WARSAW UNIVERSITY OF TECHNOLOGY

Data Compression

Faculty of Mathematics and Information Science

Portrait Mode Compression

- Final Report -

By **Elie SAAD & Matthieu Dabrowski**

January 22, 2021

Contents

1 Problem Formulation

Before explaining right away the subject of our project, we need to define several concepts that are quite important in the context of our project. We will then explain how we came up with this idea, before clearly defining what the end goal of this project is.

1.1 Definitions

1.1.1 Portrait Photo

This project, and the algorithm we are aiming to design in the context of this project, is made to be applied almost exclusively to what we call ”Portrait Photos”. This term denominate a certain kind of picture where the subject of the photo (in other words, what the photograph was trying to take a picture of) can be clearly established. Said subject can usually be found in the foreground of the picture, and the information we find in this area is relevant to the picture. On the other hand, the background of such pictures, and the objects and information it may contain, are less relevant to the picture.

1.1.2 Portrait Mode

A lot of smartphones allow you to take pictures in a ”Portrait Mode”, in which the subject of the portrait photo appears clearly, while the background is blurred, in what is often described as a somewhat artistic way. This is made so that the eye is automatically attracted by the subject of the picture, while the background, which is less relevant, can be left unnoticed. Most of those ”Portrait Modes” rely at least partly on the fact that the subject of the picture can be found in the foreground. The resulting image therefore contains less information than the same image taken without this ”Portrait Mode” feature, as there is some loss of information in the background.

1.2 Project Idea

The idea for this project came from an observation. We noticed that a lot of smartphones allow you to take pictures in a ”Portrait Mode”, but even though the resulting picture actually contains less information than the same photo without this ”Portrait Mode” feature after the computational photography algorithms have been applied, the size of the Portrait mode picture is generally the same as for the same image without this ”Portrait

Mode" feature and before the computations.

This is why we started thinking about a way to compress images in such a way that the foreground, where the subject of a portrait is supposed to be located, would be compressed without any loss of information, while there could be some loss of information on the background, since that information is less relevant for a portrait photo.

If this kind of algorithm was revealed to be performing well, it would be useful among other things for storing photos found on the Internet, as portrait photos are quite common on social media websites. This could also be applied in some cases of medical imaging, for images obtained via a CT-Scan for example, where only some parts of the images are relevant for medical purposes.

1.3 End Goal of the project

The end goal of this project is to design a new compression algorithm, that can be applied to any picture but would be more efficient on what we defined as Portrait Photos.

This algorithm would first differentiate the foreground of the image, where the subject of the photo is supposed to be, from its background, which is supposed to contain less relevant information. It would then compress the image by applying a lossless compression on the foreground to avoid any loss of the relevant information that can be found here, and apply a lossy compression on the background, since we can afford some loss of this less relevant information. Part of the end goal is also to design the corresponding decompression algorithm.

The main success criteria concerns the size of the obtained compressed file, that must be smaller than the size of the input image. The loss of information should be visible on the background of the decompressed image, as we find it interesting, but that is not a success criteria for this project.

2 Theoretical Background

2.1 Cross-Task Consistency

To understand this method, we first have to start off y explaining what is consistency in machine learning, why is it important, and how do we use it. And then finally we will explain how that correlates to our work of depth estimation.

First of all let us tackle the first question of what consistency is. Suppose an object detector detects a ball in a particular region of an image, while a depth estimator returns a flat surface for the same region. This presents an issue – at least one of them has to be wrong, because they are inconsistent. More concretely, the first prediction domain (objects) and the second prediction domain (depth) are not independent and consequently enforce some constraints on each other, often referred to as consistency constraints.

Second of all, we will tackle the question of why is it important to consider consistency in learning: First, desired learning tasks are usually predictions of different aspects of one underlying reality (the scene that underlies an image). Hence inconsistency among predictions implies contradiction and is inherently undesirable. Second, consistency constraints are informative and can be used to better fit the data or lower the sample complexity. Also, they may reduce the tendency of neural networks to learn ”surface statistics” (superficial cues), by enforcing constraints rooted in different physical or geometric rules. This is empirically supported by the improved generalization of models when trained with consistency constraints.

Figure 1 demonstrates the impact of disregarding consistency in learning as well as the effectiveness of augmenting learning with cross-task consistency constraints. Each window shows surface normals predicted out of a domain itself predicted out of the image (i.e. image $\Longrightarrow$ {prediction domain X} $\Longrightarrow$ surface normals). The normals in the upper row (learning without consistency) are poor and different/inconsistent with each other for the same underlying image. The lower row shows the same except when learning image $\Longrightarrow$ {prediction domain X} was augmented with cross-task consistency constraints with normals. Inferred surface normals look better and more similar to each other regardless of the middle prediction domain, which demonstrates all of the middle domains were successfully made cross-task consistent w.r.t to normals. In [7], they further extend this concept to many arbitrary

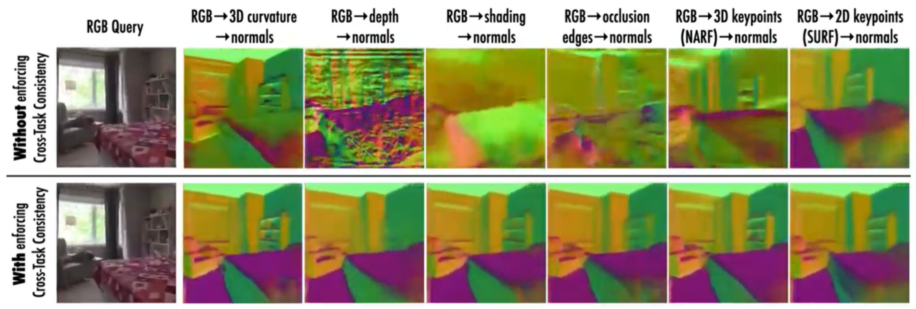

RGB Query
RGB→3D curvature →normals
RGB→depth →normals
RGB→shading →normals
RGB→occlusion edges→normals
RGB→3D keypoints (NARF)→normals
RGB→2D keypoints (SURF)→normals
Without enforcing Cross-Task Consistency
With enforcing Cross-Task Consistency

Figure 1: Impact of disregarding cross-task consistency in learning, illustrated using surface normals domain.

domains with arbitrary inference path lengths, using a general and fully computational learning framework.

