DTU Compute
Software engineering

Neural lossy image compression

Bachelors Project

Neural lossy image compression

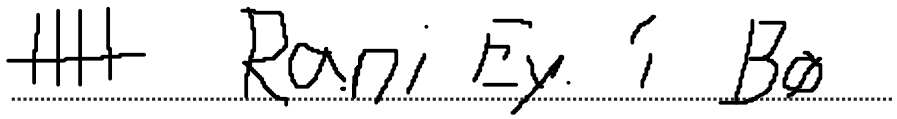
Bachelors Project
Maj, 2023

By
Hans Henrik Hermansen, Rani Eyðfinnsson í Bø

Cover photo: Vibeke Hempler, 2012
Github link: https://github.com/s194042/DeepLearningBachelorProject

Approval

Hans Henrik Hermansen, Rani Eyðfinnsson í Bø - s194042, s194067



Signature

31-05-2023

Date

Abstract

The exponential growth of data transfer and storage since the advent of the internet has created a pressing need for efficient solutions. One approach is to develop advanced infrastructure, but this is resource-intensive. As a result, compression has emerged as a crucial field with potential for significant advancements. While handcrafted algorithms have traditionally dominated compression techniques, the recent success of AI methods raises the question of their applicability in image compression. In this project, we delve into the realm of lossy neural image compression by constructing our own auto encoder model and entropy encoder for compressing images. Additionally, we propose a novel solution to a fundamental problem in the field—the loss function—by training a convolutional neural network to align with the human visual system. While our auto encoder did not surpass the performance of the traditional JPEG method, the results demonstrated promising potential, suggesting that with further training or a simplified architecture, superior performance could be achieved. Moreover, our developed loss function exhibited better alignment with the human visual system compared to other commonly used loss functions. However, we encountered issues with our loss function that enabled our compression model to exploit it and achieve near-zero loss. Although we did not resolve these issues, we present potential solutions for future improvements. In conclusion, AI methods possess significant potential to revolutionize compression, as they have done in various other domains. Nonetheless, there are numerous open problems that must be addressed before this transformative impact can be fully realized.

Contents

1 Introduction

In this current age, most of us send and receive tons of data trough our phones, laptops and televisions. Even some watches, lighting and more unconventional data devices do use data and data communication to perform tasks. It is also getting more common to use large amounts of data to perform tasks. This immense amount of data requires great connectivity to be transferred and a lot of storage space to be stored. These requirements are not without obstacles or cost.

Remote locations or locations in poorer areas can suffer from lack of connectivity infrastructure, making it slow and cumbersome to transfer data. Furthermore, facilitating great connectivity infrastructure is very costly, and there are a lot of other pressing issues which could benefit from further funding. Even when in an area with good connectivity, it can take a long time to send large files, and many data transfer methods have caps on how large a file can be (for email it is often in the range of 20-30mb).

All this data must also be stored somewhere. If it is stored locally on a device, larger requirements are set on the local storage space which can be quite costly for the consumer. Many devices also run slower and consume more energy when more local storage is used. Data that is stored in the cloud can be stored more efficiently by utilizing the storage space to its fullest, and by not having copies of data on multiple devices. However, cloud data has to be transferred in order to be used, bringing with it the problems of data transfer.

One way to lessen the burden of transferring and storing data, is by means of compression. By making the files smaller, one can lessen the storage and the connectivity requirements, thereby saving on costs and or providing a more responsive user experience. Costs and user experience are key for companies that need to maintain a competitive advantage, and to other entities for other reasons (two examples being funding or functionality).

One of the most common large files are images. Most people frequently use phones and other devices to take and share images trough platforms like Snap-Chat, Instagram and Facebook. Thus better image compression schemes would likely be of benefit to many.

Neural network based approaches have in recent years reached state of the art image compression results[1], and there are still open questions in the field[1]. This is the reason for our interest in the field and our choice of this project, where we compress images using neural networks.

We will show how our train of thought lead to the choices regarding methods, model architecture, training method and more. Resulting in us training two neural networks, one convolutional auto encoder that compresses the images into a small latent layer, and one convolutional neural network that estimates the qualitative difference between an image and the reconstruction of the same image. Furthermore we also make use of entropy coding to further compress the latent layer of the convolutional auto encoder and implement the commonly used image compression method JPEG, to compare and benchmark our method.

2 Information Theory: what is compression?

Most likely everybody has at least an abstract idea of what compression is. That it can take some data, in our case image files, and make it smaller. However in order to move from the abstract to the concrete we need information theory. Information theory provides us with a way to describe and discuss information in a methodical and quantifiable way. It defines various important measures of information like entropy and redundancy, which we will explain shortly.

In general there are two kinds of compression, lossless and lossy compression. The names are rather fitting as lossless compression is compression without any loss of information, like for example PNG image compression[2], whereas lossy compression intentionally discards information in order to achieve greater compression rates.

Lossless compression is deeply tied to information theory and we will therefore combine the presentation of both these subject in the following section.

2.1 Lossless compression

As mentioned, lossless compression is comprised of methods of compression that compresses a source without any loss of information, which allows for perfect reconstruction of the original source. But what does no loss of information mean? How do we measure information?

2.1.1 The bit

In his paper, "A Mathematical Theory of Communication", [3] Claude E. Shannon defines the most simple type of information, the "bit", as the base information unit for information theory and analysis. Most people are familiar with the bit, but we find it important to reintroduce it from the perspective of information theory as a measure of information. The bit can be seen as the smallest amount of discrete information we can have. A bit is either "Yes" or "No", "0" or "1" et cetera. Essentially it represents two states. If it is needed to represent more than two states, this can be done with $\lceil \log_2(states) \rceil$ bits, as each bit we have available doubles the number of states we can represent. This way we can turn any discrete set of states into a bit representation. As such it is a suitable choice for a unit of information.

2.1.2 Compression as communication

At its origin information theory was developed as a study of communication systems and as such, much of the important theory is closely tied to this perspective. We will therefore show how to state the compression problem as a communication system in order to seamlessly apply information theory.

In his paper Shannon presents three overall types of communication systems namely discrete, continuous and mixed. Compression can be seen as a discrete version of Shannon's general communication model seen below in figure 2.1:

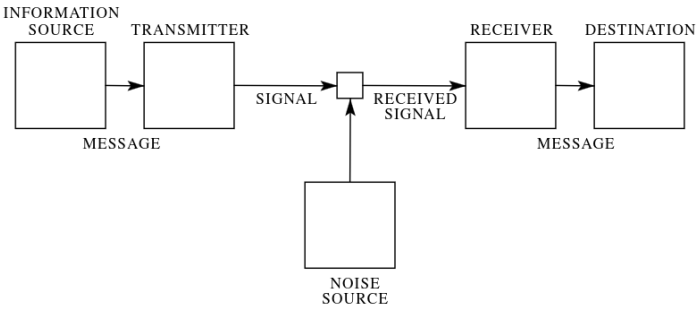


Figure 2.1: General communication model presented by Shannon[3]

In our case we can reformulate the components to be the following:

- *Information source*: Source or data to compressed. This is more or less the same as before.
- *Transmitter*: This step encodes the original source into some new compressed representation
- *Signal and received signal*: The compressed representation of the original source.
- *Noise source*: Not relevant for our case.
- *Receiver*: This step decodes the compressed representation back into the original source for lossless compression or something close to the original source for lossy compression.
- *Destination*: Reconstruction of the original source

We can now present our somewhat simplified and less general model for our use-case in figure 2.2:

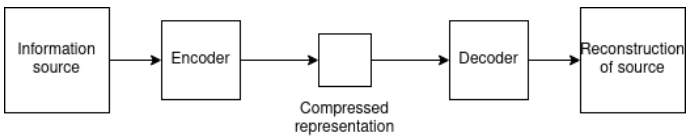


Figure 2.2: Compression as an communication model

This model illustrates the general process of a compression method for both lossless and lossy methods as a communication system. We can now apply information theory to our problem directly.

Information source
From information theory we define an information source as a set of discrete random variables indexed by time, also known as a stochastic process, $\{X_t\}\, t \in T$, where each variable can assume values from a set of symbols $\Sigma = \{s_0, s_1, \ldots s_n\}$ each with an associated probability from the set $\Pi = \{p_0, p_1, \ldots p_n\}$, where naturally $\sum_i p_i = 1$. From this stochastic process we can produce messages of any given length N. This stochastic process functions as a model of our data/source assuming our source is memoryless. However many information sources are not memoryless. Take for example the English language. If we produce the symbol "T" this would increase the probability of the next symbol being "H". Such dependencies can not be modelled by our current model. Shannon shows how we

can expand our model by defining transition probabilities where the probability of each symbol is defined by the symbols before it. The simplest case is where each symbol only depends on the symbols just before it resulting in the transition probabilities $p_i(j)$ where $p_i(j)$ is the probability that letter j follows symbol i in the message. This can be extended to have each letter depend on the last m letters by adding more and more sets of transition probabilities from $p_{i,j}(k)$ until $p_{i_1,i_2,\ldots,i_{m-1}}(i_m)$. This way it is possible to model any information source of arbitrary complexity by simply defining enough transition probabilities.

It might not initially make intuitive sense to view an information source as a random process as it doesn't make sense to simply send a message of random symbols, but it only takes a slight change in perspective and it becomes rather intuitive. When receiving a message over a channel we typically have no idea what the first symbol might be, so from the receivers perspective it is essentially random and if we have no knowledge about the source, that will continue to be the case as symbols appear. If we know the message is in English, then we can expect some symbols to be more likely to appear like "e" or "t" compared to "q" or "z". This is however still random, in the sense that we can't be sure what symbol is next, but not random in layman's terms, where each symbol appears with a uniform distribution and the symbols are i.i.d.

It is also worth mentioning that a symbol can be whatever one defines it to be. If we return to the English language example, we might choose words as our symbols instead of letters. This might make more sense as we derive meaning from the words of a language and not the letters individually. If we wanted to, we could even define symbols as entire sentences. This however is a great example of the trade off to be aware of when choosing the symbols for ones model. The size of ones model greatly depends on the choice of symbols. As there in the English language are an infinite number of possible sentences, we would need an infinitely large model to model the probability that each sentence occurs. Whereas there are around 600.000 words in the oxford dictionary[4], which is a far cry from infinite, but far more than the 26 letters in the English alphabet. So it is critical to be mindful of model size when choosing symbols, especially in our case, as we also need to save the model when doing entropy encoding, in order to know how to reconstruct our data. This will be further presented later.

Entropy

Given a model for an information source, we want a measure of how much information is produced by the source. Essentially we want a measure of how much information is inherent to the source. This is especially important for lossless compression as this can help provide a lower bound for how much something can be compressed. Shannon also provides this in his paper[3]. This measure is called entropy, noted as $H(X_t)$ and measures how many bits per symbol we can expect to need to represent a message, or how many bits of information is carried by each symbol. The entropy of a random variable at index t, X_t, is defined as follows:

$$H(X_t) = \sum_i \sum_j P_i \cdot p_i(j) \cdot \log_2(p_i(j)) \text{ where } 0 \cdot \log_2(0) \text{ is taken to be zero}$$

Where P_i is the probability of being in a state i and $p_i(j)$ is the probability of the next symbol being j when in state i. Firstly to illustrate how the entropy measure behaves, we present an example from Shannon's paper. Assume our symbols to be $\{0, 1\}$. We define the probabilities of each symbol appearing to be $\{q, 1-q\}$ and define each symbol in the message to be independent of any before it, and as such we have no transition probabilities.

In this case it is important to explain that $P_i = 1$ since there is no dependency on the state, and therefore we are in a sense always in the same state. Now we can plot the entropy as a function of q as in figure 2.3:

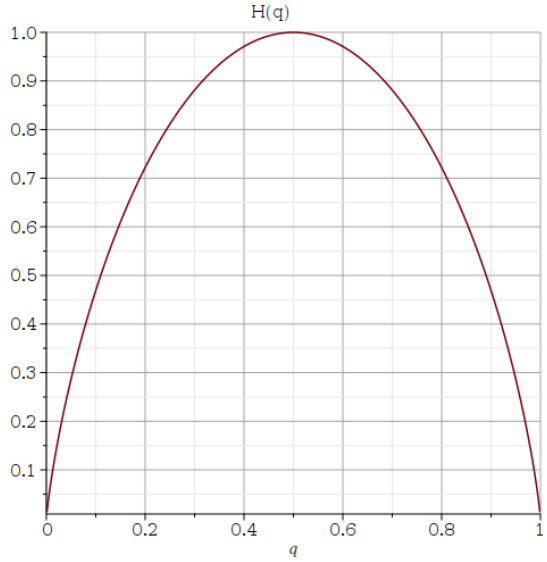


Figure 2.3: The entropy of an information source with symbols $\{0, 1\}$ as the probabilities are shifted

From this plot we see clearly that the entropy is largest when the probabilities are uniformly distributed and decreases as one of the symbols becomes more likely to appear. This perfectly portrays the behaviour of entropy. For a given information source with N symbols, the maximum possible entropy is achieved when the probabilities for each symbol appearing is uniformly distributed and independent. When the probabilities are unevenly distributed entropy decreases. Intuitively we can understand this as symbols, that are more likely to appear, provide less information. If one were to throw a truly random coin a bunch of times and note a "0" for each time it landed tail and a "1" for head, we would need the entire sequence of 0's and 1's in order to perfectly represent the tosses. However if we know that the coin lands on head 99% of the time, we could naively just write down the index of each time the coin landed on tails and perfectly recreate the sequence of throws. Even though to note the index of each throw requires $\lceil \log_2(N) \rceil$ where $N =$ number of throws, it is still clear that we can save bits this way simply due to the infrequency of the coin coming up tails[1]. This is a very simple example of lossless compression. In our example, the coin coming up heads would carry essentially no information, whereas the coin coming up tail would carry around $\log_2(N)$ information. This of course is not a rigorous interpretation of the situation, but serves as good intuition.

When our information source is memoryless we can use the entropy of any X_t as the entropy rate, r, of our source. However when we need to calculate the entropy rate of a more complex information source, we need to consider the state, i.e. the symbols that came before. This is quite a bit more complicated as we now need to track the state for each symbol. Even when the model is only dependent on the last symbol, the state is dependent on all symbols before it. The calculation of $H(X_t)$ depends on X_{t-1} which in

[1]This is only works for $N < 2^{100}$ but as that is more than the total amount of coin tosses made by the human species we deem it an acceptable flaw of this compression method. There are of course better methods, but it serves as good example

turn depends on X_{t-2} and so on. Therefore the entropy of a given X_t is not the same for all t. As a result the entropy rate can be described by:

$$r = \frac{1}{N} \cdot \sum_t H(X_t)$$

For our purposes we will be using simple memoryless models, but in general models with more dependency have a lower entropy than models without. This is due to the fact that maximal entropy is when the symbol probabilities are i.i.d. and follow a uniform distribution which is never the case for models with dependency. Dependency only makes sense if the probabilities change based on the previous symbols, and if the probabilities change they can no longer be uniform.

Redundancy and Shannon's source coding theorem
Redundancy[5] is the measure of how much excess information is used to transmit a message, or more formally it is the difference between the bitrate used and the entropy rate of the source. In terms of compression, redundancy puts a limit on the maximum achievable compression rate. There are two kinds of redundancy: absolute redundancy and relative redundancy. They can be expressed as follows:

- r: The average entropy per symbol of the source. The entropy rate.
- R: The number of bits used per symbol to transmit the message. The message bitrate.
- $D = R - r$: The difference between the two rates. The absolute redundancy
- $\frac{D}{R}$: The relative redundancy. Provides a strict upper bound for the compression rate, when taken as a percentage of how much of the data can be compressed away.

Using the following model as an example:

- Σ: $\{A, B, C, D\}$
- Π: $\{0.5, 0.1, 0.1, 0.3\}$

We find $r = 1.685$, and with four different symbols we take $R = 2$. Therefore we have $D = 0.315$ and $\frac{D}{R} = 0.158$ and it will be impossible to compress the data to less than 84.2% of its original size without loosing information.

So redundancy provides an easy way to know the limit of compression for a source, but we are also interested in how much it is possible to compress a source. These are different. Redundancy tells us that is impossible to compress more than the relative redundancy, but not that it is possible to reach that level of compression. Luckily we can also look to Shannon here. For our purposes which is memoryless sources we can use Shannon's source coding theorem.

> For N i.i.d. random variables, each with entropy $H(X)$, it is possible to compress them into $N \cdot H(X)$ bits with negligible risk of loosing information as $N \to \infty$. If they are compressed into fewer than $N \cdot H(X)$ bits it is virtually guaranteed that information will be lost[6]

So for large messages we can expect to be able to approach the limit expressed by the relative redundancy.

2.2 Lossy compression

Lossy compression, also sometimes called irreversible compression, is compression where information is lost and as a result a perfect reconstruction of the original source is not possible. Instead a reconstruction of the source that is hopefully close enough to the original is made. By throwing away information lossy compression enables us to go beyond the limit stated by redundancy and achieve even better compression rates. Methods of lossy compression do this by attempting to throw away unimportant information. This leads us to the main problem of lossy compression: what is unimportant information?

2.2.1 Redundancy vs. Perception

As established lossless compression works by removing redundant information until only what is needed to recreate the source is left. So in a sense lossless compression defines important information as anything needed to reconstruct the original source. This however does not take perception into account. This can be fine if higher compression rates are not essential and one does not know the best way to perceive ones data, but with for example images we humans have an intuitive perception of what is important for the structure of an image, even if we have a hard time describing that perception. This is something that algorithms like JPEG take advantage of in order to throw out information not important to our perception of the image. Some intuition behind lossy compression might be simply noting that a part of an image is grey, rather than what exact shade of grey it is. In general all lossy compression can be expressed as an optimization problem of the form:

$$\min R + \lambda D$$

where R is the bitrate, D is the distortion and λ is some chosen weight for the distortion.

The way of measuring D or defining what information is important depends heavily on the method of compression and the type of data that is being compressed, as it requires thorough understanding of the data and especially whoever or whatever will perceive the data. This is an important difference between lossless and lossy compression. What we have covered about lossless compression is general and as a result most lossless compression methods can be applied somewhat indiscriminately. Of course there can be differences in the effectiveness of the applied methods, but the original source can always be recovered. With lossy compression choosing the wrong compression method can be catastrophic and lead to serious degradation of the data.
Many lossy compression methods, like JPEG, compress the data into a smaller latent space which then is entropy encoded. This is also the method for our model.

3 AI and Information Theory

3.1 Transition probabilities/ dependencies

As previously stated, if one can assume that the information source consists of i.i.d discrete variables, then one can compress them efficiently by using entropy coding. However, just by looking at one of the images from the data set together with a random sampling from the same image, it is clear that there are dependencies among pixels.

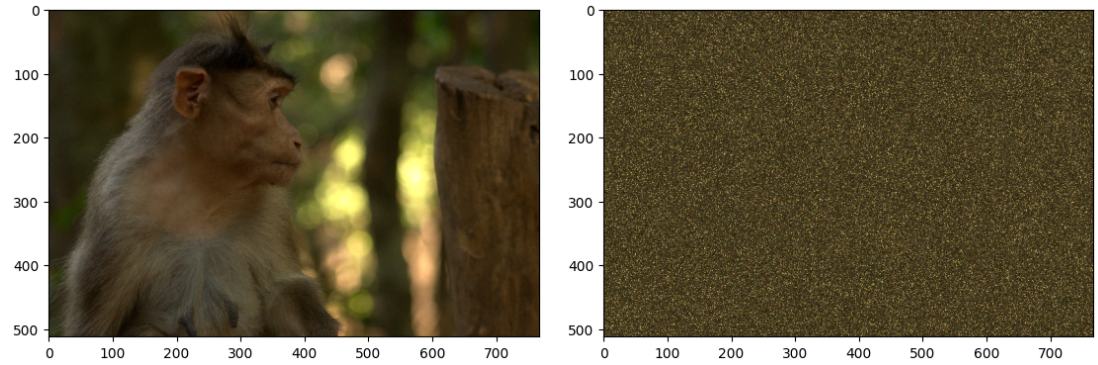


Figure 3.1: Image of a monkey and a random sample of pixels from the same image

Furthermore, the set of all images also imposes some dependencies on the pixel value distribution which also has to be taken into account. For instance, if we hypothetically assume that the set of all images I only contained two images i_0, i_1, then the information about the set would allow us to encode the entire image down to one bit representing either i_0 or i_1. There are clearly more than two images in I, since we show more than two distinct images in this report. If we calculate the number of combinations of pixels in an 512×768 color image, we get:

$$256^{512 \cdot 768 \cdot 3} = 256^{1.179.648} = 2^{9.437.184}$$

However if one were to take a sample from the uniform distribution of all possible pixel value combinations, it is clear that humans do not recognise it as an image but rather as noise:

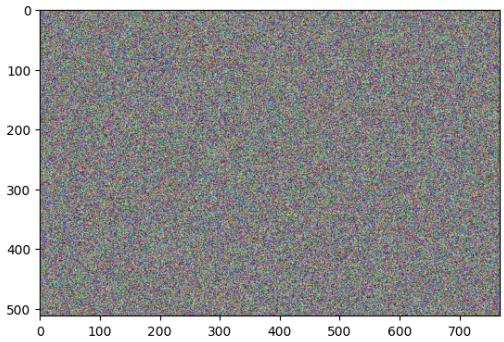


Figure 3.2: Random composition of pixel values

Thus for our purpose, we define the set of images I as the subset of the set of pixel combinations Π, that humans recognise as an image and not noise.

