CompressAI smoothening every texture and JPEG pixelating them, which made them difficult to compare.

Further work could include implementing a faster and more optimal quaternary encoding algorithm in CompressAI instead of the Huffman and Goldman code implemented in this project, as well as using another autoencoder, such as the variational autoencoder with a hyperprior from [6]. Another interesting addition would be error simulation and correction in the quaternary entropy coding algorithm, which was lossless in this project but would not be if the DNA string was synthesized into DNA and sequenced to reconstruct the string.

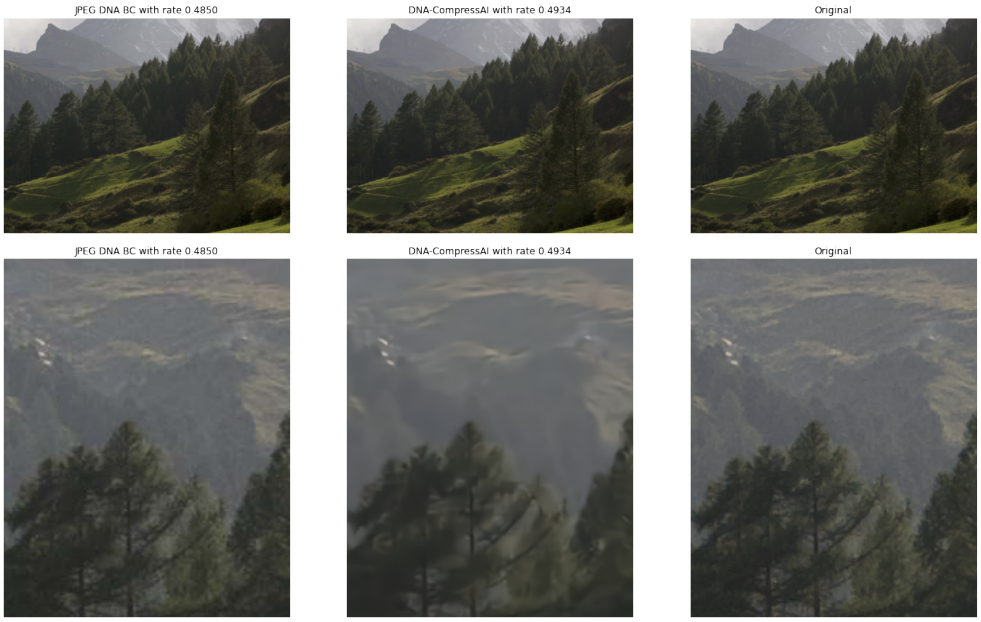


Figure 16: JPEG AIC image 9 at a rate of ≈ 0.49 nts/pixel

Figure 17: JPEG AIC image 4 at a rate of ≈ 0.39 nts/pixel

Figure 18: Kodak image 19 at a rate of ≈ 0.27 nts/pixel

References

[1] Ogi Djuraskovic. 30+ big data statistics (2023) - amount of data generated in the world, Dec 2022. URL: https://firstsiteguide.com/big-data-stats/#:~:text=By%202022%2C%20the%20big%20data,of%20data%20in%202019%20alone.

[2] Stephanie Safdie. What is the carbon footprint of data storage?, Oct 2022. URL: https://greenly.earth/en-us/blog/ecology-news/what-is-the-carbon-footprint-of-data-storage.

[3] ROBERT F. SERVICE. DNA could store all of the world's data in one room, Mar 2017. URL: https://www.science.org/content/article/dna-could-store-all-worlds-data-one-room.

[4] Marc Antonini, Luis Cruz, Eduardo da Silva, Melpomeni Dimopoulou, Touradj Ebrahimi, Siegfried Foessel, Eva Gil San Antonio, Gloria Menegaz, Fernando Pereira, Xavier Pic, et al. DNA-based Media Storage: State-of-the-Art, Challenges, Use Cases and Requirements version 8.0. 2022.

[5] Nikita Duggal. Top 10 machine learning applications and examples in 2023, Dec 2022. URL: https://www.simplilearn.com/tutorials/machine-learning-tutorial/machine-learning-applications.

[6] Johannes Ballé, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. Variational image compression with a scale hyperprior. *arXiv preprint arXiv:1802.01436*, 2018.

[7] The Editors of Encyclopaedia Britannica. DNA, Nov 2022. URL: https://www.britannica.com/science/DNA.

[8] CK-12 Foundation. 4.6: Genetic code, Mar 2021. URL: https://bio.libretexts.org/Bookshelves/Introductory_and_General_Biology/Book%3A_Introductory_Biology_(CK-12)/04%3A_Molecular_Biology/4.06%3A_Genetic_Code.

[9] Samuel Greengard. The Future of Data Storage. *Commun. ACM*, 62(4):12, mar 2019. doi:10.1145/3311723.

[10] WG1 JPEG. 1st JPEG DNA Workshop. URL: https://www.youtube.com/watch?v=U5EKe-RDYN4.