Third of all we will tackle the question of how can we design a learning system that makes consistent predictions. [7] proposes a method which, given an arbitrary dictionary of tasks, augments the learning objective with explicit constraints for cross-task consistency. The constraints are learned from data rather than apriori given relationships. For instance, it is not necessary to encode that surface normals are the 3D derivative of depth or occlusion edges are discontinuities in depth. This makes the method applicable to any pairs of tasks as long as they are not statistically independent; even if their analytical relationship is unknown, hard to program, or non-differentiable.

The primary concept behind the method is inference-path invariance. That is, the result of inferring an output domain from an input domain should be the same, regardless of the intermediate domains mediating the inference. When inference paths with the same endpoints, but different intermediate domains, yield similar results, this implies the intermediate domain predictions did not conflict as far as the output was concerned. The authors in [7] apply this concept over paths in a graph of tasks, where the nodes and edges are prediction domains and neural network mappings between them, respectively. Satisfying this invariance constraint over all paths in the graph ensures the predictions for all domains are in global cross-task agreement.

In figure 2: **(a)** shows what a typical multitask setup where predictors $x \implies y_1$ and $x \implies y_2$ are trained without a notation of consistency (either completely independently or with a shared encoder and dedicated

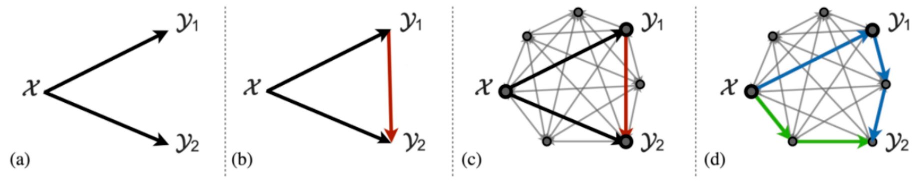


Figure 2: Enforcing cross-task consistency.

decoders). **(b)** depicts the elementary triangle consistency constraint where the prediction x_1 is enforced to be consistent with $x \implies y_2$ using a function that relates y_1 to y_2 (i.e. $y_1 \implies y_2$). **(c)** shows how the triangle unit from (b) can be an element of a larger system of domains. Finally, **(d)** illustrates the generalized case where in the larger system of domains, consistency can be enforced using invariance along arbitrary paths, as long as their endpoints are the same (here the blue and green paths). This is the general concept behind ”inference-path invariance”. The triangle in (b) is the smallest unit of such paths. The loss function derived in [7] that is used to enforce consistency between the models is

$$\mathcal{L}^{\text{perceptual}}_{xy_1y_2\ldots y_k} = |f_{xy_1}(x)-y_1|+|f_{y_{k-1}y_k}\circ\ldots\circ f_{y_1y_2}\circ f_{xy_1}(x)-f_{y_{k-1}y_k}\circ\ldots\circ f_{y_1y_2}(y_1)|$$

which is the loss for training f_{xy_1} using the arbitrary consistency path $x \implies y_1 \implies y_2 \implies \ldots \implies y_k$ with length k.

Finally, let us discuss how the aforementioned work correlates to our depth estimation problem. Let’s say that you’re interested in predicting some y_1 given x (in our case, depth given an RGB image). Instead of simply training a network to do $x \implies y_1$, we train it to do $x \implies y_2 \implies y_1$ (in our case we picked y_2 to be surface normals). It will fit the data better with improved RGB $\implies$ depth prediction.

All the pretrained networks in [7] are based on the UNet architecture described in [6]. They take in an input size of 256x256, upsampling is done via bilinear interpolations instead of deconvolutions. All models were trained with the L1 loss.

2.2 Separation of the Foreground from the Background

Given a greyscale image matrix M with elements of values between 0 and 255, we can flatten it by concatenating the row vectors a single long vector

v. Thus, given a vector v of a flattened image matrix of length n, the q-th percentile of v is the value $q/100$ of the way from the minimum to the maximum in a sorted copy of v. The values and distances of the two nearest neighbors as well as the interpolation parameter will determine the percentile if the normalized ranking does not match the location of q exactly. This procedure is the same as the median if $q = 50$, the same as the minimum if $q = 0$ and the same as the maximum if $q = 100$.

For an arbitrary value of q, we take the elements $v_i \in v$ that are $v_i \leq q$ and set them to $v_i = 0$, and all the elements $v_i \in v$ that are $v_i > q$ and set them to $v_i = 255$. Thus recreating the greyscale image M gives us a binary image of excursively black and white. The reconstructed matrix can then be compared with our original colored version of the image, and the pixels that are represented with white can be extracted as the background whereas those that are represented with black can be represented as the foreground.

This allows us to not have to look into the raw pixel values of every image to find a satisfying threshold, and instead, using the percentile variable q, we then need fewer tries to find the good value that obtains a satisfactory binary image. This method also allows us to understand what the values that we are trying as inputs mean more easily.

2.3 Lempel–Ziv–Welch

LZW is a general compression algorithm capable of working on almost any type of data. It is generally fast in both compressing and decompressing data and does not require the use of floating-point operations. Also, because LZW writes compressed data as bytes and not as words, LZW-encoded output can be identical on both big-endian and little-endian systems.