If we would sample from Π 256^3 times without getting a composition resembling an image, then we would have shown that the number of images possible in the format 512×768 is no more than:

$$256^{512 \cdot 768 \cdot 3 - 3} = 256^{1.179.645} = 2^{9.437.160}$$

One could thus save one full pixel just using this knowledge of the distribution of images. In other words, we would have showed that the set of all images exists in a lower

dimensional manifold of the set of all combinations of pixel values (this would in reality be proven well before, since after e.g. 256 images one could already save one channel of one pixel, and before that one bit of one channel of one pixel). It turns out that it is quite common for high dimensional data to be situated on a lower dimensional (possibly highly non linear) manifold [7]. In our case, we do not know the size of the manifold, but auto encoders are generally capable of learning an approximation of the lower dimensional data manifold[7][8]. Furthermore auto encoders seem capable of figuring out spacial dependencies between pixels[9].

Thus it seems like one can let the auto encoder figure out the probabilistic dependencies, and if the auto encoder performs well while the bottle neck layer is small enough the parameters in the bottleneck would likely be approximately independent since the auto encoder would otherwise be using space inefficiently and the bottle neck could therefore be made even smaller. If the parameters in the bottleneck layer are approximately independent, we can straightforwardly use information theory to further compress the latent layer of the auto encoder. However techniques like entropy coding require discrete values while auto encoders need continuous values to facilitate gradient decent. One could transform the continuous latent layer to discrete values by rounding, but the non linear nature of auto encoders would likely produce poor results when rounding.

To solve the compatibility issue of auto encoders and entropy encoding, we look at the act of rounding. For any number N the act of rounding consist of either subtracting or adding another number n, where n is between 0 and 0.5. This is because the distance from any number N to the nearest whole number is at most 0.5. Thus if the auto encoder is robust to noise of magnitude 0 to 0.5 in the latent layer, then the auto encoder is also robust to rounding. If the model would converge well during training with such noise added to the latent layer, then the model would essentially be compatible with entropy encoding. Similar approaches have been done before with good results, one example being [10].

Since neural networks are general function approximators, one could also try to avoid entropy encoding by having a deeper network with an even smaller bottle neck layer, since the network could learn entropy encoding or possibly something better. However, since entropy coding is proven to be close to optimal assuming the latent parameters are close to i.i.d, it seems to be the better option. Furthermore, having a deeper network with a smaller bottle neck makes the network harder to train and more computationally intensive both under training and in deployment. Therefore it is our assessment that entropy encoding of the latent layer is the better choice.

3.2 What data should we lose?

If one looks at the set of possible images I, there are many that humans would not notice are different, especially if the images are not compared side by side. Not being able to see the images side by side is the usual case for compression, since people usually only see the final compressed image. Four very similar but distinct images are shown below:

Figure 3.3: Four different but very similar images

Under the assumption that these images are meant to be viewed by humans, one would think that humans would not care which one they saw. As a result one can further reduce the dimensionality of the manifold by only keeping one image for each set of images that humans can not distinguish. Under the assumption that the loss function used to train the auto encoder judged the reconstruction like a human would, then it would not care if the auto encoder made some human unnoticeable mistake. Assuming that the auto encoder has a small bottle neck layer, then it is forced to throw away some information. Because of the loss function, the auto encoder would prefer to throw away these unnoticeable differences, since it does not lead to a higher loss. Thus if training goes well and the loss function judges like a human would, then the auto encoders could exploit the human like judgement of the loss function to further reduce the dimensionality without decreasing perceived quality.

4 Arithmetic Encoding

For both our model and for our JPEG implementation we will need a way to perform entropy encoding. One of the most effective methods for entropy encoding is arithmetic encoding[11]. When compressing an information source with no dependencies, i.e. the variables are i.i.d, arithmetic encoding can get very close to the theoretical limit of a message. As we described in section 3 we think/assume that our neural model will create a latent space representation where dependency is low, which then should result in arithmetic encoding being very well suited to further compress this latent layer.

4.1 The Infinite Precision Algorithm

Arithmetic encoding works by assigning every possible sequence of a set of symbols to its own interval between $[0, 1[$ and then the compressed message is a binary fraction precise

enough such that it is wholly within the message interval. The size of an interval associated with a sequence is equal to the probability of that message occurring given the model. Therefore sequences with a high probability of appearing are associated with a larger interval which results in lower precision and fewer bits needed. In this chapter we will cover how the algorithm works, when we assume that infinite precision is available. This is of course not true in practice, but it provides a helpful intuition and a fundamental insights for understanding the real algorithm. The finite precision case will be covered in the implementation chapter 10.

4.1.1 Defining the model

When defining a model for arithmetic encoding four things are needed. An alphabet $\Sigma = \{s_0 \dots s_n\}$ of symbols that can appear in our sequence, a symbol O from our alphabet defined as a stop symbol, a set $\Pi = \{p_0 \dots p_n\}$ of probabilities associated with each symbol and a method of assigning each symbol to an interval between $[0, 1[$. As explained in chapter 2 the probabilities for each symbol could be dependent on the other symbols in the sequence, which would result in a more complex model. This is something arithmetic encoding can handle but since we have no way to properly know these dependencies, if there are any, we will be using a simple i.i.d. model.

For assigning each symbol to an interval we define the mapping $I(s_i)$:

$$I(s_i) = \left(\sum_{j=0}^{i-1} p_j, \sum_{j=0}^{i} p_j \right)$$

This is the same for all models, so all we need to define to encode a sequence is Σ, O and P.

4.1.2 Finding the interval: Encoding

Given a model the question now is how do we go from message to interval? We define a upper bound u and lower bound l for our interval starting with $u = 1$ and $l = 0$. Then we iterate through our sequence and subdivide our interval by updating u and l depending on the symbol. We update u and l as follows:

$l = 0$
$u = 1$
for s in message **do**
    $(x, y) = I(s)$
    $r = u - l$
    $u = l + r \cdot y$
    $l = l + r \cdot x$
end for

To illustrate this process we will provide an example with the model defined as below:

- $\Sigma : \{X, Y, Z\}$
- $O : Z$
- $\Pi : \{\frac{1}{4}, \frac{5}{8}, \frac{1}{8}\}$
- $I(s_i) : \{X \to (0, \frac{1}{4}), Y \to (\frac{1}{4}, \frac{7}{8}), Z \to (\frac{7}{8}, 1.0)\}$

To encode the message $XYYZ$ the steps are as follows

- 0: Encoded = $\{\emptyset\}$ $\{u = 1, l = 0\}$

$$(x, y) = I(X) = \left(0, \frac{1}{4}\right)$$
$$u = 0 + (1 - 0) \cdot \frac{1}{4} = \frac{1}{4}$$
$$l = 0 + (1 - 0) \cdot 0 = 0$$

- 1: Encoded = $\{X\}$ $\{u = \frac{1}{4}, l = 0\}$

$$(x, y) = I(Y) = \left(\frac{2}{8}, \frac{7}{8}\right)$$
$$u = 0 + \left(\frac{2}{8} - 0\right) \cdot \frac{7}{8} = \frac{7}{32}$$
$$l = 0 + \left(\frac{2}{8} - 0\right) \cdot \frac{2}{8} = \frac{2}{32}$$

- 2: Encoded = $\{XY\}$ $\{u = \frac{7}{32}, l = \frac{2}{32})$

$$(x, y) = I(Y) = \left(\frac{2}{8}, \frac{7}{8}\right)$$
$$u = \frac{2}{32} + \left(\frac{7}{32} - \frac{2}{32}\right) \cdot \frac{7}{8} = \frac{51}{256}$$
$$l = \frac{2}{32} + \left(\frac{7}{32} - \frac{2}{32}\right) \cdot \frac{2}{8} = \frac{26}{256}$$

- 3: Encoded = $\{XYY\}$ $\{u = \frac{51}{256}, l = \frac{26}{256})$

$$(x, y) = I(Z) = \left(\frac{7}{8}, 1\right)$$
$$u = \frac{26}{256} + \left(\frac{51}{256} - \frac{26}{256}\right) \cdot 1 = \frac{408}{2048}$$
$$l = \frac{26}{256} + \left(\frac{51}{256} - \frac{26}{256}\right) \cdot \frac{7}{8} = \frac{383}{2048}$$

- 4: Encoded = $\{XYYZ\}$ $\{u = \frac{408}{2048}, l = \frac{383}{2048}\}$. We have encoded our stop symbol, so we are done.

This process is also shown in figure 4.1:

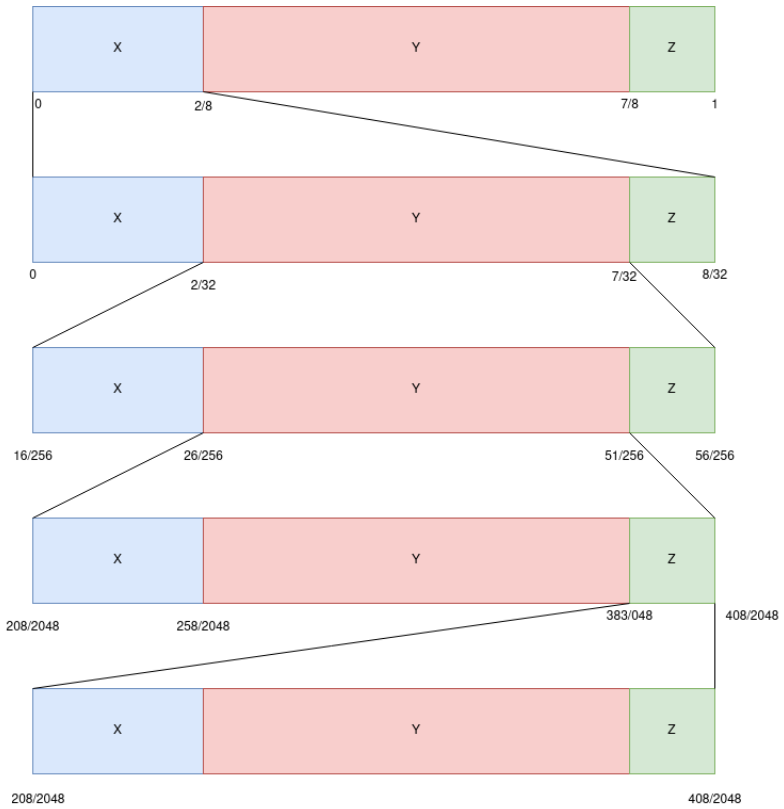


Figure 4.1: Finding the interval for the simple message $XYYZ$

In order to find the bit encoding of this interval we need to find a binary fraction that represents an interval wholly between u and l[12]. We represent this interval with a binary string b representing the lower bound of the interval and the upper bound is equal to $b + 2^{-|b|}$. As a result trailing zeros are necessary to describe the interval. To find b the following procedure is used:

$i = 1$
$b =$ ""
do
  if $float(b) + 2^{-i} > u$ **then**
    b += "0"
  else if $float(b) < l$ **then**
    b += "1"
  end if
while $float(b) < l$ and $float(b) + 2^{-i} > u$
return b

This process is illustrated in figure 4.2. From figure 4.2 we see that our example message is encoded as the binary string 0011000 with a length of 7 compared to the naive encoding length 8. Lets compare this to the expected entropy of the message from the source. The entropy per symbol for our source is very close to 1.3, so from Shannon's source code theorem the expected length would be $1.3 \cdot 4 = 5.2$. So the best we could hope for is a length of 6. There are several reasons as to why we didn't reach the optimal compression length. First Shannon's code theorem is not always applicable and especially not for short messages. Second and more importantly the model from the example is not a perfect fit for the message. A more fitting Π for this message would be $\{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\}$. For this model the

entropy per symbol becomes exactly 1.5, so the expected length is 6. We won't go through the procedure again, but the resulting encoding for this model is 001001 which matches the expected length. This illustrates the importance of having a model that matches the message when using arithmetic encoding.

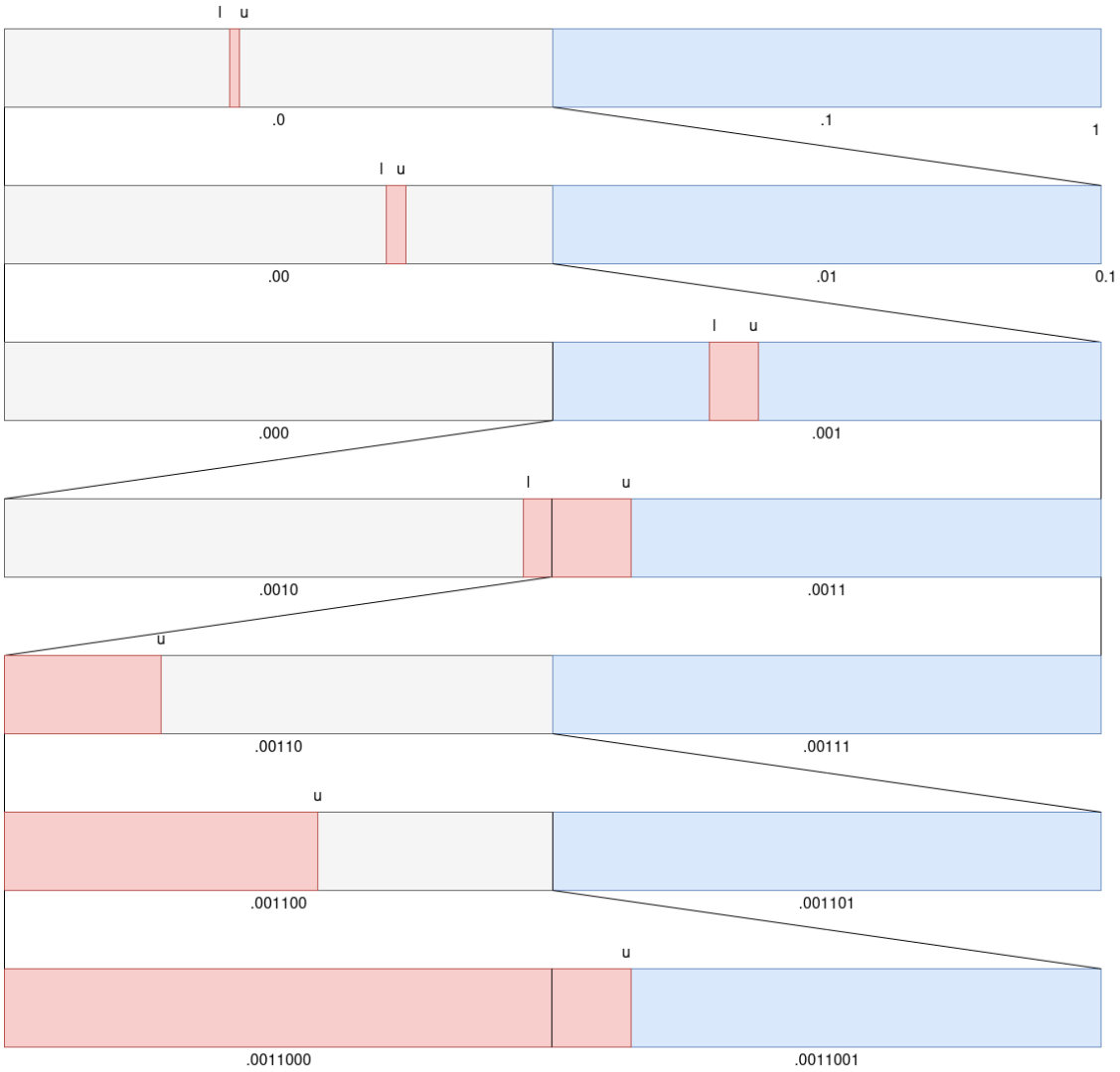


Figure 4.2: Finding the binary interval for the message $XYYZ$

4.1.3 Finding the interval: Decoding

Decoding the encoded message is quite similar to encoding. The difference lies in how to subdivide the interval. In encoding the subdivision was predefined, where in decoding we choose the subdivision that contains the encoded binary fraction. But otherwise we have the same u and l and they are updated in the same manner. To find the right subdivision we define the mapping $S(\beta)$

$$S(\beta) = s_i \mid \{(x, y) = I(s_i) \wedge x \leq \beta < y\}$$

The decoding procedure then is as follows

b = The message to be decoded
$message$ = ""
s = ""

while s is not O **do**
  $r = u - l$
  $\beta = \frac{float(b) - l}{r}$
  $s = S(\beta)$
  $message+ = s$
  $(x, y) = I(s)$
  $u = l + r \cdot x$
  $u = l + r \cdot y$
end while
return $message$

For our 0011000 example the decoding process is illustrated in figure 4.3 below

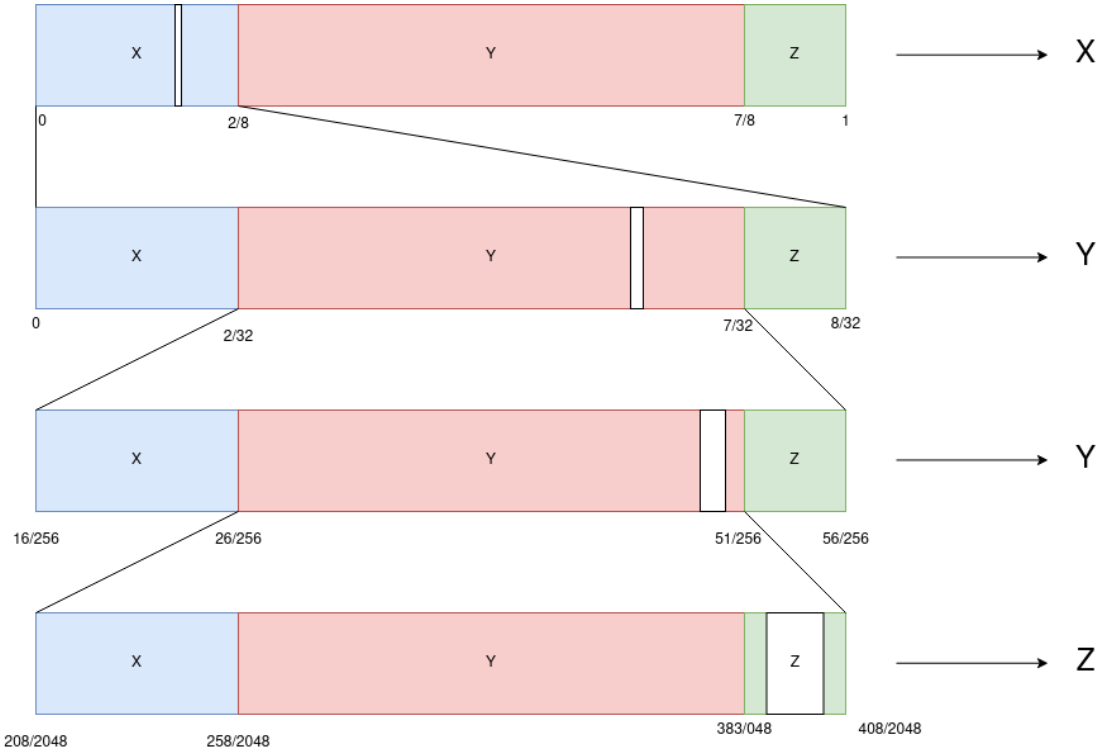


Figure 4.3: Decoding process for ”0011000”

4.2 Effectiveness

In this section we will show that the expected effectiveness of arithmetic encoding is very close to entropy of the a model. We shall do this assuming an i.i.d. model as this is the same assumption we make for our implementation. Firstly we define:

- $\Sigma = \{s_0 \dots s_n\}$. The set of symbols available for the message.
- $O = s_n$. The end of file symbol.
- $\Pi_S(s_i)$. The probability mapping for each symbol.
- $X = \{X_0, X_1 \dots\} \mid X_i = x_0 x_1 \dots x_k s_n,\ x_j \in S \setminus \{s_n\}\ \forall j \in [0, k]$. The set of all possible finite messages ending in s_n.
- $\Pi_X(X_i) = \left(\prod_{j=0}^{k} \Pi_s(x_j)\right) \cdot \Pi_s(s_0)$ The probability of the message X_i appearing.