[11] Sang Yup Lee. DNA data storage is closer than you think, Jul 2019. URL: https://www.scientificamerican.com/article/dna-data-storage-is-closer-than-you-think/.

[12] Latchesar Ionkov. DNA: The Ultimate Data-Storage Solution, May 2021. URL: https://www.scientificamerican.com/article/dna-the-ultimate-data-storage-solution/.

[13] Sara Brown. Machine Learning, explained, Apr 2021. URL: https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained.

[14] Ahmed Fawzy Gad. Image compression using autoencoders in Keras, Apr 2021. URL: https://blog.paperspace.com/autoencoder-image-compression-keras/.

[15] Arden Dertat. Applied deep learning - part 3: Autoencoders, Oct 2017. URL: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798.

[16] Sayantini Deb. How to perform data compression using autoencoders?, Sep 2020. URL: https://medium.com/edureka/autoencoders-tutorial-cfdcebdefe37.

[17] Andrea Missinato. DNA storage: The solution to Big Data in a strand?, Oct 2018. URL: https://www.nonteek.com/en/dna-storage-solution-to-big-data-in-a-strand/.

[18] JPEG DNA Benchmark Codec, Oct 2022. URL: https://gitlab.com/wg1/jpeg-dna/jpeg-dna-benchmark-codec.

[19] Reducible. The Unreasonable Effectiveness of JPEG: A Signal Processing Approach. URL: https://www.youtube.com/watch?v=0me3guauqOU&t=1474s.

[20] Johannes Ballé. PCS 2018 – Learned Image Compression, July 2018. URL: https://www.youtube.com/watch?v=x_q7cZviXkY.

[21] Johannes Ballé, Valero Laparra, and Eero P Simoncelli. End-to-end optimized image compression. *arXiv preprint arXiv:1611.01704*, 2016.

[22] Jean Bégaint, Fabien Racapé, Simon Feltman, and Akshay Pushparaja. CompressAI: a PyTorch library and evaluation platform for end-to-end compression research. *arXiv preprint arXiv:2011.03029*, 2020.

[23] Xavier Pic and Marc Antonini. Image Storage on Synthetic DNA Using Autoencoders. *arXiv preprint arXiv:2203.09981*, 2022.

A Bug fixes in `HuffmanCoder`

There were a few bugs in the `decode` function of `HuffmanCoder`. `decode` takes an encoded string `code` to decode, and goes through it linearly (since Huffman codes are prefix free, meaning no codeword starts with another codeword), matching codewords to their keys and finally returns a list of integers (decoded version of the input). An issue not detected by tests was that the `decode` started going through `code` with the first 2 characters. For example, with "AGCTCGTA", `decode` first checked whether "AG" was a codeword, then "AGC", then "AGCT" and so on until it found the codeword and its key. This is obviously problematic if "A" itself is a codeword, and was easily fixed by replacing the starting index in the `for` loop.

When a codeword is found, `code` is modified to remove the codeword. In the previous example where `code`="AGCTCGTA", if "AGC" was a codeword, once it was found, `code` would be set to "TCGTA". The procedure goes on until `code` is the empty string. This lead to another bug because, when a codeword is found and `code` is updated, the whole process of decoding should start over. Instead, the `for` loop that looked through `code`, adding a character to try to find the codeword continued. In the example above, `code`="AGCTCGTA", the `for` loop ranged over the indices of `code` and at each iteration, looked for the codeword. Again, we assume the codeword is "AGC", meaning `code` is updated to "TCGTA". Then, the `for` loop index is $i = 2$ (because the codework found was from indices 0 to 2, inclusive). By continuing the `for` loop, the codeword from "TCGTA" started at $i = 3$ (iteration after "AGC" was found), meaning the first searched codeword is from indices 0 to 3, "TCGT". Obviously this does not always work (consider the case where "T" is a codework). Adding a `break` in the `for` loop when a codework was found solved the issue, as the whole process started again for the new updated codeword.

The original code is the following:

```
1  def decode(self, code):
2      decoded = []
3      max_len_codeword = 0
4      for el in self.dic.values():
5          if len(el) > max_len_codeword:
6              max_len_codeword = len(el)
7      while code != "":
8          for i in range(1, min(len(code)+1, max_len_codeword)):
9              res = self.find_codeword_key(code[0:i+1])
10             if res is not None:
11                 decoded.append(res)
12                 code = code[i+1:]
13     return decoded
```

See the modified code below. The only changes are the starting index of `range` on line 8, as well as the added `break` on line 13.

```
def decode(self, code):
    decoded = []
    max_len_codeword = 0
    for el in self.dic.values():
        if len(el) > max_len_codeword:
            max_len_codeword = len(el)
    while code != "":
        for i in range(0, min(len(code)+1, max_len_codeword)):
            res = self.find_codeword_key(code[0:i+1])
            if res is not None:
                decoded.append(res)
                code = code[i+1:]
                break
    return decoded
```