LZW is referred to as a substitutional or dictionary-based encoding algorithm. The algorithm builds a data dictionary (also called a translation table or string table) of data occurring in an uncompressed data stream. Patterns of data (substrings) are identified in the data stream and are matched to entries in the dictionary. If the substring is not present in the dictionary, a code phrase is created based on the data content of the substring, and it is stored in the dictionary. The phrase is then written to the compressed output stream.

When a reoccurrence of a substring is identified in the data, the phrase of the substring already stored in the dictionary is written to the output. Because the phrase value has a physical size that is smaller than the substring it represents, data compression is achieved.

Decoding LZW data is the reverse of encoding. The decompressor reads a code from the encoded data stream and adds the code to the data dictionary if it is not already there. The code is then translated into the string it represents and is written to the uncompressed output stream.

LZW goes beyond most dictionary-based compressors in that it is not necessary to preserve the dictionary to decode the LZW data stream. This can save quite a bit of space when storing the LZW-encoded data. When compressing text files, LZW initializes the first 256 entries of the dictionary with the 8-bit ASCII character set (values 00h through FFh) as phrases. These phrases represent all possible single-byte values that may occur in the data stream, and all substrings are in turn built from these phrases. Because both LZW encoders and decoders begin with dictionaries initialized to these values, a decoder need not have the original dictionary and instead will build a duplicate dictionary as it decodes.

TIFF, among other file formats, applies the same method for graphic files. In TIFF, the pixel data is packed into bytes before being presented to LZW, so an LZW source byte might be a pixel value, part of a pixel value, or several pixel values, depending on the image's bit depth and number of color channels.

The TIFF approach does not work very well for odd-size pixels, because packing the pixels into bytes creates byte sequences that do not match the original pixel sequences, and any patterns in the pixels are obscured. If pixel boundaries and byte boundaries agree (e.g., two 4-bit pixels per byte, or one 16-bit pixel every two bytes), then TIFF's method works well.

GIF requires each LZW input symbol to be a pixel value. Because GIF allows 1- to 8-bit deep images, there are between 2 and 256 LZW input symbols in GIF, and the LZW dictionary is initialized accordingly. It is irrelevant how the pixels might have been packed into storage originally; LZW will deal with them as a sequence of symbols.

The GIF approach works better for odd-size bit depths, but it is difficult to extend it to more than eight bits per pixel because the LZW dictionary must become very large to achieve useful compression on large input alphabets.

LZW does a very good job of compressing image data with a wide variety of pixel depths. 1-, 8-, and 24-bit images all compress at least as well as they do using RLE encoding schemes. Noisy images, however, can significantly degrade the compression effectiveness of LZW. Removing noise from an image, usually by zeroing out one or two of the least significant bit planes of the image, is recommended to increase compression efficiency. In other words, if your data does not compress well in its present form, transform it to a different form that does compress well.

One method that is used to make data more "compressible" by reducing the amount of extraneous information in an image is called differencing. The idea is that unrelated data may be easily converted by an invertible transform into a form that can be more efficiently compressed by an encoding algorithm. Differencing accomplishes this using the fact that adjacent pixels in many continuous-tone images vary only slightly in value. If we replace the value of a pixel with the difference between the pixel and the adjacent pixel, we will reduce the amount of information stored, without losing any data.

With 1-bit monochrome and 8-bit gray-scale images, the pixel values themselves are differenced. RGB pixels must have each of their color channels differenced separately, rather than the absolute value of the RGB pixels' differences (difference red from red, green from green, and blue from blue).

Differencing is usually applied in a horizontal plane across scan lines. In the following code example, the algorithm starts at the last pixel on the first scan line of the bitmap. The difference between the last two pixels in the line is calculated, and the last pixel is set to this value. The algorithm then moves to the next to last pixel and continues up the scan line and down the bitmap until finished, as shown in the following pseudo-code:

Algorithm 1 Horizontally difference a bitmap

```
1: for $Line \leftarrow 0, ..., NumberOfLines - 1$ do
2:     for $Pixel \leftarrow NumberOfPixelsPerLine - 1, ..., 1$ with step of $-1$ do
3:         $Bitmap[Line][Pixel] \leftarrow Bitmap[Line][Pixel] - Bitmap[Line][Pixel - 1]$
4:     end for
5: end for
```

Vertical and 2D differencing may also be accomplished in the same way. The type of differencing used will have varied effectiveness depending upon the content of the image. And, regardless of the method used, differenced images compress much more efficiently using LZW.

2.4 Huffman Coding

Huffman coding is an efficient method of compressing data without losing information. Huffman coding provides an efficient, unambiguous code by analyzing the frequencies that certain symbols appear in a message. Symbols that appear more often will be encoded as a shorter-bit string while symbols that aren't used as much will be encoded as longer strings. Since the frequencies of symbols vary across messages, there is no one Huffman coding that will work for all messages. This means that the Huffman coding for sending message X may differ from the Huffman coding used to send message Y. There is an algorithm for generating the Huffman coding for a given message based on the frequencies of symbols in that particular message.

Let us consider the following problem:

1. **Input:** Known alphabet (e.g., English letters a,b,c,...)
   Sequence of characters from the known alphabet (e.g., "helloworld")
   We are looking for a binary code—a way to represent each character as a binary string (each such binary string is called a codeword).

2. **Output:** Concatenated string of codewords representing the given string of characters.

The problem above amounts to the following. We are given an alphabet $\{a_i\}$ with frequencies $\{f(a_i)\}$. We wish to find a set of binary codewords $C = \{c(a_1), ..., c(a_n)\}$ such that the average number of bits used to represent the data is minimized:

$$B(C) = \sum_{i=1}^{n} f(a_i)|c(a_i)|.$$

Equivalently, if we represent our code as a tree T with leaf nodes $a_1, ..., a_n$, then we want to minimize

$$B(T) = \sum_{i=1}^{n} f(a_i)d(a_i),$$

where $d(a_i)$ is the depth of a_i, which is also equal to the number of bits in the codeword for a_i.