Notably $\sum_{i \in X} \Pi(i) = 1$

For this general case model we want an estimate for the expected length L of an encoded message. We can express L as $\sum_{i=0}^{\infty} \Pi(X_i) \cdot l(X_i)$ where $l(X_i)$ is the encoded length of X_i. So lets find an estimate for $l(X_i)$. To encode X_i we need to find an m such that we have enough bits to describe an interval wholly between u and l. It is clear that if the binary interval is no more than half the size of $u - l$ then we can easily fit our interval wholly within u and l. Therefore we have $\frac{u-l}{2} \geq \frac{1}{2^m}$. If we rewrite this we have $u - l \geq \frac{1}{2^{m-1}}$. A quite intuitive result we can apply here is that $u - l = \Pi_X(X_i)$. So we get the following:

$$\begin{gathered}
\Pi_X(X_i) \geq \frac{1}{2^{m-1}} \\
\Leftrightarrow \\
\log_2(\Pi_X(X_i) \geq \log_2\left(\frac{1}{2^{m-1}}\right) \\
\Leftrightarrow \\
\log_2(\Pi_X(X_i) \geq -m + 1 \\
\Leftrightarrow \\
-\log_2(\Pi_X(X_i) + 1 \leq m
\end{gathered}$$

To satisfy the above we can choose $l(X_i) = m = -\log_2(\Pi_X(X_i)) + 1$. So now we can express L as :

$$\begin{gathered}
L \leq \sum_{i=0}^{\infty} \Pi_X(X_i) \cdot (-\log_2(\Pi_X(X_i)) + 1) \\
= \\
\sum_{i=0}^{\infty} -\log_2(\Pi_X(X_i)) \cdot \Pi(X_i) + \sum_{i=0}^{\infty} \Pi_X(X_i) \\
= \\
\left(\sum_{i=0}^{\infty} -\log_2(\Pi_X(X_i)) \cdot \Pi(X_i)\right) + 1
\end{gathered}$$

Now lets define a model M where the set X is the source symbols with symbol probabilities defined as $\Pi(X_i)$. Drawing a random symbol from this model carries the same information as the average message from our original model. So the entropy $H(M)$ is a lower bound for the amount of bits needed on average to encode a message from our model. So of course we have $H(M) \leq L$. However due to our definition of M we also have than $H(M) = \sum_{i=0}^{\infty} -\log_2(\Pi_X(X_i)) \cdot \Pi_X(X_i)$. Finally we have:

$$\begin{gathered}
H(M) \leq L \leq \sum_{i=0}^{\infty} -\log_2(\Pi_X(X_i)) \cdot \Pi_X(X_i) + 1 \\
\leftarrow \\
H(M) \leq L \leq H(M) + 1
\end{gathered}$$

So on average arithmetic encoding encodes a message to length within one bit of the

entropy.

5 JPEG: How the world compresses images

As our goal is to use AI methods to create a lossy compression method for images we will use one of the most popular methods for image compression as our benchmark. JPEG is a compression standard created in 1992 by the Joint Photographic Experts Group, from which it also gets its name. JPEG is not only one of the most used compression methods in the world[13], but it is also well suited for benchmarking purposes. The JPEG standard enables variable rate compression by altering a parameter described as the Quality factor, Qf, in order to increase or decrease the amount of information thrown away. The publicly available JPEG implementations we could find, only had ~ 3 quality settings. Therefore we will implement JPEG ourselves, so we can adjust the quality factor ourselves to properly match JPEG compressed images to our neural network results.

The JPEG algorithm consists of six major steps. A color space transform, sub-sampling, the 2D discrete cosine transform, a quantization step, a run-length encoding step and finally entropy encoding. There are several different possible methods of entropy encoding. JPEG usually uses Huffman coding, but since we will implement arithmetic encoding for our neural network model, this is also our choice for our JPEG implementation. Therefore we will only cover the other steps of JPEG in this section.

5.1 Color space transform

RGB images have three color channels for each pixel, namely red, green and blue. Humans are far better at distinguishing changes in light values as opposed to changes in color. Therefore JPEG transforms the image to the YCbCr space, where Y is the luminance, or brightness, of a pixel, and Cb and Cr are the chrominance, or color, of a pixel. This allows for different compression rates on the different channels, since the Cb and Cr channels are not as important for our perception of an image, more of that information can be thrown away. The JPEG standard provides a defined way to compute the YCbCr values from given RGB values and the reverse:

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.402 \\ 1 & -0.344136 & -0.714136 \\ 1 & 1.772 & 0 \end{bmatrix} \begin{bmatrix} Y \\ Cb - 128 \\ Cr - 128 \end{bmatrix}$$

5.2 Sub-sampling

After the color transform, sub-sampling is performed on the Cb and Cr channels. Sub-sampling simply means choosing a single value to represent one or more values around it. As the Y channel is the most important sub-sampling is never performed on that channel. JPEG uses three different levels of sub-sampling of the format j:a:b. These are 4:4:4, 4:2:2 and 4:2:0. These can be seen in figure 5.1. These examples are quite drastic for illustrative purposes. In reality color values next to each are often very close numerically.

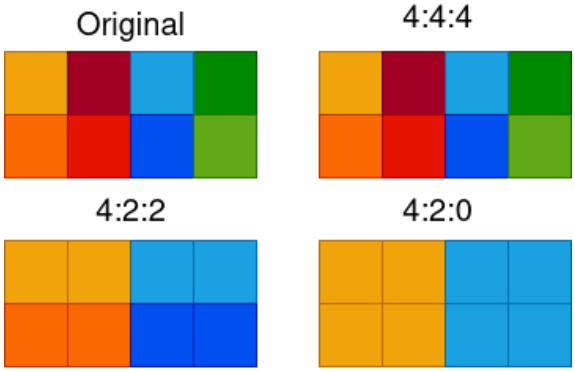


Figure 5.1: How the different JPEG sub-sampling methods affect groups of pixels

Sub-sampling 4:4:4 is the same as no sub-sampling and keeps all the color data, whereas 4:2:2 and 4:2:0 throw away 50% and 75% respectively. When decoding the sub-sampled value is simply copied.

5.3 The Discrete Cosine Transform

A process used by many different methods of compression is the discrete cosine transform. This takes N data points and transfers them to the frequency domain. This is done by transforming the original N data points into N cosine coefficients for cosines with increasing frequencies. The N data points can then be represented as a weighted sum of N cosine functions. The discrete cosine transform is, as mentioned, used for many different purposes and several different version of the process exist. For our purposes we will focus on the one used by JPEG and how it aids in compression.

5.3.1 DCT II, Blocking and Multidimensionality

The most commonly used discrete cosine transform is the DCT II, more commonly simply called the DCT. For N data points $\{x_0 \dots x_{N-1}\}$ the DCT II coefficients $\{X_0 \dots X_{N-1}\}$ are defined as:

$$X_u = \alpha(u) \sum_{i=0}^{N-1} x_i \cos\left(\tfrac{2i+1}{2N} u\pi\right)$$

$$\text{where } \alpha(x) = \begin{cases} \frac{\sqrt{2}}{4} & \text{if } x = 0 \\ \frac{1}{2} & \text{otherwise} \end{cases}$$

If we examine this a bit closer we can express X_u as a vector dot product. Then the elements of the vector $\mathbf{X}$ are results of dot products, which means that the vector $\mathbf{X}$ is actually just a result of a vector-matrix product. This we can express as:

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{bmatrix} \qquad \mathbf{c}_u = \alpha(u) \begin{bmatrix} \cos\left(\frac{1}{2N}u\pi\right) \\ \cos\left(\frac{3}{2N}u\pi\right) \\ \vdots \\ \cos\left(\frac{2(N-1)+1}{2N}u\pi\right) \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} | & | & & | \\ \mathbf{c}_0 & \mathbf{c}_1 & \dots & \mathbf{c}_{N-1} \\ | & | & & | \end{bmatrix} \qquad \mathbf{X} = \mathbf{x}^{\mathrm{T}}\mathbf{C}$$

So the DCT II is simply a linear transformation. An interesting result we won't prove but simply use is that the columns of $\mathbf{C}$ are all orthogonal to each other and as result $Det(\mathbf{C}) \neq 0$ for all cases. Multiplying each $\mathbf{c}_u$ with $\alpha(u)$ also insures that $\mathbf{C}$ is a orthogonal matrix. The invertibility of the DCT II follows directly from this as we then have:

$$\mathbf{x} = \mathbf{X}\mathbf{C}^{-1} = \mathbf{X}\mathbf{C}^{\mathrm{T}}$$

The inverse of the DCT II is usually called the DCT III or the inverse DCT. The normal DCT II only works on one dimensional data, so JPEG uses the M-D DCT II and M-D DCT III, which are DCT for multidimensional data and are therefore more suited for the 2D data of our image channels. This we will refer to as the DCT and inverse DCT going forward.

For a $N \times M$ matrix $\mathbf{x}$ with data points $\{x_{0,0} \dots x_{N-1,M-1}\}$ the coefficient matrix $\mathbf{X}$ is simply the result of applying the DCT II row wise and then column wise as follows:

$$\mathbf{c}_u = \alpha(u) \begin{bmatrix} \cos\left(\frac{1}{2N}u\pi\right) \\ \cos\left(\frac{3}{2N}u\pi\right) \\ \vdots \\ \cos\left(\frac{2(N-1)+1}{2N}u\pi\right) \end{bmatrix} \qquad \mathbf{q}_v = \alpha(v) \begin{bmatrix} \cos\left(\frac{1}{2M}v\pi\right) \\ \cos\left(\frac{3}{2M}v\pi\right) \\ \vdots \\ \cos\left(\frac{2(M-1)+1}{2M}v\pi\right) \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} | & | & & | \\ \mathbf{c}_0 & \mathbf{c}_1 & \dots & \mathbf{c}_{N-1} \\ | & | & & | \end{bmatrix} \qquad \mathbf{Q} = \begin{bmatrix} | & | & & | \\ \mathbf{q}_0 & \mathbf{q}_1 & \dots & \mathbf{q}_{M-1} \\ | & | & & | \end{bmatrix}$$

$$\mathbf{x} = \begin{bmatrix} x_{0,0} & x_{0,1} & \dots & x_{0,M-1} \\ x_{1,0} & x_{1,1} & \dots & x_{1,M-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N-1,0} & x_{N-1,1} & \dots & x_{N-1,M-1} \end{bmatrix} \qquad \mathbf{X} = (\mathbf{x}\mathbf{C})^{\mathrm{T}}\mathbf{Q}$$

The inverse is simply:

$$\mathbf{x} = (\mathbf{X}\mathbf{Q}^{-1})^{\mathrm{T}}\mathbf{C}^{-1}$$

The DCT has no theoretical limit on N and M, but if these become large, computational complexity and precision become issues. It is not practical to perform DCT on an entire channel at once as each coefficient is dependent on each of the $N \times M$ data points and the time complexity becomes $\mathrm{O}(N^2M^2)$. In theory the DCT is a lossless transformation, but due to limited floating point precision in computers, rounding errors can happen. For example when N or M becomes large, the $\cos\left(\frac{2i+1}{2N}u\pi\right)$ and $\cos\left(\frac{2j+1}{2M}v\pi\right)$ terms become

very small, which can lead to loss of precision and as a result rounding errors occur. Therefore JPEG splits every channel into 8×8 blocks and then performs the DCT on each block individually. If the image dimensions do not divide by 8 then padding of zero valued elements is added until they do. The time complexity then becomes $\mathrm{O}(64 \cdot NM) = \mathrm{O}(NM)$. When performing the DCT, JPEG also subtracts 128 from all values in the channel. This is to decrease the dynamic range of the resulting DCT, which in turn helps with precision.

5.3.2 The effectiveness of the DCT

Representing the data in the frequency domain has several benefits for compression, the main one being that the high-frequency parts of an image signal typically do not contribute very much to the image. To illustrate we will showcase the DCT performed on a grey scaled image from the Kodak data set seen in figure 5.2.

Figure 5.2: Example image from Kodak data set

First we will showcase how a single DCT block signal approaches the true signal when we increase the number of coefficients used starting with low frequencies. This can be seen in figure 5.2.

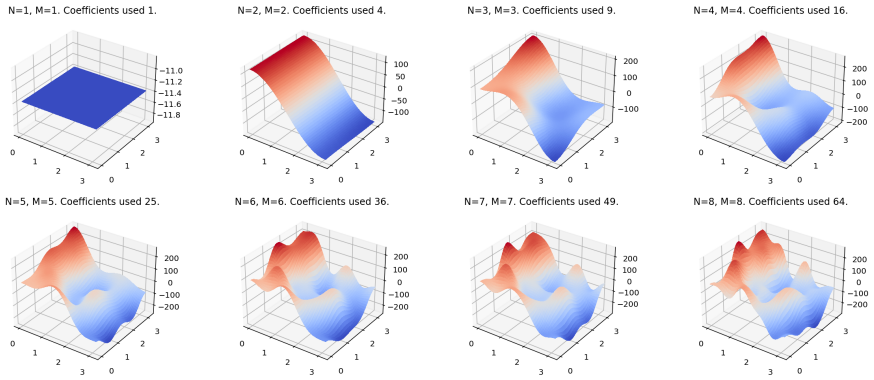


Figure 5.3: The DCT block signal as more coefficients are used

From figure 5.3 we can see that with just 16 out of the 64 available coefficients the signal already has the general shape of the true signal. There are still clear differences that we can distinguish, but it turns out that humans are very bad at noticing the high frequency

parts of an image, so approximating with the general shape of the true signal is often good enough. This is illustrated in figure 5.4. Looking at image 5.4d it is already very hard to distinguish between it and image 5.4h. For images 5.4e, 5.4f 5.4g it is near impossible to spot any differences. Figures 5.3 and 5.4 showcase how the DCT is naturally suited for compression but also how it allows to take advantage of our human perception of images.

5.4 Quantization

So it is possible to keep most of the quality of an image with only a few of the DCT coefficients. The question then becomes what coefficients should we keep? This is where quantization comes in. Quantization is the process of taking a variable which can assume a number of values and mapping it to a space with fewer possible values. Simply rounding a floating point number is an example of quantization. In JPEG quantization is done by dividing each coefficient in a block with a number and then rounding the result. These numbers come from a predefined quantization matrix. JPEG has two standard quantization matrices with $Qf = 50$, one for the Y-channel (Q_Y) and one for the Cb- and Cr-channels (Q_C)[14]. These are:

$$\mathbf{Q}_Y = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix} \quad \mathbf{Q}_C = \begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix}$$

The application of quantization $\mathbf{Q}_Y$ on the DCT block from image 5.2 is shown below:

$$\mathbf{DCT}_{coeff} = \begin{bmatrix} -11.37 & 1.52 & 18.39 & -7.49 & 12.12 & -10.85 & -8.83 & -6.45 \\ 129.15 & -3.02 & 12.79 & 11.43 & -15.50 & -5.39 & -2.00 & -6.63 \\ 24.74 & -59.75 & -25.29 & -10.96 & -11.8 & 8.15 & -2.45 & 4.4 \\ 53.43 & -18.81 & -35.43 & -13.96 & 5.48 & 3.21 & 1.46 & 9.01 \\ -30.62 & 1.18 & 17.27 & 9.61 & 28.87 & 8.42 & 2.18 & 5.17 \\ -3.43 & 16.10 & -35.36 & 2.85 & 19.57 & 8.64 & -5.45 & 4.25 \\ -9.84 & -3.95 & 8.29 & -1.19 & 19.38 & -1.31 & -3.20 & -3.73 \\ -4.60 & -41.99 & 5.39 & 22.82 & 9.68 & -8.69 & -3.33 & -2.65 \end{bmatrix}$$

$$\mathbf{Quant}_{i,j} = \frac{\mathbf{DCT}_{coeff(i,j)}}{\mathbf{Q}_{Y(i,j)}} \qquad \mathbf{Quant} = \begin{bmatrix} -1 & 0 & 2 & 0 & 1 & 0 & 0 & 0 \\ 11 & 0 & 1 & 1 & -1 & 0 & 0 & 0 \\ 2 & -5 & -2 & 0 & 0 & 0 & 0 & 0 \\ 4 & -1 & -2 & 0 & 0 & 0 & 0 & 0 \\ -2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

From this we can see that many of the coefficients become zero due to rounding. This will make the next lossless step of compression much more effective as it practically introduces redundancy that can be compressed by entropy encoding methods.

(a) $N = M = 1$. In total 1 coefficient used.

(b) $N = M = 2$. In total 4 coefficient used.

(c) $N = M = 3$. In total 9 coefficient used.

(d) $N = M = 4$. In total 16 coefficient used.

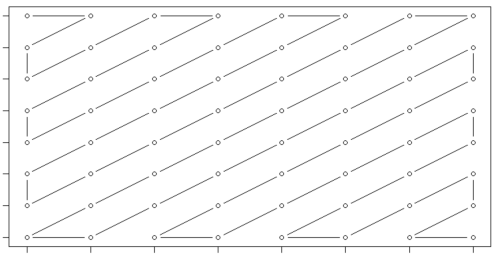
(e) $N = M = 5$. In total 25 coefficient used.

(f) $N = M = 6$. In total 36 coefficient used.

(g) $N = M = 7$. In total 7 coefficient used.

(h) $N = M = 8$. In total 64 coefficient used.

Figure 5.4: Image as DCT coefficients used increases



Line representation of zig-zag pattern

$$\begin{bmatrix} 1 & 2 & 6 & 7 & 15 & 16 & 28 & 29 \\ 3 & 5 & 8 & 14 & 17 & 27 & 30 & 43 \\ 4 & 9 & 13 & 18 & 26 & 31 & 42 & 44 \\ 10 & 12 & 19 & 25 & 32 & 41 & 45 & 54 \\ 11 & 20 & 24 & 33 & 40 & 46 & 53 & 55 \\ 21 & 23 & 34 & 39 & 47 & 52 & 56 & 61 \\ 22 & 35 & 38 & 48 & 51 & 57 & 60 & 62 \\ 36 & 37 & 49 & 50 & 58 & 59 & 63 & 64 \end{bmatrix}$$

Matrix representation of zig-zag pattern

The quantization matrices can be scaled based on Qf. With $Qf = 100$ the matrix should just be full of ones and the values should increase in size as $Qf \to 0$. The scaling is done with the following formula

$$\mathbf{Q}_{i,j} = \max(\lfloor S\frac{(\mathbf{Q}_{i,j} + 50)}{100} \rfloor, 1)$$
$$\text{where } S = \begin{cases} 200 - 2Qf & \text{if } Qf \geq 50 \\ \frac{5000}{Qf} & \text{otherwise} \end{cases}$$

During decoding the values in the **Quant** matrix are multiplied pairwise with $\mathbf{Q}_Y$ or $\mathbf{Q}_C$.

5.5 Run-length Encoding

As a result of the quantization step each block usually has a lot of zero valued elements. So to make the arithmetic coding step more efficient long sequences of zeros between two non-zero elements are compacted into a symbol $Zeros(x)$ to represent a string of x zeros. Intuitively one would traverse the block row by row, but to increase the length of each string of zeros the block is instead traversed in a zig-zag pattern as shown below:

When the a sequence of zeros where there is no non-zero value at the end is encountered the symbol $Zeros(x)$ is not used. Instead a end-of-block symbol is used. This results in a sequence of symbols for each block that can be collected and then we can perform arithmetic encoding on this sequence and have the final encoded result.

During decoding the block is simply reconstructed from the decoded sequence following the same zig-zag pattern.

5.6 The JPEG/JFIF file format

Technically JPEG is a compression standard that is saved as a JFIF file[15]. This is not relevant for our purposes, as our JPEG implementation is just for benchmarking. As such we won't bother with file formatting but simply save our JPEG encoded images as a bitstream.