The following algorithm, due to Huffman, creates an optimal prefix tree for a given set of characters $C = \{a_i\}$. Actually, the Huffman code is optimal among all uniquely readable codes, though we don't show it here.

Algorithm 2 Huffman Tree

```
1: $n \leftarrow |C|$
2: $Q \leftarrow C$                          ▷ a min–priority queue keyed by frequency
3: **for** $i \leftarrow 1$ **to** $n-1$ **do**
4:     Allocate new node $z$
5:     $z.left \leftarrow x \leftarrow$ EXTRACT-MIN($Q$)
6:     $z.right \leftarrow y \leftarrow$ EXTRACT-MIN($Q$)
7:     $z.freq \leftarrow x.freq + y.freq$
8:     $Q$.INSERT($z$)
9: **end for**
10: **return** EXTRACT-MIN($Q$)              ▷ Return the root of the tree
```

2.5 Linde, Buzo, and Gray

Vector Quantization (VQ) technique outperforms other techniques such as pulse code modulation (PCM), differential PCM (DPCM), Adaptive DPCM which belongs to the class of scalar quantization methods. Vector quantization (VQ), one of the most popular lossy image compression methods is primarily a c-means clustering approach widely used for image compression as well as pattern recognition, speech recognition, face detection speech and image coding because of its advantages which include its simple decoding architecture and the high compression rate it provides with low distortion- hence its popularity. Vector Quantization involves the following three steps: (i) Firstly, the image in consideration is divided into non-overlapping blocks commonly known as input vectors. (ii) Next, a set of representative image blocks from the input vectors is selected referred to as a codebook and each representative image vector is referred to as a code-word. (iii) Finally, each of the input vectors is approximately converted to a code-word in the codebook, the corresponding index of which is then transmitted.

Codebook training is considered sacrosanct in the process of Vector Quantization, because a codebook largely affects the quality of image compression. Such significance of Codebook training gave new impetus to many researchers leading to the proliferation of researches to design codebook using several projected approaches. Vector Quantization methods are categorized into two classes: crisp and fuzzy. Sensitive to codebook initialization, Crisp VQ follows a hard decision making process. Of this type, C-Means algorithm is the most representative of all. In [5] propounded the famous Linde-Buzo-Gray (LBG) algorithm in this respect.

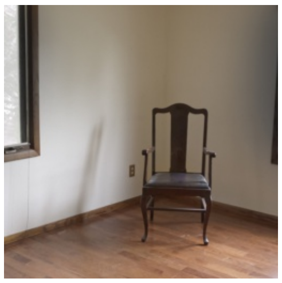
LBG generates a codebook with minimum error from a training set. Training sets are a set of vectors that are derived from image vectors. The code vectors must minimize the distortion. LBG algorithm assumes the codeword length to be fixed. It is an iterative procedure and the basic idea is to divide the group of training vectors and use it to find the most representative vector from one group. These representative vectors from each group are gathered to form the codebook. (i) Divide the given image into blocks, so that each block appears as a d-dimensional vector. (ii) Initial codebook is chosen randomly. (iii) The initial chosen codebook is set as the centroid and the other vectors are grouped according to the nearest distance with the centroid similar to the k-mean clustering. (iv) Find the new centroid of every group to get a new codebook iterative. Repeat the steps (ii) (iii) till the convergence of the centroid of every group.

3 Experimentation

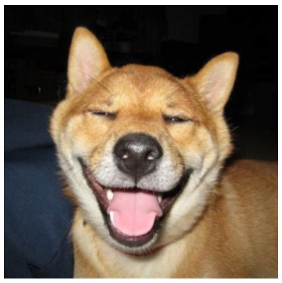
We have implemented the compression algorithms using the programming language called Python. We made use of a library called Numpy, which is a library that adds the ability to easily handle multi-dimensional arrays and matrices, and adds a large collection of high-level mathematical functions to operate on these arrays, which we have made use of to write the functions for the different compression algorithms that we have tested. We have also made use of another library called OpenCV, which is a ibrary of programming functions mainly aimed at real-time computer vision and machine learning, which we have made use of with the handling of our test images for the compression algorithms.

We first will begin by implementing the Cross-Task Consistency algorithm for Depth Estimation to have a basis for the compression algorithms. Then we will test out the lossless compression algorithms that we have researched in the theoretical part of our work, mainly the Lempel–Ziv–Welch (LZW) and the Huffman Coding which we have applied on images in our own way. And finally we will test out the lossy compression algorithm called Linde, Buzo, and Gray (LBG).

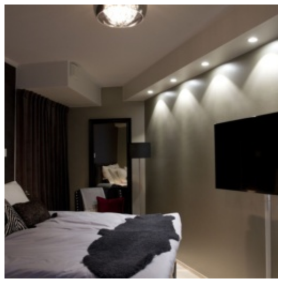
The experimentation will be conducted on six different images that we have personally selected to test the limits of the algorithms' capabilities. As shown in figure x, image numbers one and six are selected cause of their bright colour ranges. Image numbers three and five are selected cause of their mixture of bright and dark color ranges. And finally image numbers two and four are selected cause of their dark colour ranges.


(a) Image#1

(b) Image#2


(c) Image#3

(d) Image#4


(e) Image#5

(f) Image#6

3.1 Cross-Task Consistency for Depth Estimation

As explained in the theoretical background, the cross-task consistency method can be applied for depth models, allowing us to obtain an improved depth estimation image from an RGB image. In the interest of time, we have decided to use a pretrained model for our depth estimation that has been provided by the authors of [7] and that is publicly available on their paper's Github page.

Applying a threshold to the images obtained from the pretrained networks allows us to obtain an binary image, with exclusively black and white pixels. The white pixels represent a locally bigger depth estimation than the black pixels. This means object in the white area, which corresponds to the background, are supposed to be farther away from the camera than the objects in the black area, which therefore corresponds to the foreground.

Depending on the chosen threshold, those two areas could be either made bigger or smaller, and we could end up with the main subject of the picture accidentally ending up partially or fully in the background, or with some irrelevant objects ending up in the foreground. To avoid that, we need to think carefully about how we choose to apply the threshold.

It has already been explained in the theoretical background that this threshold is chosen as a percentile of the pixel values in the depth image. This means what we actually choose is the percentage of pixels in the depth image that need to be below that threshold (all the other pixels being above it). So we decided to try several thresholds applied on each of our selected test images. Here is a table representing the best threshold for every picture:

	Best Threshold
Image#1	60%
Image#2	60%
Image#3	70%
Image#4	70%
Image#5	60%
Image#6	60%

We can clearly see the best threshold on average is the 60 percentile, so we chose this threshold as the default threshold in the implementation of our algorithm, but we do have to stress that this parameter can still be manually changed depending on the image, since even in digital cameras, the focus of the focus of the subject is sometimes done manually rather than automatically.

3.2 Lempel–Ziv–Welch for Images

It has already been stated that we had decided to apply the Lempel-Ziv-Welch algorithm to images. We computed a compressed file for each of our images. Here are the results we got from that implementation:

	Uncompressed	Compressed
Image#1	192 kB	474 kB
Image#2	192 kB	481 kB
Image#3	192 kB	510 kB
Image#4	192 kB	470 kB
Image#5	192 kB	472 kB
Image#6	192 kB	569 kB
Mean	192 kB	496 kB

These results are not satisfactory at all, as the obtained compressed files are bigger in size than the image that are being compressed. We wondered why the Lempel-Ziv-Welch algorithm was performing so poorly in terms of Compression Ratio. We considered the fact that the Lempel-Ziv-Welch algorithm is mostly used on GIFs, that contain more information since they actually contain several images. And we figured that this might be linked to its lower performance on images.

Indeed, the Lempel-Ziv-Welch algorithm, like many compression algorithm, relies on finding some kind of repeated pattern in the data it is compressing. And if this algorithm is good on compressing sequences of images, it is because it is good at finding and compressing repeated patterns between one image and the next image in that sequence. But on a single image, such patterns can't be found, therefore the compression becomes way less impressive.

We also decided to try the Lempel-Ziv-Welch algorithm on grayscaled images, to see if it would achieve better performances. Here are the results of such experimentations:

	Uncompressed (Colored and Grayscaled)	Colored Compressed	Grayscale Compressed
Image#1	192 kB	474 kB	448 kB
Image#2	192 kB	481 kB	464 kB
Image#3	192 kB	510 kB	494 kB
Image#4	192 kB	470 kB	458 kB
Image#5	192 kB	472 kB	455 kB
Image#6	192 kB	569 kB	555 kB
Mean	192 kB	496 kB	479 kB

We can notice here that the Lempel-Ziv-Welch algorithm seems to be working better with grayscaled images than with images with three channels. But its performance on grayscaled images is still not satisfying for us. Moreover, the Cross-Task Consistency model can not obtain binary images from grayscaled images, therefore those kind of images would be useless for us.

At this point, we decided to change our minds and use the Huffman Coding instead of the Lempel-Ziv-Welch algorithm as the lossless compression we were going to apply to the foreground of our images.

3.3 Huffman Coding for Images

Based on what we have learned from the research on the theory part of our work, the idea of implementing the Huffman Coding to images did not seem too far fetched to us. So, we tried to adapt the concept of Huffman Coding to images. And what follows is the different ways we have come up with to adapt the Huffman Coding to images. But we knew that the Huffman Coding is applied on a sequence of data, and thus we had to come up with a method that transforms our images into a sequence of data instad of them being represented as a tensor.

Images are represented as tensors, which are a generalization of vectors and matrices and are easily understood as multidimensional arrays. The main representation of images is in what is called the RGB space, or Red-Green-Blue space, where each of the three axis in a 3-dimensional space represents the intensity of a color. The color intensity is represented by the numbers starting from 0 to 255, thus a combination of the three primary colors which are Red, Green, and Blue make up the entirety of the visible colors. Each one of these color is represented as what is called a channel in an image, thus an image is formed of three channels.

Given this knowledge, we have come up with the idea to flatten the three channels of the image into vectors, and use them as input to our Huffman Coding compression function to compress the vectors and generate the codec(s) required to losslessly decode the vectors and restructure them into the original image. And thus, we faced another problem where we have to decide how the codec is created. And thus, we came up with three different ideas which are explained below.

The first method that we came up with consists of splitting the image into three channel vectors, then concatenating the vectors one after the other to create a larger vector. Then we use this large vectors to compute the codec which will then be used to encode it. The following is are the results of our experimentation

	Uncompressed	Compressed
Image#1	192 kB	154 kB
Image#2	192 kB	165 kB
Image#3	192 kB	172 kB
Image#4	192 kB	161 kB
Image#5	192 kB	160 kB
Image#6	192 kB	183 kB
Mean	192 kB	165.8 kB

The second method that we came up with consists of splitting the image into three channel vectors, then we create a codec for each one of the vectors then encode each of the vectors with its proper codec. The following is are the results of our experimentation

	Uncompressed	Compressed
Image#1	192 kB	154 kB
Image#2	192 kB	166 kB
Image#3	192 kB	173 kB
Image#4	192 kB	161 kB
Image#5	192 kB	161 kB
Image#6	192 kB	183 kB
Mean	192 kB	166.3 kB