6 Loss function

6.1 Commonly used loss functions in neural compression

As with most other areas in deep learning, many different loss functions are prevalent in the field of neural lossy image compression. The mean squared error (MSE) is quite common[1] and has been used to get some quite good results[10]. In many areas of machine learning, the MSE is a default choice because it is simple and punishes large errors more

heavily. However, when working with lossy image compression, the ultimate judge is usually going to be the people looking at the images. Thus it is evident that since the loss function influences what data is favoured by the model, the loss function needs to align well with the human visual system (HVS), in order for the model to generate good reconstructions of the original image[16][17]. Unfortunately the MSE does not align that well with the HVS[16][17]. This can be seen in figure 6.1 from[17]. The original image is (a), and all other images have the same MSE of 210, even though they are clearly of different qualities when judged by humans:

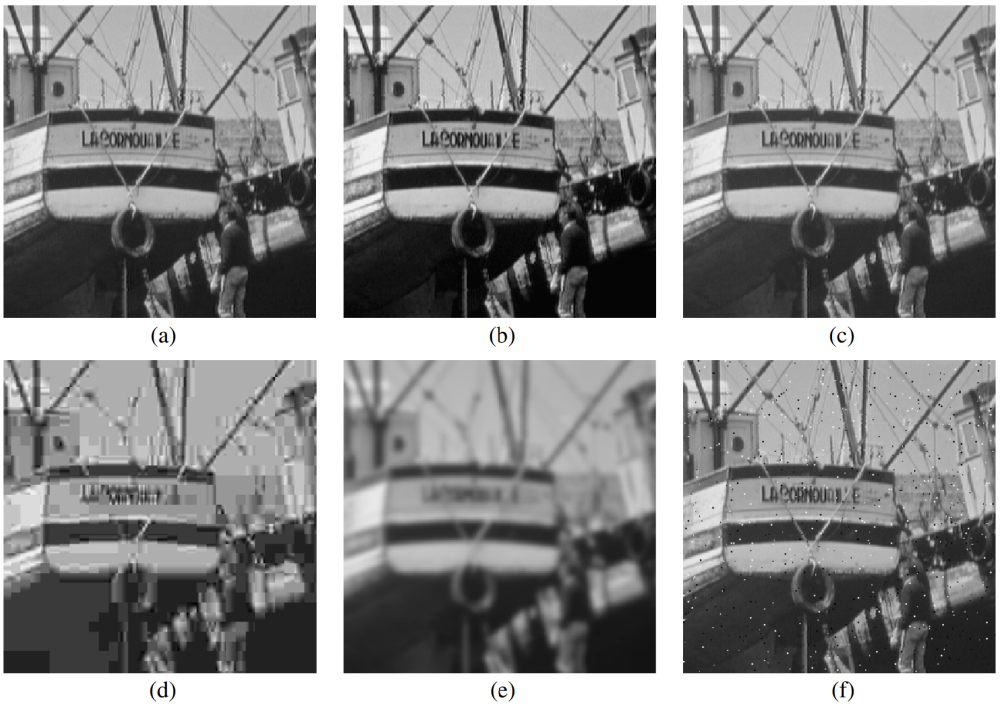


Figure 6.1: Different reconstructions all with MSE of 210 from [17]

There have been many attempts at crafting a loss function that aligns better with the HVS. Two of the most notable examples are the SSIM[17] and the MS-SSIM[18][1] that have been sited 43.529 and 5.783 times respectively according to Google Scholar. The definitions are:

$$l = \frac{2\mu_x\mu_y+c_1}{\mu_x^2+\mu_y^2+c1}$$
$$c = \frac{2\sigma_x\sigma_y+c_2}{\sigma_x^2+\sigma_y^2+c_2}$$
$$s = \frac{\sigma_{xy}+c_3}{\sigma_x\sigma_y+c_3}$$
$$SSIM = l^\alpha \cdot c^\beta \cdot s^\gamma$$
$$MS\_SSIM = l_M^{\alpha_M} \cdot \prod_{j=1}^{M} c_j^{\beta_j} s_j^{\gamma_j}$$

All these values are calculated on 11×11 blocks of the image, these are then averaged for the total loss, sometimes called MSSIM and MMS-SSIM where the leading M stands for mean[17][18] (we generally leave out the leading M in this report). The MS-SSIM is based on the SSIM and they are quite similar and suffer from many of the same problems. We will mostly discuss the SSIM, since it is the most prevalent and the base of other loss functions like MS-SSIM, CW-SSIM, 3D-SSIM. Most points discussed should be applicable for the derivatives of the SSIM loss function[19][18]. The SSIM loss function is created under the assumption that the HVS is highly adapted for extracting structural information

from a scene[17]. It then tries to exploit this by being a measure of structural similarity (hence the name Structural Similarity Index Measure). It is somewhat unclear whether or not the SSIM really measures structural similarity[19], but it does seam to align better with the HVS compared to the MSE. Looking back at figure 6.1, the SSIM loss of images (b) to (f) are 0.9168, 0.9900, 0.6949, 0.7052, 0.7748, which from our assessment aligns much better with the HVS compared to the MSE.

We did a small preproject on the subject (appendix A and B), where we used and compared the SSIM, MS-SSIM, MSE and the MAE (mean absolute error) loss functions. The SSIM and the MS-SSIM performed better visually, however, they were also less stable, often suddenly blowing up the error or getting stuck. It was somewhat possible to circumvent these problems by reducing the learning rate and pretraining the network with more stable loss functions like the MAE or MSE, but this made the training process more cumbersome and slow. We also thought that the basis for the SSIM was too simplistic, since it was originally created for black and white images and only tries to simulate how the HVS sees structural similarity, not other types of similarity (e.g. color) [17][18].

All these factors made it clear to us that we were not satisfied with the SSIM loss function. In our preproject (appendix A and B), we tried to improve upon the SSIM loss function, by splitting the image into its YCbCr components and using the SSIM only for the Y lumen component and other loss functions like the MAE on the Cb and Cr chromatic components. The idea was that since the SSIM is not designed with color in mind, and that the Y component should contain all the structural data, one could try to exploit the supposed strong sides of the SSIM while mitigating the weak aspects. This approach worked quite well, clearly outperforming the normal SSIM loss function after tweaking the resulting hyper parameters that arise from balancing the weight of the lumen and chromatic components.

However, we were still not satisfied with the results, we did not like that the SSIM is blocking the image into sections, since this is one of the criticisms of the JPEG algorithm where it leads to blocking effects[1][20][21]. We could also not find any good loss function to pair the SSIM with, since we could not find any loss function that aligns well with how the HVS perceives color. Furthermore, the issue with the loss function being unstable was still present.

[19] shows that there are many other fundamental problems with the SSIM loss function, some of these are: There appears to be a direct relationship between MSE and SSIM, and since the MSE is not a perception-based metric, SSIM might not be one either[19]. One can also derive the PSNR (Peak signal to noise ratio) from the SSIM which also suggests that SSIM is not a perception based index[19]. There are also many cases where SSIM does not align well with the HVS [19], one is shown in figure 6.2 below:

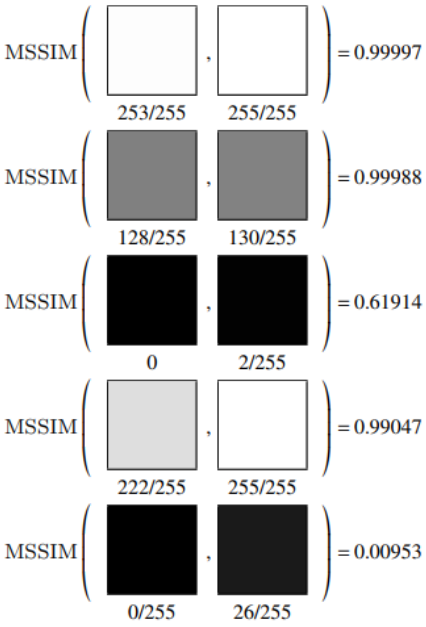


Figure 6.2: Misalignment with HVS and SSIM from[19]

Here human judgement clearly disagrees heavily with the SSIM. Further in [19] they also show, portrayed in figure 6.3, how poorly the SSIM can perform with color, and how large an influence the pixel density of the image has on the SSIM:

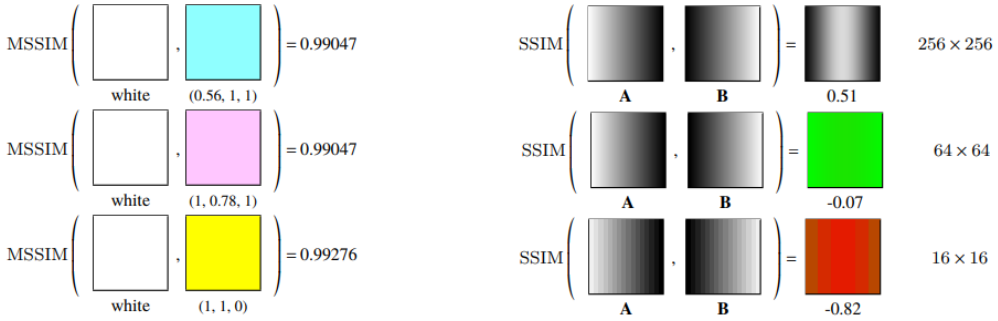


Figure 6.3: Poor performance of SSIM caused by color and granularity of images from[19]

Because of these problems, we will try to find a new loss function that aligns better with the HVS and has fewer problems than the SSIM.

6.2 Our approach

In order to find inspiration that might help us find or create a loss function superior to the above mentioned approaches, we look to related areas of research. Face recognition and face verification are two fields where hand crafted metrics and methods used to be a weak link (We will refer to face recognition and face verification interchangeably, since both are applicable to our problem in the same way)[1][22][23][24]. The best face recognition pipelines used to rely on hand-engineered features[24], but the best modern approaches have discarded hand crafted metrics, in favour of learned metrics[24][25][26][27].

When training a model to measure similarity between faces, one typically has a data set with multiple pictures of multiple people. One example being in [27] where they

had 4.000.000 facial images belonging to more than 4.000 identities. There are a few different architectures that are most common for learning facial similarity, probably the most successful being the Siamese network with a triplet loss function[27][25][26]. Figure 6.4 from [22] showing the architecture can be seen below:

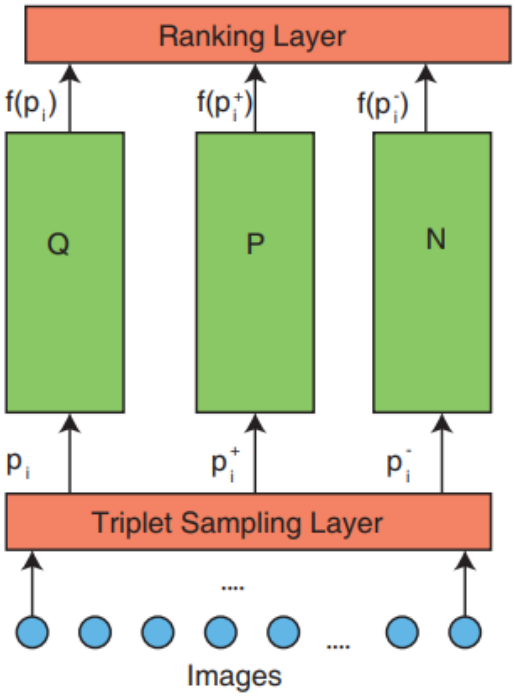


Figure 6.4: Siamese triplet loss architecture from[22]

Where Q, P and N are typically the same convolutional neural network (CNN) with identical parameters, and P_i is the original image P_i^+ is the positive image (Different image of the same person), and P_i^- is the negative image (Image of some other person)[26][22]. Where the triplet loss function defined by:

$$L = \sum_i^N [||f(P_i) - f(P_i^+)||_2^2 - ||f(P_i) - f(P_i^-)||_2^2 + \alpha]_+$$

Where f is the neural network representing Q, P and N. Due to the α, the loss function forces P_i and P_i^+ to be closer together than P_i and P_i^- as measured by the euclidean distance[26][22]. It is crucial for fast convergence, that P_i^- is quite similar to P_i and P_i^+, otherwise the task will be to easy for the network, and gradient descent will slow down to a crawl[26].

By using this approach, it was possible to discard the hand-crafted metrics and get superior performance[24][25][26][27].

One could question the need of this rather complex architecture that outputs a high dimensional point, rather than a more simple architecture that directly outputs a probability of the two faces being from the same person. One of the reasons why this architecture is so attractive for face recognition, is because the large computational load of neural networks, lead to it being beneficial to minimize the amount of images parsed trough the network. By placing all images into a D dimensional space, one can efficiently compare N faces to one another, by only inserting each face into the model once, and using normal distance measures to compare outputs of the N different faces[26]. This can reduce the

parses trough the network from O(N^2) to O(N) when trying to find the two most similar faces in a database. This architecture also allows for parallelism, since one can parse any number of images trough any number of neural networks and compare them afterwards. Furthermore, this method also allows for precomputation of all faces in a data set, such that if one wants to see if a new image has a match in the data set, one only needs to parse this one new image once trough the model, in order to compare it to all other images in the data set, reducing the parses from O(N) to O(1)[26][22][23]. Since typical face verification tasks, can contain a database of thousands of faces, efficiency and precomputation can be very important for a face verification tool.

Because of the complexity of the HVS and the lack of understanding about the HVS[16], we believe that deep learning approaches might be even more needed in measuring human aligned image similarity, compared to face recognition. We believe that the Siamese network with triplet loss method should work for image similarity as well, because the tasks are very similar. However, we do not think that the computational advantages of this approach are going to have a large benefit for our needs. We will only use the distance metric as a loss function and will most likely not need to compare one image to more than one other image.

Thus a simpler architecture where both images pass trough the network simultaneously, resulting in an sigmoid that assesses the similarity might be a better fit for our needs. It would make intuitive sense that a simpler approach would work better with a smaller data set and a smaller model. Furthermore a smaller network would also be faster, which in turn would speed up the training of the network that is supposed to use this loss function. These advantages would be great since we have limited computational resources.

Other than the computational advantage, this architecture also allows us to have more control over the output of the loss function, since we can specify what the "correct" loss between two images should be. Thus we do not need to group the images into binary categories of acceptable and not acceptable (like P_i^+ and P_i^-), instead we can specify a continuous range of acceptability depending on how similar the images are. This more continuous range of for example 0 to 1, could also smoothen training, since there are no extreme values which could result in the gradient decent performing a huge step and possibly worsening the training.

7 Data

7.1 Base image data

Now that we want to use neural networks to compress images, we need data, preferably a lot of good data. Furthermore, in order to facilitate our method to have free reign in choosing which parts of the image data to loose, we will not use images that have been compressed beforehand. This limits the types of images that we are willing to use, since some image formats like JPEG alter the image. When going trough all papers listed in a benchmark of end-to-end lossy image compression[1], we found that most papers used the Kodak data set, containing 24 512×768 PNG color images[28]. There where some other data sets used, most of which contained JPEG images. 24 images is not a huge amount of data, especially when comparing with the amount of data used in some of the cutting edge deep learning image processing papers, e.g. in Google's FaceNet, where they had 260.000.000 images in the training set[26].

Luckily due to an interest in photography, we have a private data set of 6.645 3264×4928 NEF color images (raw uncompressed images). Most of the figures in the report come from

this data set. The data set is quite diverse with images of nature, people, animals and more, but we intentionally chose to only show images of flora and insects for privacy reasons. These images are a lot larger than the the images in the Kodak data set, and would thus be more demanding of our computational resources. In order to lessen the computational load and be better able to compare ourselves with the literature, we downsize our images to fit the Kodak image format of 512×768. By using a combination of cutting and downsizing the images, we can get the added benefit of also increasing the size our data set.

To make the images 512×768, we start by trimming the edges, in order for the x axes to be divisible by 768 and the y axes to be divisible by 512. Then the image gets cut into pieces as can be seen in figure 7.1:

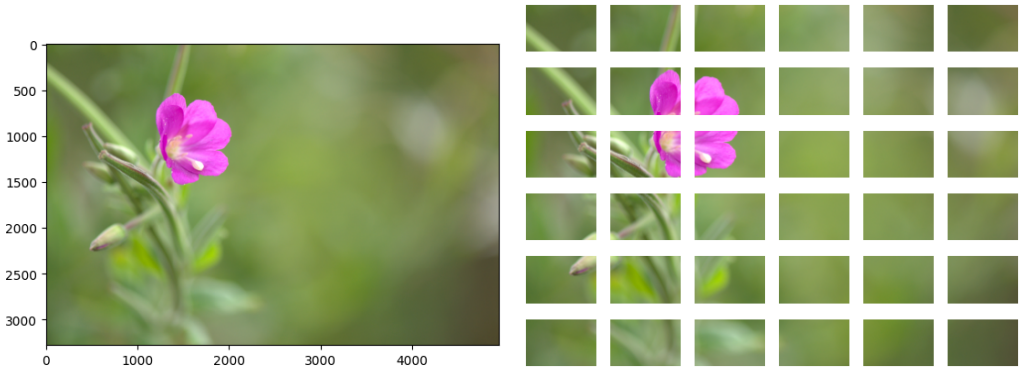


Figure 7.1: Cutting of the original image

Each of these pieces is 512×768 and gets added to the data set. Afterwards the trimmed image gets downsized and cut again, until it is no longer possible. This can be seen in figure 7.2 below:

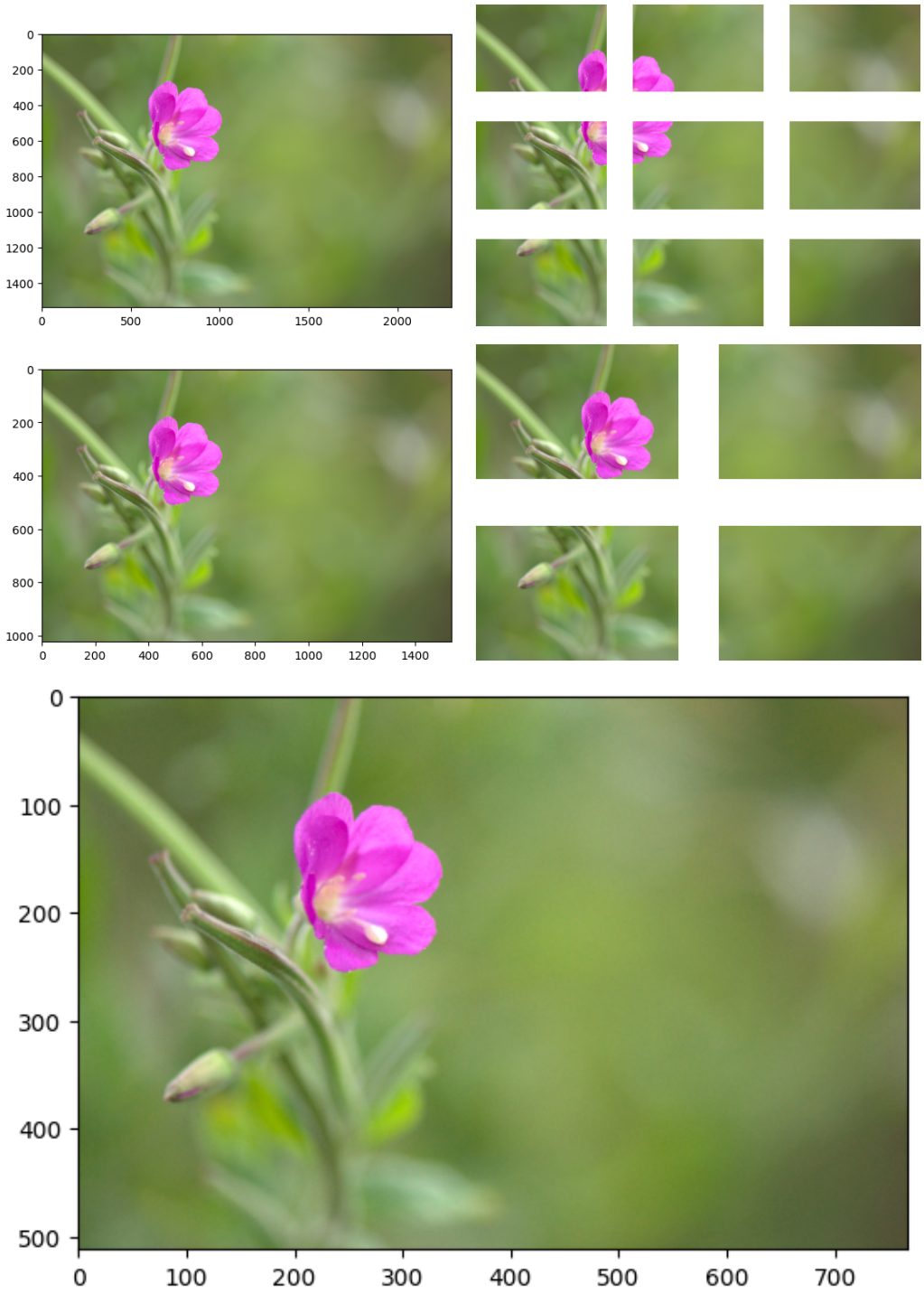


Figure 7.2: Further cutting of the original image

After this procedure, there are 50 512×768 images for each original image, leading to there being 332.250 images in total. We also apply flipping and rotation of the images, these can be seen below:

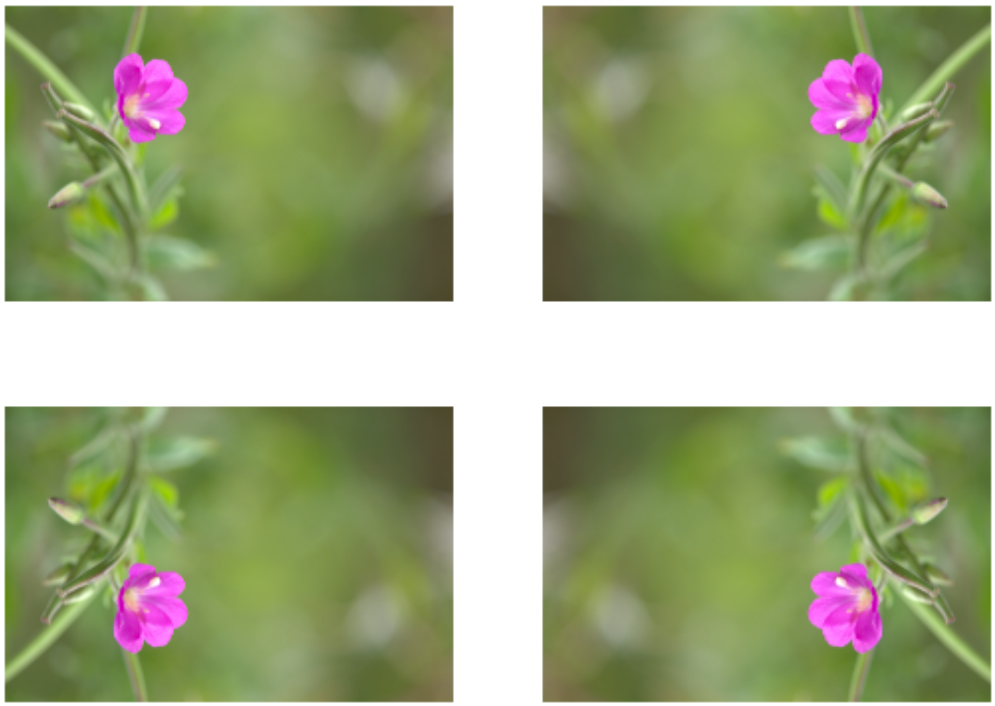


Figure 7.3: Flips of image

This further increases the set of images by a factor of 4, leading to a training set of about 1.329.000 images. Many further transformations of the images are possible, these include rotations of the color channels and shearing etc. It can be debated which of these transformations would produce a combination of pixels that can be regarded as an image (as described in section 3). Thus inclusions of some other transformations might make it harder for the euto encoder to figure out the underlying lower dimensional manifold of the set of images, and thereby possibly reduce real world performance.