The third method that we came up with consists of splitting the image into three channel vectors and one larger vector consiting of all the three channel vectors concatenated one after the other. We use the data from this bigger vector to create one codec, and encode each one of the smaller vectors using the codec obtained from the larger vector. The following is are the results of our experimentation

	Uncompressed	Compressed
Image#1	192 kB	161 kB
Image#2	192 kB	172 kB
Image#3	192 kB	178 kB
Image#4	192 kB	168 kB
Image#5	192 kB	170 kB
Image#6	192 kB	190 kB
Mean	192 kB	173.2 kB

Those three aforementioned methods are all technically and theoretically working and have been tested as displayed in the tables above, but the main goal of our algorithm is to achieve the smallest file size at the end, and thus we can clearly see from the result tables displayed above that the first method, computing only one codec and only encoding only one vector grants the lowest file size overall, and thus is the method that we have decided to use for the implementation of our algorithm later.

3.4 Linde, Buzo, and Gray

The Linde-Buzo-Gray algorithm, as it has been explained already, uses several parameters apart from the image to compress: the size of the codebook, and the shape of the blocks used for the compression. Those two parameters have an influence on the size of the compressed file, but also on the quality of the decompressed image. Indeed, the bigger the size of the blocks is, the smaller the size of the compressed file is. In the same idea, if the codebook contains less elements, it is smaller, so the compressed file is smaller as well. But if we have bigger blocks, we have less of them, so that necessarily reduces the quality of the decompressed image. And if we have a smaller codebook, we have fewer different blocks in the decompressed image, which means the quality is reduced as well.

It is also worth stating that these parameters have an influence on the computing time of the compression algorithm. Generally speaking, if we choose to set our parameters in a way that will increase the quality of the decompressed image, and increase the size of the compressed file, there is a good chance the computing time will increase as well. But we were not very interested in optimizing the computing time of our algorithm, as it remained satisfying most of the time.

We therefore tried several values for the size of the codebook, on the same image (Image #3), with blocks of size (8,8), to see how that parameter would influence the compression exactly. Here are the results of these experiments:

Codebook Size	Blocks Size	Uncompressed	Compressed
2	$(8, 8)$	192 kB	4 kB
4	$(8, 8)$	192 kB	6 kB
8	$(8, 8)$	192 kB	11 kB
16	$(8, 8)$	192 kB	16 kB
32	$(8, 8)$	192 kB	26 kB

Then, for a codebook of size 8, we tried several values for the size of the block, for the same purpose as the precedent experiment. Here are the results:

Codebook Size	Blocks Size	Uncompressed	Compressed
8	$(1, 1)$	192 kB	38 kB
8	$(2, 2)$	192 kB	13 kB
8	$(4, 4)$	192 kB	7 kB
8	$(8, 8)$	192 kB	11 kB
8	$(16, 16)$	192 kB	23 kB

We can see here that, while changing the size of the codebook had a significant and linear impact on the size of the compressed file, as we predicted, changing the size of the blocks had a different impact than what we expected before. Indeed, if we increase the size of the blocks of more than $(4, 4)$, (for this image and a codebook size of 8), we notice that the size of the compressed file is actually increasing. This may seem surprising, but actually, when we increase the size of the blocks, we also increase the size of the vectors to be encoded using the codebook. The codebook contains the centroids of the clusters where the vectors representing the image blocks can be found, which means it contains vectors of the same size as those representing the blocks. Therefore by increasing the dimensions of the blocks we increase the size (in Bytes) of the codebook without increasing its length, and this is why the size of the compressed file is increasing as well.

It then seems obvious that the best blocks size (at least for this image) is $(4, 4)$, as it corresponds to the smallest compressed file, but does not reduce the quality of the decompressed image too much. The dimensions of the images to compress are sure to have an important impact on the optimal block size to choose, as it must be a divisor of the image shape (without considering the number of channels). Here, all our images' dimensions are $(256, 256, 3)$.

Considering the size of the codebook, we chose 8 as a default size, as it seemed like a good compromise between the fact that we want a small compressed file, and the fact that we do not want the quality of the decompressed image to be too bad. So our default parameters for this algorithm are $codebook\_size = 8$ and $block = (4, 4)$.

4 Implementation

After all those experimentations with our different algorithms, we were finally able to build our own compression algorithm based on what we had learned. Once again, our goal was to apply a compression to the background of the image and an other compression on the foreground. The following is a description of how we made all those algorithms work together to achieve that goal.

First, let us talk about the inputs of our algorithm. The first input, and the only non-optional one, is the path to the image being compressed. The other inputs are the necessary parameters for obtaining a binary image (that is to say, the percentile) and for the Linde-Buzo-Gray algorithm (the codebook length and the blocks shape). As it has been stated before, those parameters have a default value, and don't need to (but can) be changed manually.

Our algorithm does several things in a linear way. First, it computes the binary image thanks to the cross-task consistency model, and using the percentile parameter. All the pixels from the original image that have been detected as the foreground are then put in a vector, ordered by their coordinates: it is important that the top-left pixel is the first one in that vector, then we can find the other pixels of the top row, from left to right, then the row under it, and so on. All pixels from the first channel of the image are put in the vector first, then the pixels from the second channel, and then the third. Using the Huffman encoding, we create a codec from that vector, and then we encode this very same vector thanks to the codec we just created.

Then, using the codebook length and blocks shape parameter, we compress the original image using the Linde-Buzo-Gray algorithm, which means we obtain a codebook containing the centroids of the clusters where we can find all the vectors representing the image blocks, and a small image which is the compressed version of the original image, and contains as pixel values the indices of the corresponding centroid in the codebook that this block should be represented by.

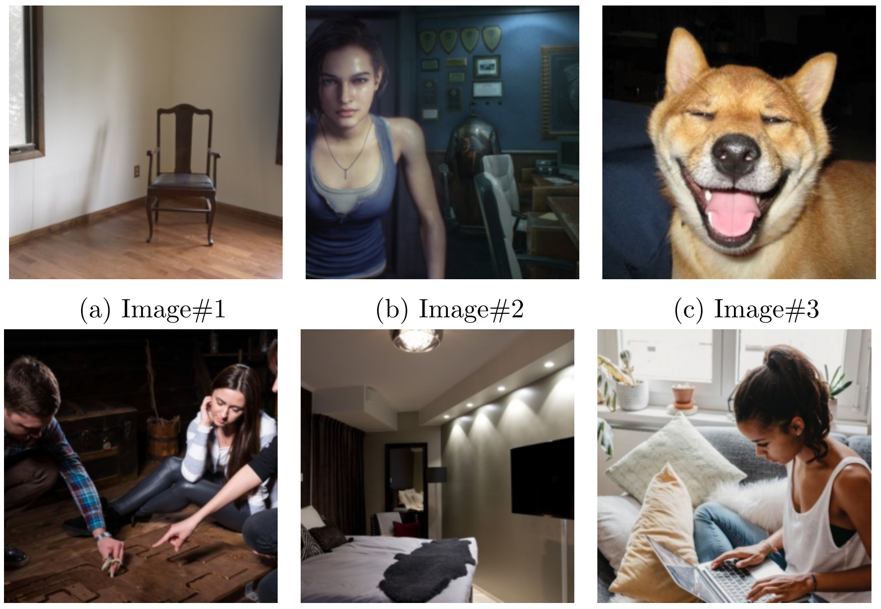
Then we need to save several things in the compressed file: the codec and the encoded vector, as we need it for the Huffman decoding; the binary image, so we know which pixels correspond to the background and which pixels correspond to the foreground; the codebook and the compressed image, for the Linde-Buzo-Gray decompression; the block shape, so we know what shape should the centroid vectors in the codebook have in the decompressed image. Those informations are all the data available for the decompression. We call the obtained file the Portrait Mode compressed file

Once we extracted all the necessary objects and data from the compressed file, the decompression can be done in several steps. First, we use the codebook, the compressed image and the blocks shape to decompress the original image, and obtain the Linde-Buzo-Gray decompressed image. Then we use the codebook to decode the encoded vector. And finally, we iterate through the binary image and, everytime we encounter a black pixel (which corresponds to the foreground), we replace the corresponding pixel (the pixel of same coordinates) in the Linde-Buzo-Gray-decompressed image by the first value in the decoded vector, and delete the value we just used from said decoded vector. At the end of this process, the decoded vector should be empty, and the obtained image corresponds to the Portrait Mode Decompression of the compressed file.

The following table represents the compressed files size for our compression algorithm, compared to the size of the original images. The average compression ratio of our algorithm is of approximately 1.12 : 1 and the compression percentage is 12%.

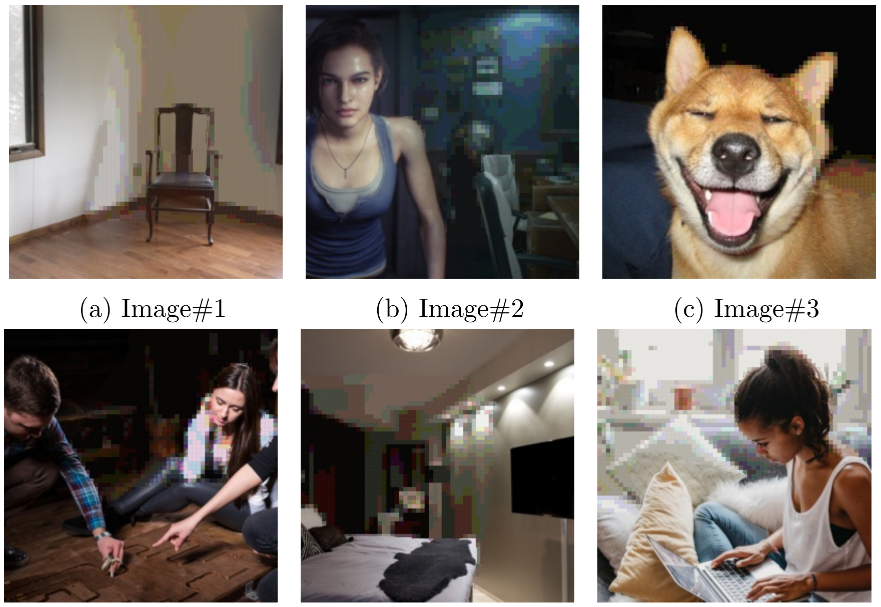
	Uncompressed	Compressed
Image#1	192 kB	161 kB
Image#2	192 kB	173 kB
Image#3	192 kB	180 kB
Image#4	192 kB	169 kB
Image#5	192 kB	167 kB
Image#6	192 kB	191 kB
Mean	192 kB	173.5 kB

The figures 4 and 5 allow for the comparison of original images and decompressed images (using our algorithm), illustrating the visible loss of information on the background:


(a) Image#1 (b) Image#2 (c) Image#3

(d) Image#4 (e) Image#5 (f) Image#6

Figure 4: Original Images



(a) Image#1 (b) Image#2 (c) Image#3

(d) Image#4 (e) Image#5 (f) Image#6

Figure 5: Decompressed Images, using Portrait Mode Decompression

5 Result Analysis

Let us start by analysing the compression algorithm we were finally able to come up with. We will also state here that this algorithm is technically a lossy compression algorithm, since there is some loss of information, even though it is a local loss.