1.329.000 images is less than 1% of the amount used in cutting edge papers like [26], however, our computational resources would and time constraints likely be insufficient to handle much more data.

Other than our private data set, we are also using the Kodak data set, to compare some of our results with the literature.

7.1.1 Human rated data set

In order to train our loss function, we need a data set containing human rated reconstructions of images along with the original images. We do not have access to nor know of any such data set, and must therefore make our own.

Lossy image reconstructions are in a way an image augmentation. We can make a preliminary data set of reconstructions by implementing image augmenting functions, and applying them to our images. These ”reconstructions” could then be rated by humans, resulting in a human rated data set of image ”reconstructions”. If there are many different augmentations that affect the images in different ways to varying degrees of human acceptability, the data set would to some degree encode human preferences. Of course, it seams impossible to make enough augmentations to fully capture the space of image reconstructions, but hopefully the neural net can extrapolate from the limited data set.

Even a relatively small set of augmenting functions would lead to a large increase of the size of the data set, since the total size of the data set would be $|images| \cdot |augmentations|$. Getting people to rate just a subset of all the augmentations would become costly and time consuming. Furthermore, there is no guarantee that the our augmentation data set would resemble the reconstructions that our image compressor would create. If the reconstructions would not be similar to the augmentations, then the trained loss function would be deployed in an environment with a different data distribution compared to the training environment. Having distinct training and deployment distributions, could lead our loss function to perform poorly in deployment. As described in [29], one can solve the problem of the differing distributions by training in a circular manner where one:

- Uses human ratings to train loss function
- Uses loss function to train compressor
- Gets data set of image reconstructions generated by compressor
- Gets humans to rate a subset of the images generated by compressor
- Repeat

This procedure is described in the the chapter about further work in section 12.1.2.

The cost and time required for this circular training scheme would be even greater than the cost of just rating the images once. In short human rated training sets are unfeasible for our current project.

7.1.2 Human rated augmenting functions

As mentioned, getting humans to rate the all images after augmentations have been applied, would require $|images| \cdot |augmentations|$ ratings. In [29], they found that it in their cases it was enough to only rate a subset of the data, since the model was able extrapolate from the subset. However, getting humans to rate even a subset of the images would likely still be a quite substantial task due to the size of $|images| \cdot |augmentations|$.

If we assume that for each augmenting function, the affect the augmenting function has on any image would get rated the same by human raters, or expressed formally:

$$\forall_f(\forall_i \forall_j(h(f(i), i) = h(f(j), j))) \mid f \in F; i, j \in I \tag{7.1}$$

Where F is the set of augmenting functions, I the set of images and h the human rating of the augmentation. Then one would only need humans to rate one augmented image for each augmentation function, since the effect on the other images would be the same. This would reduce the amount of ratings from $|images| \cdot |augmentations|$ to $|augmentations|$, which is much more doable for us.

We ended up implementing 14 augmenting functions, some examples of images being transformed by different functions the figures below:

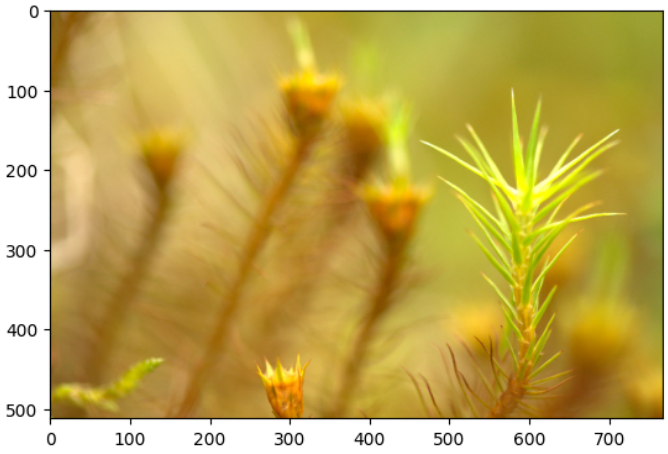


Figure 7.4: Original

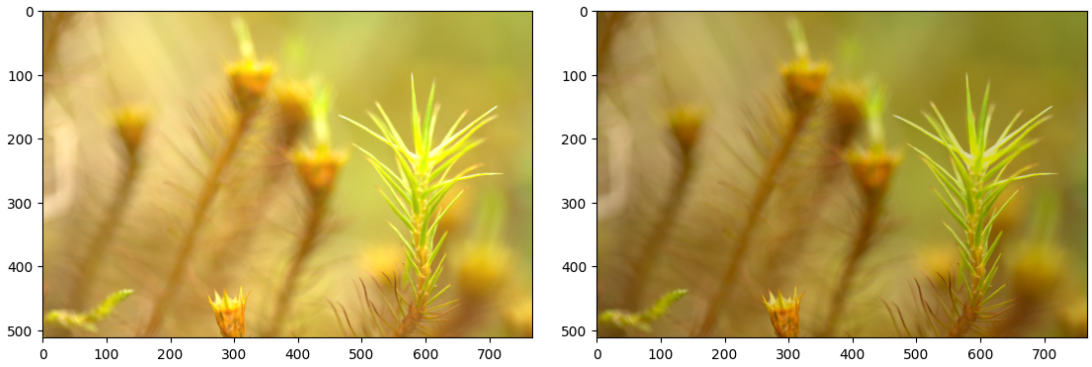


Figure 7.5: Adding by scalar, multiplying by scalar

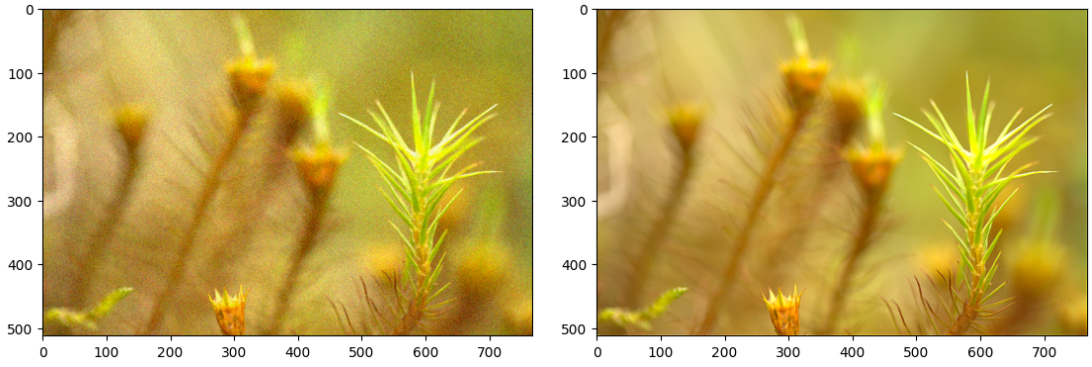


Figure 7.6: Uniform noise, unsharp masking

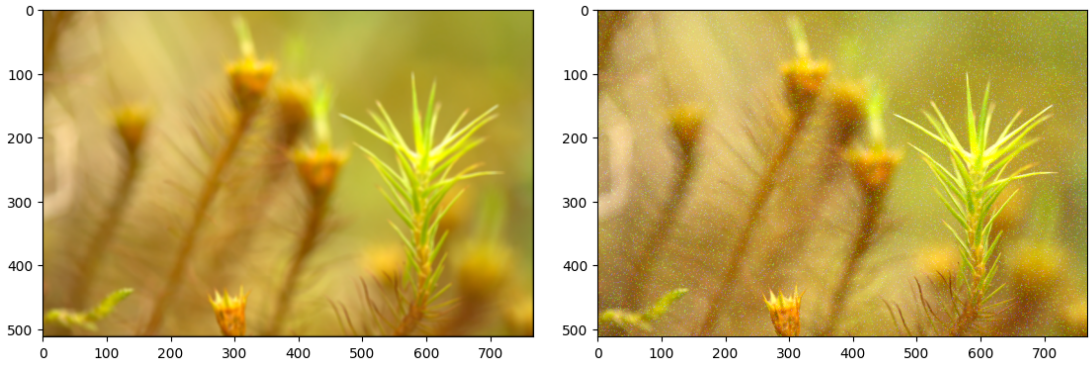


Figure 7.7: 5×5 box blur, salting

The functions we implemented came in a few different types: different Kernels, different types of random noise, adding or multiplying all pixels by values and so on. As all ready mentioned introducing these augmentations increase the size of our data set, but also increase computational load as they have be performed. Most of our code makes use of optimized code from libraries, but to apply different kernels to the image, we make use of Fast Fourier Transform to improve the running time. In order to save on memory, all the augmenting functions are implemented as generators. This means that each image augmentation is generated just in time for when it is needed, instead of generating many augmentations and looping over them. This approach is also scalable to larger images and more augmentations. Generators can be somewhat slower, but this is a necessary trade-off, since RAM quickly reaches capacity when the program must handle multiple images.

Some image comparisons of the same augmentation being applied to the different pictures can be seen below:

Figure 7.8: Peppering comparison on different images

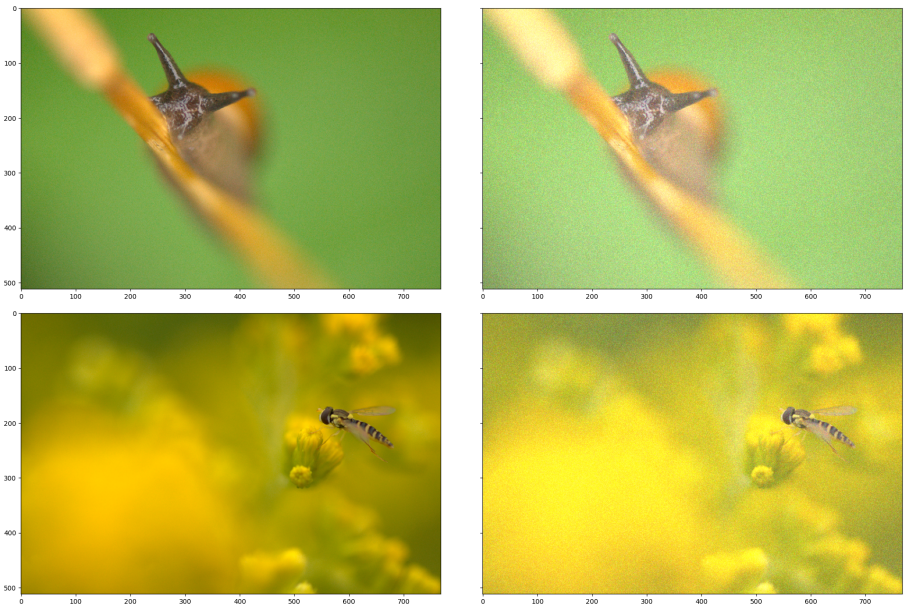


Figure 7.9: Exponential noise comparison on different images

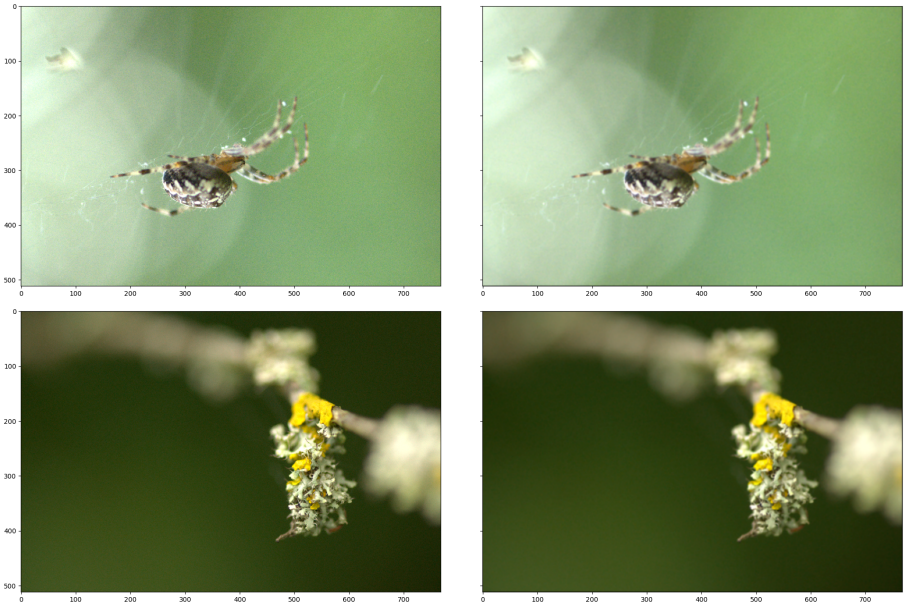


Figure 7.10: 5×5 Gaussian blur comparison on different images

The assumption from expression 7.1 has flaws but we asses it to be reasonable . The case were we found the assumption to be particularly flawed was when applying the blur filter on sharp vs blurry images:

Figure 7.11: 5×5 box blur on sharp vs blurry image

We hardly notice a difference in the before and after of the blurred image, but the sharp image clearly got degraded. But overall we think that rating the augmentations is a good compromise. Furthermore the model could also be a starting point, where one later could built upon it for example using the circular training techniques from [29], that we discussed earlier.

Thus we decided that we were going to pursue the idea. We chose 3 visually different images, and rated them on different settings of the augmentations. In total we had 119 different augmentations leading to 357 comparisons that we had to agree upon. We rated these augmentations on a scale from 0 to 1, where 0 means it is indistinguishable to us from the original, and 1 means that is far from the original. We also tweaked the tweakable augmentations in order to get a somewhat even distribution from 0 to 1. Since there are 119 augmentations and 1.329.000 images, the total loss function data set contains 158.151.000 data points.

8 Efficient deep neural networks

As mentioned, we are implementing both the loss function and the compressors using neural networks. Normally, it would be beneficial to try to optimize the architecture and hyper parameters by some sort of grid search or Bayesian optimization, iterating and tinkering until the model seams to be well adjusted. However, because we only had access to a machine with one GTX 3080, our computational resources are quite limited compared to what is used in cutting edge research. The computational limitations combined with our deadline, make us unable to train a wide array of models on a lot of data. We could limit the amount of data to a subset of our data set, and then train and compare multiple models on this data set, however, initially reducing the data set could bias our tuning towards simpler models that do not require as much data for convergence. Instead we will try to

carefully construct an architecture based on the most successful architectures and adjust for the fact that most of these architectures require a better computational setup than we have access to. Luckily there have been advances in making computationally efficient neural networks[30][31][32][9][33][34]. The hope is, that by using cutting edge ideas, we can achieve satisfactory results in only a few iterations of tweaking and tinkering.

8.1 Expressive and efficient neural networks

We want our models to be able to fit the underlying data well, and to understand the data well. Deep models are able to fit highly non linear models, and learn an approximation of the possibly highly non-linear manifold where the data recides[35][36][9]. However, deep models also tend to have more parameters, which then sets higher requirements of the setup. We will cover all the tricks we are going to employ in order to deepen our networks, while still keeping the number of parameters under control. We will also show methods to enhance performance/convergence speed without changing size of the network.

8.1.1 1×1 convolutions

1x1 convolutions are computationally relatively cheap, since they have a lot fewer weights than f.x. 3×3 convolutions. It is somewhat non intuitive to include 1×1 convolutions in CNN's, since convolutions are usually used because of some expected spacial dependencies. However as shown in [36], 1x1 convolutions are mathematically equivalent to the universal function approximator the multilayer perceptron. One can essentially get more depth per parameter of the network, thereby increasing the non-liniarity that the network can fit to.

The inception networks also use 1×1 convolutions[31][32][9][33] both for the reasons mentioned in [36], but also to compress and split data into branches[31][32][9][33].

8.1.2 Depthwise Seperable Convolutions

One can furthermore reduce the number of parameters, without affecting performance much, by using depthwise seperable convolutions[30][34]. These work by splitting a normal $N \times N$ convolutions into layerwise convolutions followed by 1x1 convolutions. The amount of parameters saved depends on the architecture, but in [30] the total number of parameters was reduced by a factor of 7.

8.1.3 Smaller convolutions

In [9], they argue that one does not need bigger convolutions e.g. 5×5, 7×7 or 11×11 convolutions, since one 5×5 convolution can be replaced by two 3×3 convolutions, which in turn reduces the number of parameters by $1 - \frac{2 \cdot 3 \cdot 3}{5 \cdot 5} = 28\%$. The reason why they say that two 3x3 can replace one 5×5, is because the information from one pixel can travel one pixel in any direction, for each 3×3 convolution, and two pixels for each 5x5 convolution[9]. This is illustrated in the picture below, borrowed from [9]:

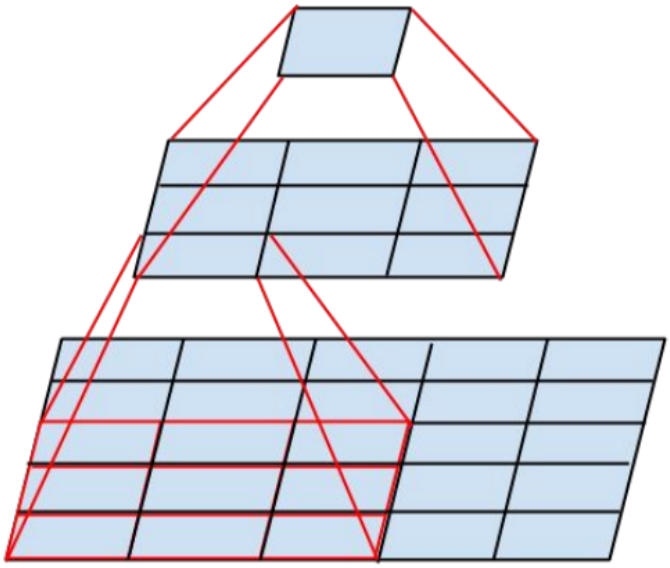


Figure 8.1: How information can travel in convolutions from [9]

One can probably even argue that the higher amount of non-linear activation functions, could increase the expressiveness of this approach. In general, the reduction is $1 - \frac{(N-3)\cdot 3\cdot 3}{N\cdot N} = 1 - 9\frac{N-3}{N\cdot N}$ assuming that N is odd.

However, in [9] they argue that one can even replace any $N \times N$ convolution by the two convolutions $1\times N$ and $N\times 1$, reducing the amount of parameters by a factor of $1-2\frac{N}{N^2}$. An image from [9] depicting how 1×3 and a 3×1 convolution can replace a 3×3 convolution is shown below:

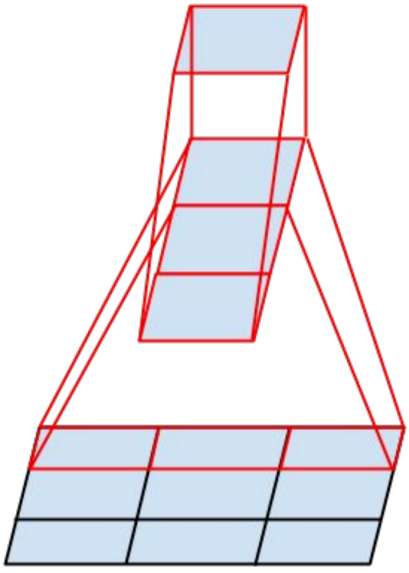


Figure 8.2: How information can travel in convolutions from [9]

One could come to the conclusion that fewer layers with larger $N\times 1$ and $1\times N$ convolutions would allow data to traverse throughout the layers with the most efficiency. However, in

[9] they argue that smaller convolutions seem to be superior in the initial layers of a convolutional network, while larger convolutions are more beneficial in later layers.

8.1.4 Activation function

Sigmoid is a classical activation function that has been used for a long time, but the sigmoid is not suited for efficient learning, since it slows down training compared to more modern activation functions[37]. One can somewhat increase the training speed by using similar activation functions with zero mean like the hyperbolic tangent[37], but these still have problems with vanishing gradients, especially when used in deep neural networks[38][39]. It has been empirically shown that the family of rectified functions (E.g. ReLU, LReLU, PReLu, SReLU...), work significantly better for training deep neural networks[38][39]. One of the newer members of the rectified family, is the ELU activation function [38], which performs significantly better than the other rectified activation functions[38].

It is possible that the biggest reason why the ELU performs better than the ReLU, is because ELU can output negative values, allowing the mean activation to be closer to zero which can reduce the bias shift[38]. When using normalization and batch normalization, the mean is set to 0, which reduces the covariate shift[32]. It is possible that the better convergence performance of ELU is because of this built-in batch normalization-like attribute. Evidence for this is also shown in [38], where they state that batch normalization does not help as much when using the ELU activation function.

This does not explain why the ELU performs better than LReLU and SReLU, since both of these can also output negative values. It is likely that the asymptotic convergence to -1 makes it easier for the ELU to encode ”Don’t care”, while LReLU is linearly decreasing for negative values. Furthermore, the derivative of the ELU is continuous, which might lead to better performance compared to both LReLU and SReLU which do not have continuous derivatives.

The computational load is not much bigger with the ELU activation function [38], so that combined with the superior performance, and limitations to batch normalization (which we will cover in the next section), leads us to primarily use the ELU activation function.

8.1.5 Bacth normalization

In batch normalization, one can minimize the internal covariate shift, by performing normalization of the mini batches[32]. Faster learning of normalized data has been known for a long time and it turns out that batch normalization also quickens learning considerably[32]. Furthermore, batch normalization also stabilizes the learning, allowing for much more aggressive learning rates and less regularization, increasing convergence speed by a factor of 14 in [32]. The main difficulty when using batch normalization is that the batches need to scrambled well [32]. Having to scramble the data well would considerably slow down our training loop, since this would not allow us to load one image and apply our cutting, downsizing and augmentations to generate more images for the batch. We would have to load one image, perform one transform on the image, and then get a new image... Loading image at a time is much slower due to the slowness of loading from the disk compared to applying transformation to images in RAM. Furthermore, we are afraid that the regularizing effect [32] might be unwanted in our case. All in all, we want to try to use batch normalization, but since the ELU has some of the benefits of batch normalization built in[38], and there are some implementational challenges of batch normalization, we will not prioritize trying out batch normalization highly.

8.1.6 Weight initialization

It used to be most common to initialize the weights of a model by using Gaussian distribution[37][40]. Later people found improved performance by semi-supervised learning, where

one trains an auto encoder where the encoder has the same structure as the original model, and using the learned weights of this auto encoder as initialization for the model [35][37]. Since the learning objective in the semi-supervised step and supervised step are so different, one can think of the semi-supervised step as a parameter initialization process, where the parameters are set to values that find some underlying order in the data manifold[35]. It was thought that this performance came only because the auto encoder was able to learn patterns from unlabeled data [35][37], but it turns out that the advantage persists even if there is an abundance of labeled data[37]. This has lead to people researching the effect of initialization on learning, even when there is plenty of data[37][39][40].

In [37], they found a quick initialization scheme (often called Glorot initialization or Xavier initialization) that vastly increases convergence speed when using the hyperbolic tangent activation function. In [39], they show that the scheme from [37] is not optimal for initializing deep models that use activation functions from the rectifier family. [39] propose another scheme (often called He initialization or Kaiming initialization) when using rectifier activation functions. The He initialization scheme was theoretically verified by [40] in the case of the ReLU activation function. In [39], they managed to get cutting edge performance with their initialization scheme, significantly improving convergence speed and the final performance.

In [38], they use the initialization proposed in [39] when training models using the ELU activation function, so it should be a good initializer for us also.

8.1.7 Residual connections

Many of the results from deep learning have become possible because of deeper neural architectures, which allow for more flexible models[41][9][33]. One disadvantage of more flexible models is the possibility of overfitting, where the model fits the training data too closely and ends up not being able to generalize well to the underlying data distribution[42]. If one assumes that there is enough training data and the training data is representative of the underlying distribution, then overfitting should not be a problem[42]. Furthermore, if overfitting is not a problem, and one has enough data to properly fit the model, then it would be logical to think that there would be no drawbacks of deeper architectures. In the case where the prediction is not contingent on highly non linear calculation, the model could always just use a subset of the layers for the computation while the remaining layers learned the identity function. Alternatively, in the case where the prediction needs highly non linear calculations, then the extra layers can facilitate them with better ability to fit the non linearity. However, dominating performance of deep models is not always the case, since deeper architectures are notoriously hard to train due to problems like vanishing or exploding gradients[41].

In [41] they found that by modifying the architecture to use residual connections, one can mitigate many of the challenges with training deep models, by making it easy for the layers of the model to learn the identity function. Furthermore, models using residual connections are in theory just as expressive[41] as the models that do not use residual connections. In the case where the layers have the same dimensions, residual connections can be expressed as[41]:

$$y = F(x, W_i) + x$$

Where y is the output, x is the input, W_i are the weights and F combines x and W_i and applies an activation function to the result. Thus when $F(x, W_i)$ is 0 in the residual connection, y is the identity.

In the case where the dimensions of x and y do not match, a linear transformation is

applied to x, such that the dimensions match[41]:

$$y = F(x, W_i) + W_s x$$

In [41], the linear transformation W_s is a convolution without activation function (the appropriate channels and strides must be selected in order to get the dimensions to add up). Residual connections make it possible to efficiently train very deep (hundreds of layers) neural networks that converge quickly[41].

The findings from [41] were later confirmed by the paper that presented inception v4 [33], which our architecture will be based on, making the results very applicable to our case. In their case the speedup of convergence was significant. Below we show a figure from [33] where a network using residual connections (red) converges about 1.5-3 times faster compared to a comparable network that does not use residual connections (green and dotted):

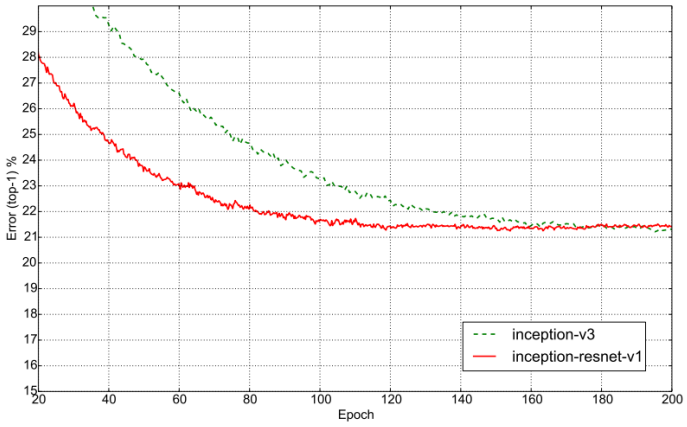


Figure 8.3: Training of residual network(red) vs non residual network(green and dotted) [33]

We will thus use residual connections in our models to increase our convergence speed.

8.1.8 Inception modules

Inception modules allow for greater computational efficiency, by being an in-between between normal convolutions and depthwise separable convolutions[34]. Images from [34] that illustrate this can be seen below:

Figure 1. A canonical Inception module (Inception V3).

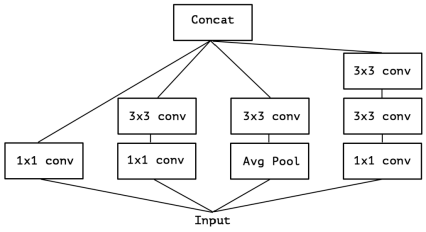


Figure 2. A simplified Inception module.

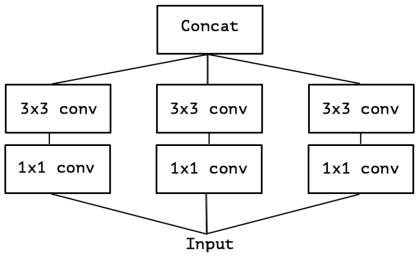


Figure 4. An "extreme" version of our Inception module, with one spatial convolution per output channel of the 1x1 convolution.

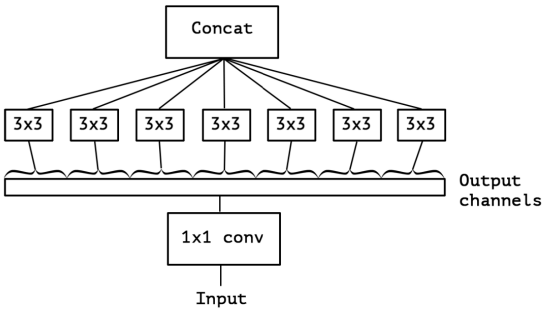


Figure 8.4: Inception and extreme Inception modules from [34]

Where the extreme inception module is more or less equivalent to the depthwise separable convolutions (the difference being the order of the 1×1 and $N \times N$ convolutions and whether activation functions are applied in between these).

Inception modules do also usually include branches with different layouts, for instance parameterless branches (e.g. pooling branches), which are computationally efficient, and larger convolutional branches like 7×7 branches[31][32][9][33]. These different branches might give some kind of balance between the different possibilities.

As mentioned in [9], the complex architecture of the inception models make it difficult to adapt the architecture to new use cases without hurting performance. Since we are making similar architectures and applying it to a new use case, we try to stick to some of the guidelines provided especially in [9]. The advice that has not been mentioned in this chapter, is to avoid representational bottlenecks especially early in the network. We do this by increasing the number of channels in the beginning, and only reducing the dimensionality slowly after quite a few layers.

8.1.9 Changing learning rates

Using learning rate decay has long been a defacto standard when training neural networks[43][44][45]. It has been shown that learning rate decay not only speeds up training by avoiding local minima and oscillation, but that it also allows for learning more complex patterns[44]. Finding optimal learning rates can be a hassle and can depend highly on the architecture of the model[32][43]. By using cyclical learning rates, one does not need to tinker with the learning rate as much and it does also often lead to better convergence and better performance[43]. We are going to use cyclical learning rates, in combination with learning rate decay. This does not increase the computational cost meaningfully and can speed up training significantly[43].

8.1.10 Optimizer

Gradient decent has been the backbone in neural networks, and methods like SGD has lead to great performance on difficult problems[46]. However, SGD is prone to getting stuck in local optimum and saddle points, which has led to research in adaptive methods like

AdaGrad, RMSProp, Adam and many more[46][45][47]. Adam is a very popular optimizer, it is built on ideas from AdaGrad and RMSProp and is generally one of the best and most popular adaptive optimizers. Adam does often initially perform much better than SGD[48], however, because of loss flattening/ weight adaptivity, Adam plateaus and generalizes worse compared to SGD[48]. Furthermore Adam has not shown great performance on the related field of image recognition[48]. This has lead to some people using Adam early in the training, but switching over to SGD later to get better generalization[48]. This can still result in worse final generalisation if the Adam optimizer is used for to long[48].

Another approach is to improve SGD by adding momentum or Nesterov momentum to the SGD[47][49]. The momentum leads to faster learning and by using a decay on the momentum term, one can still get great generalization and convergence[47][49]. This setup of SGD with momentum and decay, has been used very successfully in many cases, notably in the inception and xception networks along with many others[9][34]. There has been disputes about whether Nesterov momentum is superior to normal momentum or not. According to [47], Nestrov seams to be superior, while [49] states that there are advantages and disadvantages to both methods. Furthermore, in [9][34] they did not use Nestrov, we do not know why, but it is possible that they tried both options and Nestrov performed worse. Since our architecture is similar to [9][34], we are going to start by mimicking their setup and use the standard SGD with momentum and decay.

8.1.11 Normalized data

As mentioned under the section about batch normalization, it has long been known that normalized data leads to faster training[32]. Due the computational load required to calculate the mean and standard deviation of all pixels in our entire data set, we instead use the procedure of subtracting 128 from the pixels and then dividing by 128. Because image pixel values are between 0 and 255, this has a similar effect to normalization, centralizing the possible values around 0 making sure that the values fall in a range close to 0 (in this case between $[-1, \sim 1]$).

9 Implementation of neural networks

9.1 Training loop

9.1.1 Storing training results

In order to better understand how the training progresses and to be able to stop and restart training, we made a function that stores all the necessary parameters in the training loop. This function is called once in every learning rate cycle as described in the next section.

9.1.2 Changing learning rates

Since PyTorch does not have a built in solution for combining learning rate decay and cyclical learning rates, we implement it ourselves. This is done in a similar way as in the triangular cyclical learning rate from[43], except that the maximal learning rate, the minimal learning rate and the steps in each cycle get adjusted by a decay factor once in each cycle. The pseudo code can be seen below:

...
Other logic
...
$Initialize : max\_lr,\ min\_lr,\ steps,\ decay \leftarrow float$
$step\_size \leftarrow (max\_lr - min\_lr)/steps$
$model\_lr \leftarrow max\_lr$
$falling \leftarrow True$

for Training loop including other logic **do**
...
Other logic
...
    if falling **then**
        $model\_lr \leftarrow model\_lr - step\_size$
        if $model\_lr \leq min\_lr$ **then**
            $save\_model(parameters)$
            $falling \leftarrow False$
            $max\_lr \leftarrow max\_lr \cdot decay$
            $min\_lr \leftarrow min\_lr \cdot decay$
            $steps \leftarrow steps/decay$
            $step\_size \leftarrow (max\_lr - min\_lr)/steps$
        end if
    else
        $model\_lr \leftarrow model\_lr + step\_size$
        if $model\_lr \geq max\_lr$ **then**
            $falling \leftarrow True$
        end if
    end if
...
Other logic
...
end for

It is possible to use the same structure to change the momentum in a cyclical pattern. It is also possible to not make the number of steps in the cycle increase. However, since we save the models after every cycle and that the progress of convergence usually tapers off when training, we find that increasing the steps leads to us not having to store as many models with similar performance.

9.2 Loss function

9.2.1 Initial implementation

We made our neural network loss function implementation flexible, such that one can get four different versions of this neural network, by giving different parameters to the neural network constructor. These four versions are: the baseline that does not try to optimize the number of parameters, slim that uses the techniques discussed in [9], separable that uses the techniques discussed in [30], and one that combines these two approaches. These optimizations where mentioned in the sections 8.1.2 and 8.1.3, but to recap: the baseline has normal convolutions. Slim kernels substitutes $N \times N$ convolution with one $1 \times N$ and one $N \times 1$ convolutions. The layerwise seperable replaces an $N \times N$ convolution by an 1×1 and a layerwise seperable NxN convolution. Lastly the combined approach substitutes $N \times N$ convolutions with one 1×1 convolution followed by an $1 \times N$ layerwise seperable and one $N \times 1$ layerwise seperable convolution. This is further illustrated in the following figure:

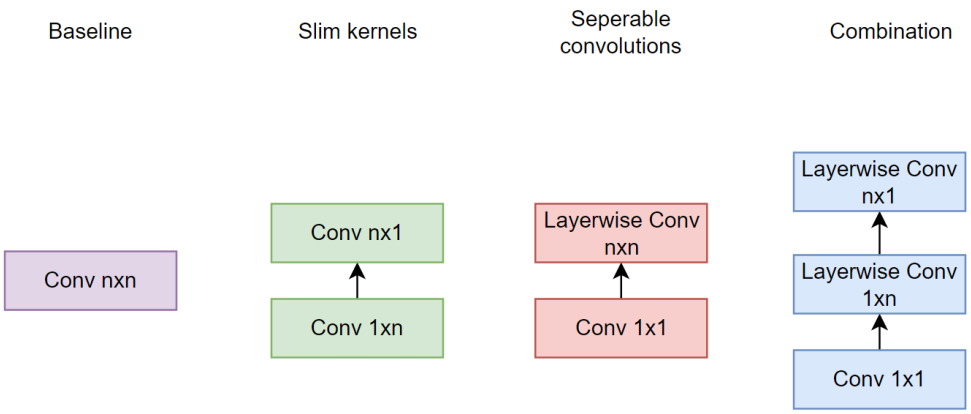


Figure 9.1: Optimization differences

One exception is the 1×1 convolutions, that are unchanged.

The network has the same overall structure regardless of the optimization used. The initial structure can be seen below:

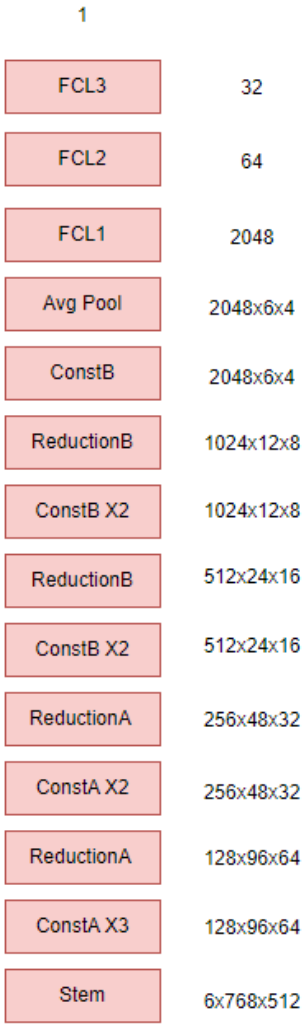


Figure 9.2: Loss function structure

Where the numbers next to the blocks show the input dimensions of the data.