If we compress and decompress the entirety of the image with the Linde-Buzo-Gray algorithm, it is because of its nature: indeed, the fact that it compresses entire blocks of pixels into one value in the compressed image prevents us from compressing and decompressing the image pixel-wise with this algorithm, as we can do with the Huffman encoding or even with the Lempel-Ziv-Welch algorithm.

This algorithm relies on the fact that patterns found in the background of an image differ from patterns found in the foreground. Indeed, in our compression algorithm, the vector that is being encoded through Huffman encoding algorithm only contains the patterns and values found in the foreground of the image, which should make the produced codec smaller and therefore the resulting encoded vector smaller too, than if the original vector was containing all the pixel values from the whole image. This is what allows it to be potentially more interesting than a simple lossless compression using Huffman encoding.

So this means that to be able to analyse our algorithm correctly, we need to compare the size of our compressed files with the size of the compressed files obtained with a lossless compression algorithm using Huffman encoding. Indeed, for our algorithm to be an interesting lossy compression algorithm using Huffman encoding, it has to be better than using only Huffman encoding. The following table represents this comparison:

	Uncompressed	Compressed using Huffman encoding	Compressed with our algorithm
Image#1	192 kB	154 kB	161 kB
Image#2	192 kB	165 kB	173 kB
Image#3	192 kB	172 kB	180 kB
Image#4	192 kB	161 kB	169 kB
Image#5	192 kB	160 kB	167 kB
Image#6	192 kB	183 kB	191 kB
Mean	192 kB	165.8 kB	173.5 kB

It becomes clear here that our lossy compression algorithm, though producing compressed files smaller than the size of the original image, produces bigger compressed files than an algorithm using only Huffman encoding, which is a lossless compression algorithm.

As stated before, we based this algorithm on the fact that the codec and encoded vector created using only the data from the background of the image would be of smaller size (in Bytes) than the codec and encoded vector created using the data from the whole image. What we can see here, is that it seems that we have over-estimated the strength of that effect, since we were expecting the compressed file from our algorithm to be smaller than the compressed file produced with Huffman encoding.

The impact on the size of the compressed file of the binary image, as well as the impact of the codebook and compressed image from the Linde-Buzo-Gray algorithm, also have to be considered, even though they probably represent a smaller part of the data contained in that file.

It means that for further optimization of our algorithm, we might need to consider using a lossless compression algorithm on the binary image before saving it. We may also consider using another lossy compression algorithm to apply to the background of our image. Indeed, as it has been stated before, the main flaw of the Linde-Buzo-Gray algorithm in the context of our project is the fact that we can not easily compress only some pixels of the image using that method. We might want to consider using another lossy compression algorithm with which such a local application of the method would be possible. Discrete Cosine Transform (DCT) compression, though computationally heavier, might be a good replacement for the Linde-Buzo-Gray algorithm.

6 Conclusion

We designed a depth-wise compression algorithm, that can be applied to Portrait Photos, and that we call "Portrait Mode Compression". The main success criteria has been reached, that-is-to-say the produced compressed files are smaller in size than the original images. But a more in-depth analysis of our results, as stated before, shows that our algorithm is actually producing bigger compressed files than the Huffman encoding used alone. So even though the algorithm has been designed the way we wanted to, the way we described it in the Problem Formulation part, it still clearly needs to be optimized.

At some point during our work we had made an informal hypothesis, which was that applying the Huffman encoding algorithm on only a part of the images was going to reduce the size of the codec and of the encoded vector enough for the resulting compressed file obtained with our algorithm to be smaller than the compressed file obtained by applying the Huffman encoding to the whole image. This informal hypothesis has been invalidated by our results. It may seem disappointing, but it is still an interesting conclusion, as we understand more about the algorithms we used.

During this project, we learned about several different compression algorithms, but we also learned about their limits and why they were mainly applied on some specific files, especially concerning the Lempel-Ziv-Welch algorithm. We were also able to experiment with these compression algorithms, and understand first-hand why and how their different parameters influence the result of the compression (and the result of the decompression).

Regardless of our seemingly disappointing compression ratio, we believe that depth-wise compression algorithms such as our Portrait Mode Compression algorithm are relevant, as they have many applications from storage to application in the Medical Field. Moreover, designing new depth-wise compression algorithms like this one imply the use of methods from various areas of computer science, including Data Compression, AI, and Computer Vision, which makes this subject even more interesting.

7 Work Contributions

Below is a list describing the contributions of each team member to the work that has been put into making this report:

- Elie Saad: Was responsible for the entirety of the theoretical background of the different algorithms and their description.
- Matthieu Dabrowski: Was responsible for the entirety of the experimentation of the different algorithms described.
- Elie Saad and Matthieu Dabrowski: Were both responsible of the implementation of the solution to the given problem formulation, the analysis of the obtained results from the implementation and the conclusion derived.

References

[1] Nasir Ahmed, T₋ Natarajan, and Kamisetty R Rao. Discrete cosine transform. *IEEE transactions on Computers*, 100(1):90–93, 1974.

[2] Ms Asmita A Bardekar and Mr PA Tijare. A review on lbg algorithm for image compression. *International Journal of Computer Science and Information Technologies*, 2(6):2584–2589, 2011.

[3] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[4] W Kinsner and RH Greenfield. The lempel-ziv-welch (lzw) data compression algorithm for packet radio. In *[Proceedings] WESCANEX'91*, pages 225–229. IEEE, 1991.

[5] Yoseph Linde, Andres Buzo, and Robert Gray. An algorithm for vector quantizer design. *IEEE Transactions on communications*, 28(1):84–95, 1980.

[6] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[7] Amir R Zamir, Alexander Sax, Nikhil Cheerla, Rohan Suri, Zhangjie Cao, Jitendra Malik, and Leonidas J Guibas. Robust learning through cross-task consistency. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11197–11206, 2020.