As mentioned in subsection 8.1.8, the modules are inspired by the inception networks, especially inception v3 and v4[33][9]. We did tinker with the different modules, by running a few iterations of the training loop at looking at the progress. A closeup of the final layout of the different modules are shown below:

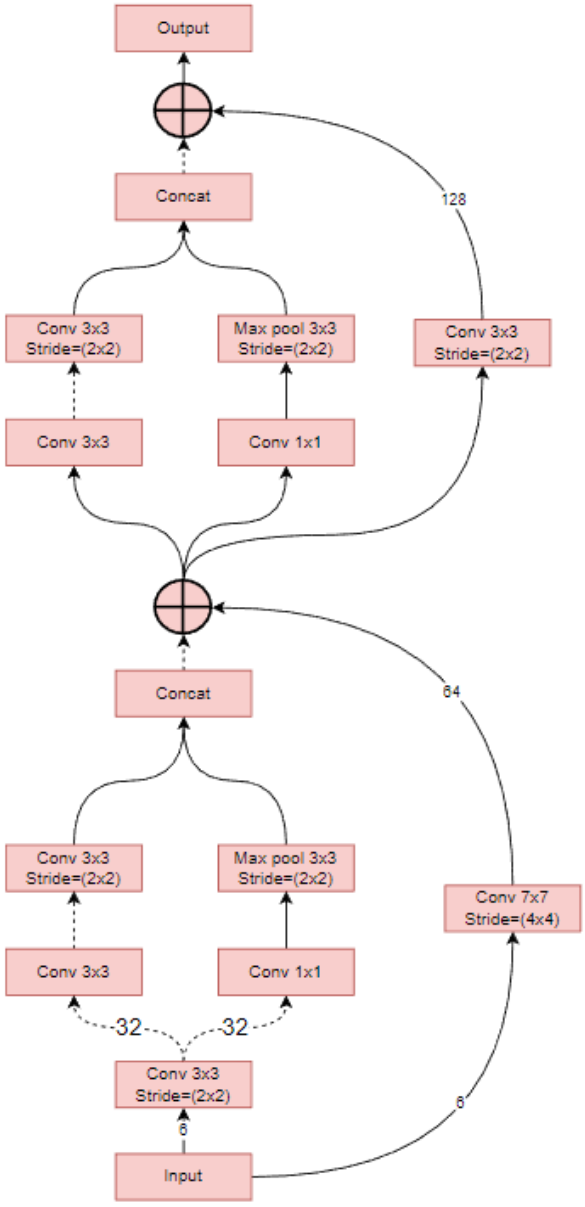


Figure 9.3: Stem

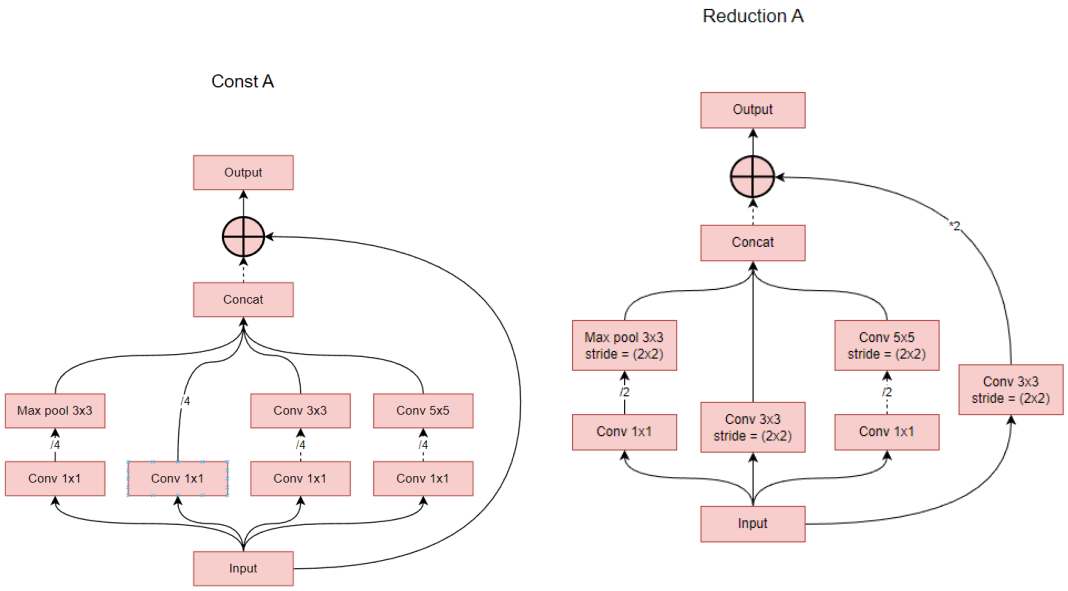


Figure 9.4: ConstA and ReductionA

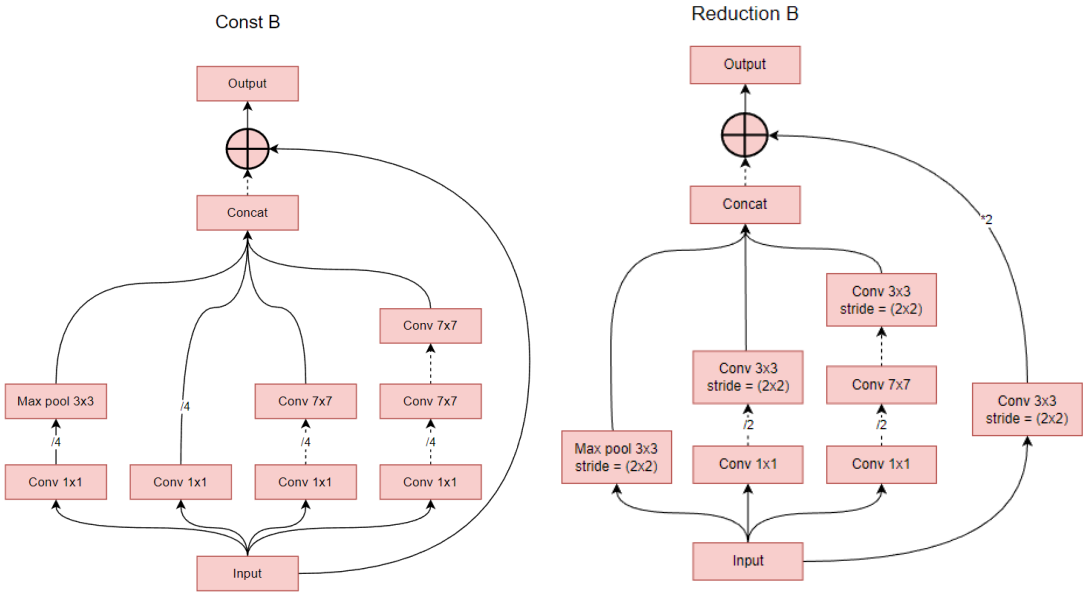


Figure 9.5: ConstB and ReductionB

These modules are for the baseline model, the others would have the differences as shown in figure 9.1. The dotted lines mean that an activation function is applied. The numbers on the transitions specify the number of channels or the reduction/increase of channels, if there is no number, then the channels are preserved from the input. Strides and similar are specified on the boxes themselves. We pad all kernels with the equivalent to ”Same padding”.

The number of model parameters of the different optimizations are shown below:

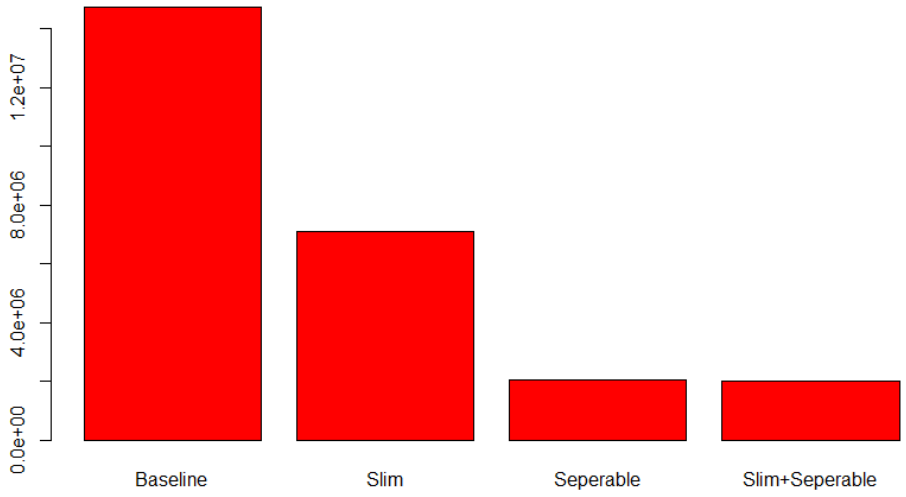


Figure 9.6: Parameters of models

The number of variables was successfully reduced by more than a factor of 7.28. However, the extra layers that come from this optimization, actually causes there to be more parameters saved for the forward and back propagation steps. As can be seen in the figure below, this seems to outweigh the benefits of these methods:

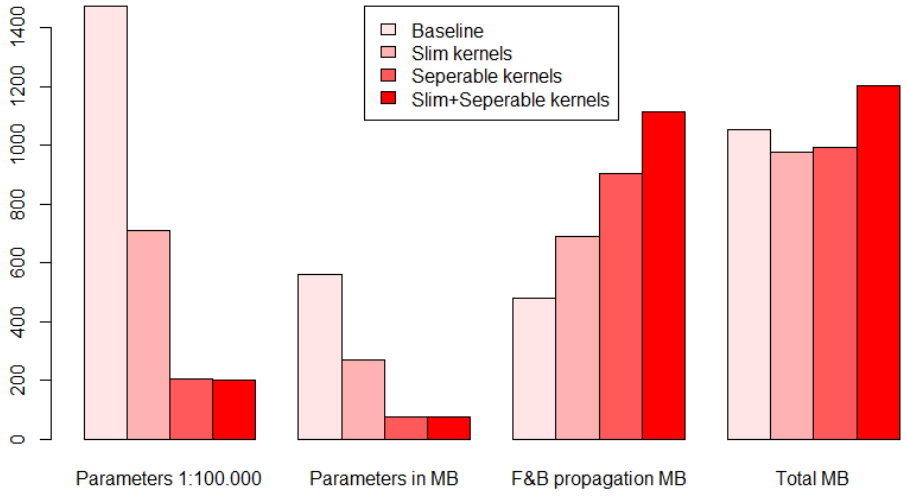


Figure 9.7: Parameters and space used by models

At first glance it seams like all this optimization as presented in [34][30][9], is the equivalent to cleaning the room by sweeping everything under the carpet. However, since e.g. [30] work under the assumption that there are a lot of computational resources under

training (when the forward and back propagation parameters need to be stored), but not on deployment since they focus on mobile device deployment, it makes a lot of sense for them to use these techniques. It just turns out to not be as useful for us, since we wanted to use it to mitigate our computational constraints during training.

We initially thought that it still made sense for us to use these optimizations for the loss function, because the loss function would not need forward and back propagation for gradient decent when in deployment. This optimization might allow us to increase the batch size or model size used for the image compressor, but it would also change the optimization problem from a first order optimization problem to a zero order optimization problem, since the model under training would not have access to the gradients. It is possible to optimize zero order optimization problems[50], one method is by gradient approximation[50], but these methods are seemingly not implemented in PyTorch and we did not want to focus on implementing it ourselves. Thus we chose to keep the gradients and have a normal first order optimization problem. When looking at the total space requirements, the optimization with slim kernels is the most space efficient, leading us to choose it.

Using the gradients of a neural network to train an other neural network might seem weird, but one can look at it as the equivalent of joining the two networks, locking the weights of the loss function part of the network, and then training the network. This is illustrated bellow:

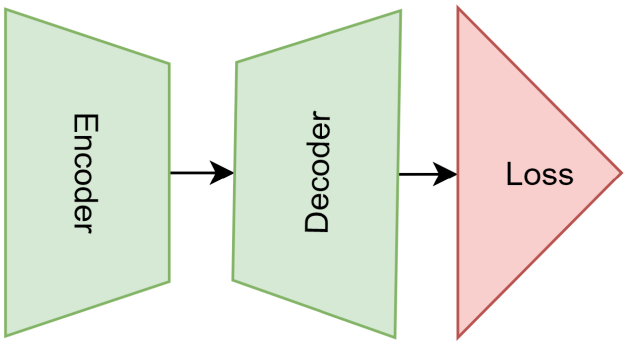


Figure 9.8: Training with neural loss function

9.2.2 Optimization

The training of the loss function was way to slow, taking days to just complete 1% of one epoch on our computer with a NVIDIA GTX 3080, making it necessary to optimize the code. In order to optimize the training, we timed the code, in order to figure out what parts of the code need to be optimized. This was done with the python time library, where we took time in between actions in the training loop. The benchmark covers all six sections of the training loop as mentioned below:

- Data loading
- Forward pass
- Calculating loss
- Backwards pass
- Accumulate running loss

- Logic for cyclical learning rates and similar

To make the benchmark reliable we limit the unnecessary processes running on the computer, and made sure that code did not print anything, since this also affects performance. We also make sure to start by running a few warm up runs before benchmarking, as this lessens the impact of startup cost on the result.

Thus the procedure was to:

- Benchmark to figure out optimization candidates
- Figure out possible optimizations
- Implement optimization that would likely have the biggest impact
- Repeat

The code we had implemented to generate images and similar could be optimized directly by using better algorithms or similar. Both PyTorch and NVIDIA do also have performance optimization guides, which we used to get optimization ideas[51][52].

In the end, there were 19 improvements that in total resulted in a 35.97 times speed up of the code. Since there were so many optimization and a lot of them were very involved, thorough discussion of them all would be cumbersome. We do not want performance optimization to be the bulk of the project and skim trough most of them. An overview of all of them and their impact can be seen in the table bellow, where cells marked with green, are those that we tried to optimize. Because of the concurrent nature of PyTorch, the distribution of time might be inaccurate, but the total time and hence the time savings is reliable. The columns of the table are: the optimization used, the reduction in running time compared to the above level of optimization, the fraction of time used for loading the data, forward pass, calculating the loss, backward pass, calculating the running loss, logic for saving models, cyclical learning rates and similar. The names of the optimizations have been abbreviated to make the table fit the page, but the ordering is the same as in the explanation here.

None	Reduction	Load	F pass	Loss	B pass	R loss	Logic
Baseline	0%	51.53%	15.09%	0.29%	25.12%	7.89%	0.04%
Cupy	35.38%	26.43%	23.5%	0.46%	39.75%	9.75%	0.1%
Only torch	32.64%	1.29%	30.69%	0.35%	52.17%	14.04%	1.46%
Channels div8	1.03%	1.33%	31.17%	0.48%	53.01%	13.87%	1.44%
No runningloss	1.08%	14.93%	31.11%	0.43%	53.37%	0%	0.15%
BL model	16.53%	3.28%	37.93%	0.41%	58.29%	0%	0.1%
Benchmark	14.73%	3.72%	39.99%	0.53%	55.64%	0.01%	0.12%
Batches	68.71%	71.25%	11.16%	0.1%	17.11%	0%	0.37%
Workers	31.08%	71.49%	11.29%	0.03%	16.83%	0%	0.36%
Less transforms	14.4%	72.03%	10%	0.13%	14.84%	0%	2.98%
No bias	1.72%	74.66%	9.14%	0.14%	14.37%	0%	1.69%
Grad=None	0.92%	75.25%	9.35%	0.12%	13.187%	0%	1.41%
Disable checks	0.59%	75.41%	9.18%	0.2%	13.71%	0%	1.5%
AMP	34.83%	56.56%	16.42%	0.35%	24.43%	0%	2.24%
Bigger batch	12.09%	63.59%	13.28%	0.33%	20.99%	0%	1.82%
Channel last	4.59%	61.28%	13.65%	0.19%	22.97%	0%	1.93%
Less gradient	3.1%	60.22%	14.23%	0.23%	23.35%	0%	1.94%
Faster kernel	0.1%	59.73%	13.19%	0.23%	24.88%	0%	1.97%
Faster ramdom	3.18%	59.51%	13.02%	0.23%	25.25%	0%	1.99%

Cupy
Even though we tried to optimize the augmentations by using Fast Fourier Transform and optimized library code, it is clearly to slow. We tried researching how we could speed up performance and because many of the transforms are essentially matrix operations, we looked into GPU computing. We found a library called Cupy [53], that gives a lot of Numpy functionality, but runs on NVIDIA GPUs. This made it quite easy to implement. There is some overhead of Cupy because of the conversions between Numpy, Cupy and torch tensors. This overhead lead to some of the simpler functions not being faster with Cupy (mainly adding or multiplying all pixels by a scalar), while other functions got a 10x speed gain (salting, peppering and poisson noise). Together, the functions had a speedup of almost 5x when measured by them selves outside the training loop.

Only using PyTorch
We found that PyTorch has many of the same functionalities built in, which makes it possible for us to limit some of the overhead of switching between Numpy, Cupy and torch tensors. This optimization was somewhat harder to implement because PyTorch does not mimic Numpy to the same degree as Cupy. However the reduced overhead combined with the superior performance of PyTorch made it well worth it, not only increasing the speed by a lot, but also reducing the load of the CPU from about 52% to 7%. Taking the load off the CPU enables us to use more concurrency and makes the CPU less of a bottleneck.

Channels divisible by 8
According to [51], it is important for performance that the channels are divisible by 8 to run more efficiently on the tensor cores. This did not give a large speedup, but it might help AMP (we write about AMP further down) to work better and since it does not hurt performance, we keep it.

Stop keeping track of loss
Keeping track of the loss forces the GPU to communicate with the CPU which lowers performance. We kept track of the loss in order to drop the learning rate when a certain performance had been reached. However, since we are using a cyclical learning rate with decay, it is not really necessary to drop the learning rate manually based on performance,

since the learning rate will drop on its own with time.

Using the baseline model
Using the baseline model instead of the model that has been optimized to reduce the number of parameter gave a significant speedup. We were surprised that the speedup came from loading the data instead of the forward and backwards pass. We reran the timing multiple times but got very similar results.

torch.backend.cudnn.benchmark=True
By setting torch.backend.cudnn.benchmark=True, PyTorch tests multiple convolutional algorithms on our model and picks the fastest one. This makes the first runs slower, but gave a significant speedup on subsequent runs.

Change data generation to enable batches
The way we had generated the training data, was by using the yield statement in python. This method does not play well together with PyTorch, since generator objects do not provide the length or similar information. By encapsulating our data generation in the MakeIter class and inheriting from torch.utils.data.IterableDataset, transforming the generator to an iterator, and writing functions that split the data among workers and specify the length of the iterator, we were able to use more of the PyTorch functionality. One such functionality was to choose a batch size. We found an optimal batch size to be about 16. Changing the batch size greatly improved performance.

If we assume that the batch is i.i.d, then the variance of the data will reduce by the square root of the batch size. The simplified version of the SGD algorithm is:

for t = 1 to ... **do**
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$
$\theta_t \leftarrow \theta_{t-1} - \gamma g_t$
end for
return θ_t

where θ_t is the parameters at time t, γ is the learning rate, g_t is gradient at time t, f_t is the model at time t.

Hence one can multiply the learning rate by the square root of the batch size and thereby increase the convergence speed while keeping the variance of the step size constant.

Change data generation to enable parallel workers
The previous change also made it possible to use multiple workers for data generation, making the communication between the CPU and GPU less of a bottleneck. We found that 2 workers was optimal while 4 workers was faster than 1 worker but slower than 2. This change also improved performance significantly.

Removing 5 most costly transforms
Some of our transforms applied the same kernel to an image multiple times (up to 20 times). By just removing the 5 slowest transforms, the performance improved significantly.

Bias=False in conv layers
When using batch normalization, one can set bias=False in convolutions layers in the model that follow the batch normalization[51]. This in turn gives an increase in performance. Since they in [38] claim that the ELU activation function has a lot of the same characteristics as batch normalization, we thought that it might not be necessary to use bias in convolutional layers that use the ELU activation function. We did not notice a significant convergence difference (longer training might result in a larger difference), but we only got a slight performance gain of 1.72%, so it might not be worth the possible convergence difference when training for longer.

Set_to_none=True on zero grad
By giving set_to_none=True as parameter when calling optimizer.zero_grad, one gets a slight speedup because it does not need to set each individual parameter to 0 and because the next backwards pass uses assignment instead of addition. This only gives a slight speed gain.

Disable checks
By disabling checks (e.g. nan checks and similar), one gets a slight speed gain at the expense of less clear error messages.

AMP
NVIDIA GPUs are cabable of running significantly faster on half precision (16 bit) than in single precision (32 bit)[54]. However, half precision can ruin training because it is more prone to overflow or underflow. AMP or Automatic Mixed Precision, automatically scales the precision of the matrix operations[54], thereby resulting in a significant memory and speed gain compared to single precision training. One problem with AMP, is that the loss might not accumulate well because of underflow, but this does not usually affect convergence and can be mitigated by scaling the loss[54]. Models trained with AMP generally have the same performance, they even tested the Inception models, that we base our model on, making the conclusion from [54] very applicable to our model.

Larger batch size
As mentioned above, training with AMP uses less memory. This allowed us to increase the batch size to 22 instead of 16, which also gave a significant speedup.

Channels last memory format
Changing the memory format to channels last, plays well together with AMP and allows for further tensor core utilization[55]. Channel last memory format gave a speedup of 4.59%.

Require gradient false except when training
When generating training data, we use PyTorch tensors and tensor operations. Tensors in PyTorch usually store gradient information, in order to be able to be used in gradient decent. By not storing the gradients except when needed in the training loop, we got a speedup of 3.1%

Faster kernel
As mentioned, some of our augmentations use kernel operations. We found that by instead of using layer wise FFT (fast fourier transform), we could use the highly optimized channel wise convolutions from PyTorch. When measured in isolation, this gave a 10x speed increase compared to FFT. However, the change was hardly noticeable in the greater context of model training, only giving a speedup of 0.1%.

Optimize by only generate random tensors and concat tensor once per image
The transforms that generate random noise, all make use of either torch.rand or torch.randn, which generate uniform and normal noise respectively. The other types of noise like exponential, poisson, salting and peppering are then generated based on the uniform and normal noise. Furthermore the images get concatenated by a zero matrix. By just generating the zero matrix, uniform matrix and normal matrix for each image, and reusing it for all transforms, one gets a speedup of 3.18%. This should not lead to worse performance, since we do generate new noise for each new image. If anything we think that it might increase performance slightly, since it makes for example the exponential and the normal noise more comparable, and it thus could easier learn the difference between the distributions.

Still not good enough
We tried a few other optimizations that did not work, and we were out of ideas. Even with a 35.97 times speedup, our code would still take almost a week for just one epoch.

Since it is quite common to get better performance after even hundreds of epochs[56], as long as the network is not overfitting to the data, it might still take months if not years for our model to fully converge.

It is possible that if we did a grid search of all the optimizations, that we could gain some speedup if two or more optimizations affect each other negatively. However this would be time consuming and we do not find it likely that it would lead to a large speed up.

DTU has a high performance computating cluster (the HPC) that we can get access to and use instead of our own PC. Using the HPC should give us a speedup, since we can get access to a NVIDIA A100 GPU. The A100 does not only run much faster, it also has 80GB of RAM allowing us to increase the batch size by a lot and get a large speedup. However, we initially gave up on using the HPC, since students can only store 30GB of files and we have 100GB of images and need to store additional information when training. But we managed to get special privileges, and thereby be able to store 400GB on the HPC. We had to rewrite some of our code, to be able to run the code trough batch scripts instead of Jupiter notebooks and it was a steep learning curve to figure out how to interact with the HPC via batch scripts and similar. In the end, we managed to get the code to run quite a bit faster, we did not time it properly as this is difficult with batch jobs, but estimate that it gave us a speedup of about 5x.

Due to our time constraints, this is still not really enough to enable us to tinker with and optimize the model to its fullest, but we can at least run the model for a few epochs, and hopefully get alright results.

9.2.3 Tinkering and hyper parameter optimization

Other than the changes made in order to speed up training, we also wanted to tinker a bit with the model hyper parameters. We did not have time to tinker much with the model, but we made a few short runs and tried to get some insight in order to optimize the model a bit. One of the things we found, was that when we had the ELU activation function before the last layer, the last layer struggled to get the non extreme values right. As can be seen in the figure below, the ELU allows for large values, whereas the gradient of the sigmoid at large values for x will be very small:

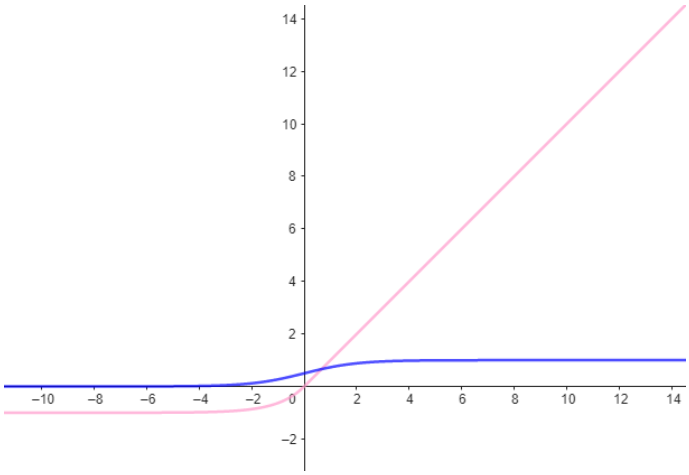


Figure 9.9: ELU(pink) and Sigmoid(blue) plotted together

We think that when large numbers get passed (indirectly) from the ELU to the Sigmoid, the small gradient leads to very slow training. We tried exchanging the ELU in the layer

preceding the sigmoid with tanh, however, this had the opposite effect where it was hard for the model to learn the extreme values. This is likely due to the combination of the small gradient still being present just one layer earlier in the model and the weights needing to be large in order to get the output from the tanh to get translated to an extreme value in the sigmoid. By splitting the second last layer into two, and have the ELU applied to one half and the tanh to the other, we experienced better training results. This can be seen in the figure below:

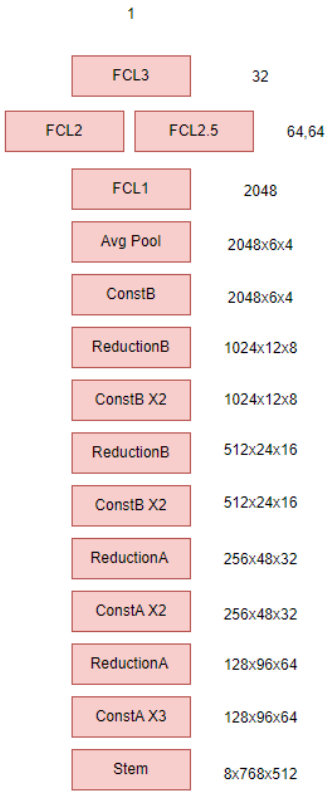


Figure 9.10: Model structure

9.3 Compression network

We implemented the compression network such that it can use the different optimizations as described in figure 9.1. The space savings from the optimizations were similar to 9.7, where the slim optimization saved a non significant amount of space. However, we ended up only using the baseline, since we found that training got hindered by the increased depth of the architecture, which seemingly resulted in vanishing gradients. We think the reason why the depth was a hindrance for this architecture but not for the loss function architecture, is because this architecture was already deeper, and it has a bottle neck layer which can increase training difficulty. We also reuse the optimizations from the loss function described in subsection 9.2.2. The overall structure is shown on the figure below:

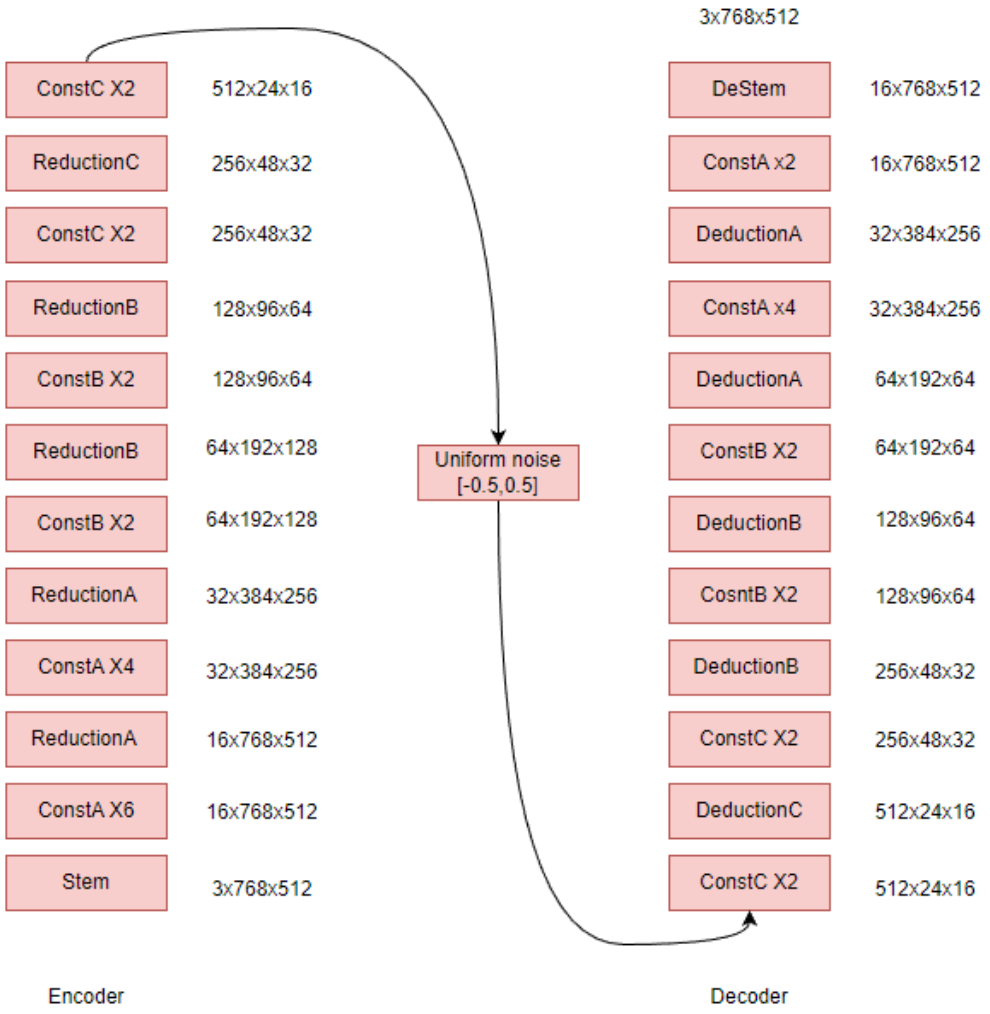


Figure 9.11: Structure of the compression model

The module labeled Uniform noise is only there in the training step, and gets replaced by entropy coding (described further in section 4) when the model is deployed. All the modules here have a differing implementation compared to in the loss function. A closeup of all the compression modules (after tinkering) can be seen below:

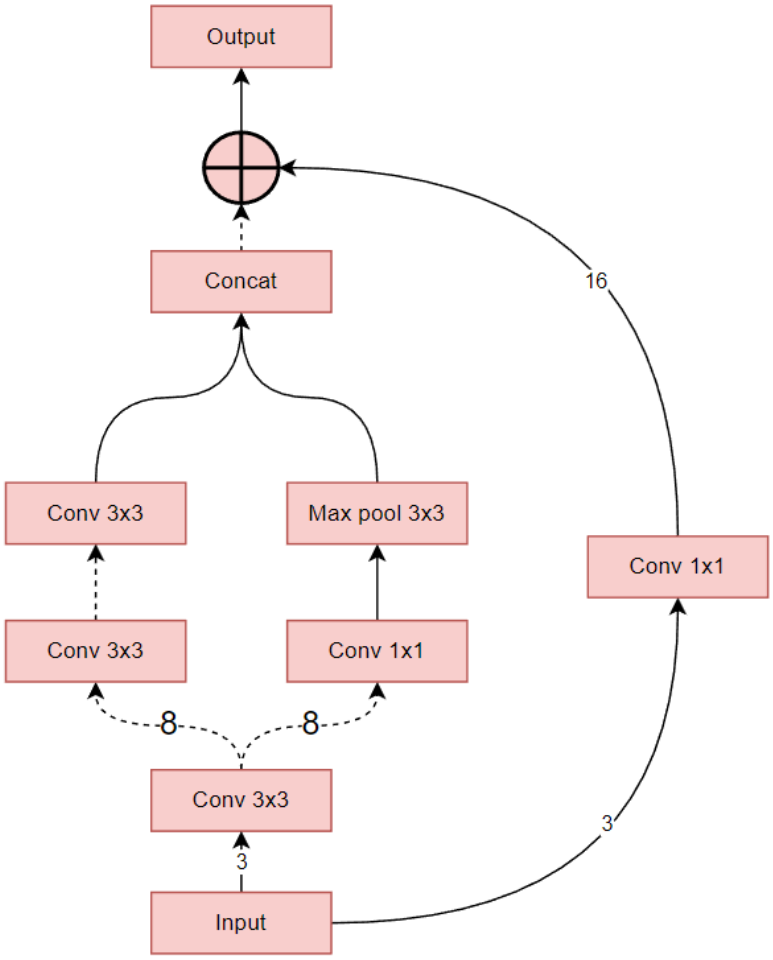


Figure 9.12: Stem of compression model

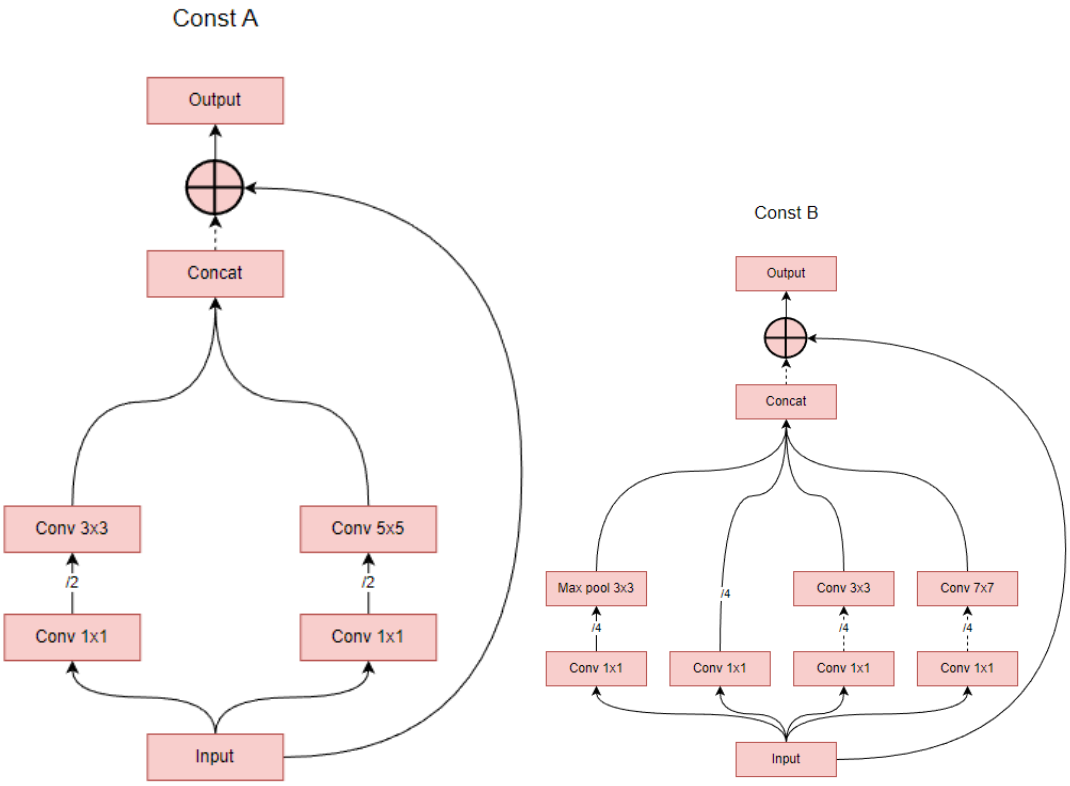


Figure 9.13: ConstA and ConstB modules of compression model

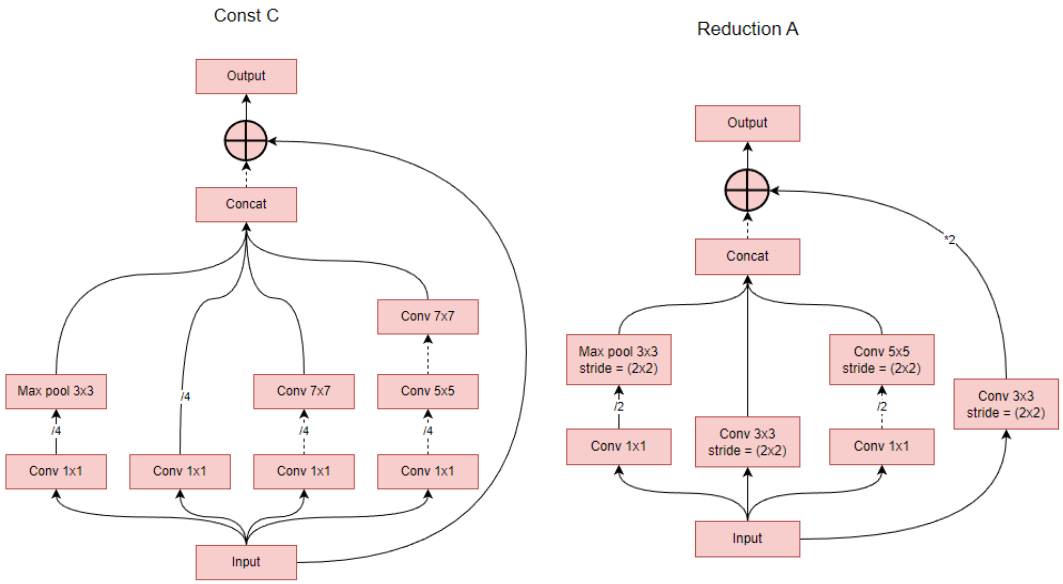


Figure 9.14: ConstC and ReductionA modules of compression model

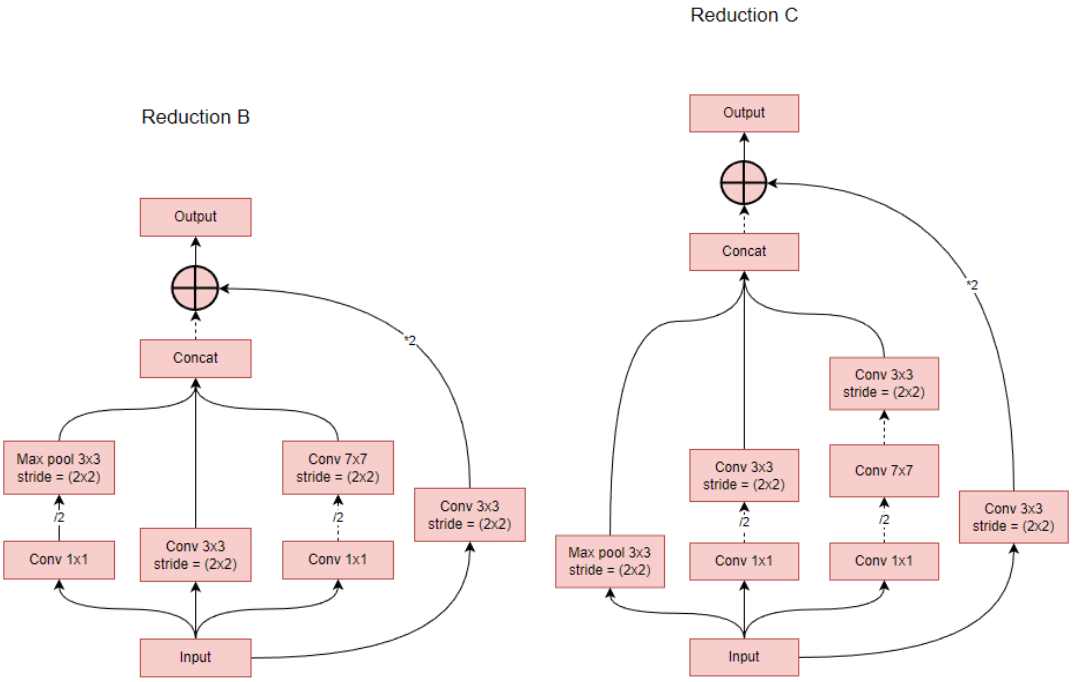


Figure 9.15: ReductionB and ReductionC modules of compression model

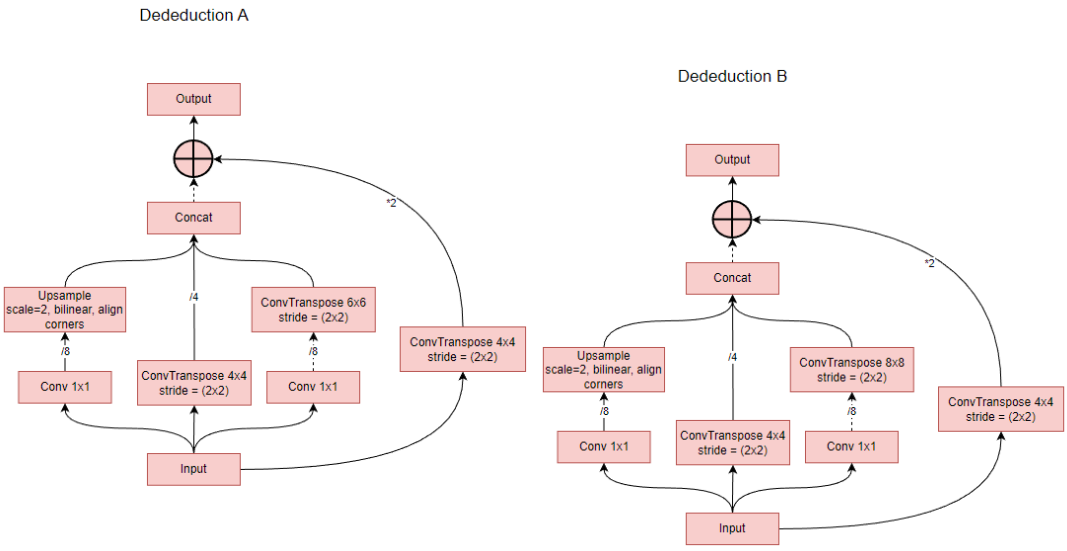


Figure 9.16: DeductionA and DeductionB modules of compression model

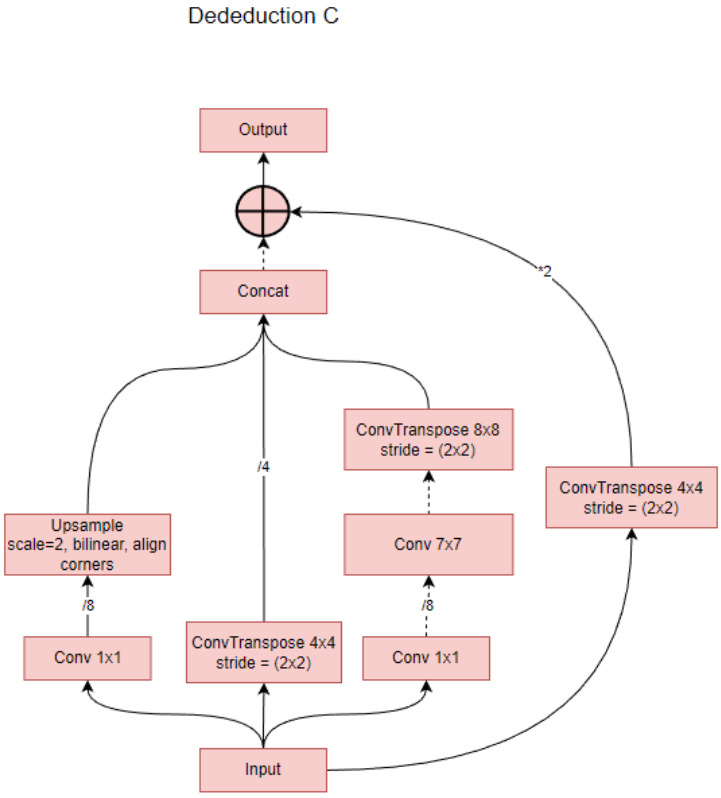


Figure 9.17: DeductionC module of compression model

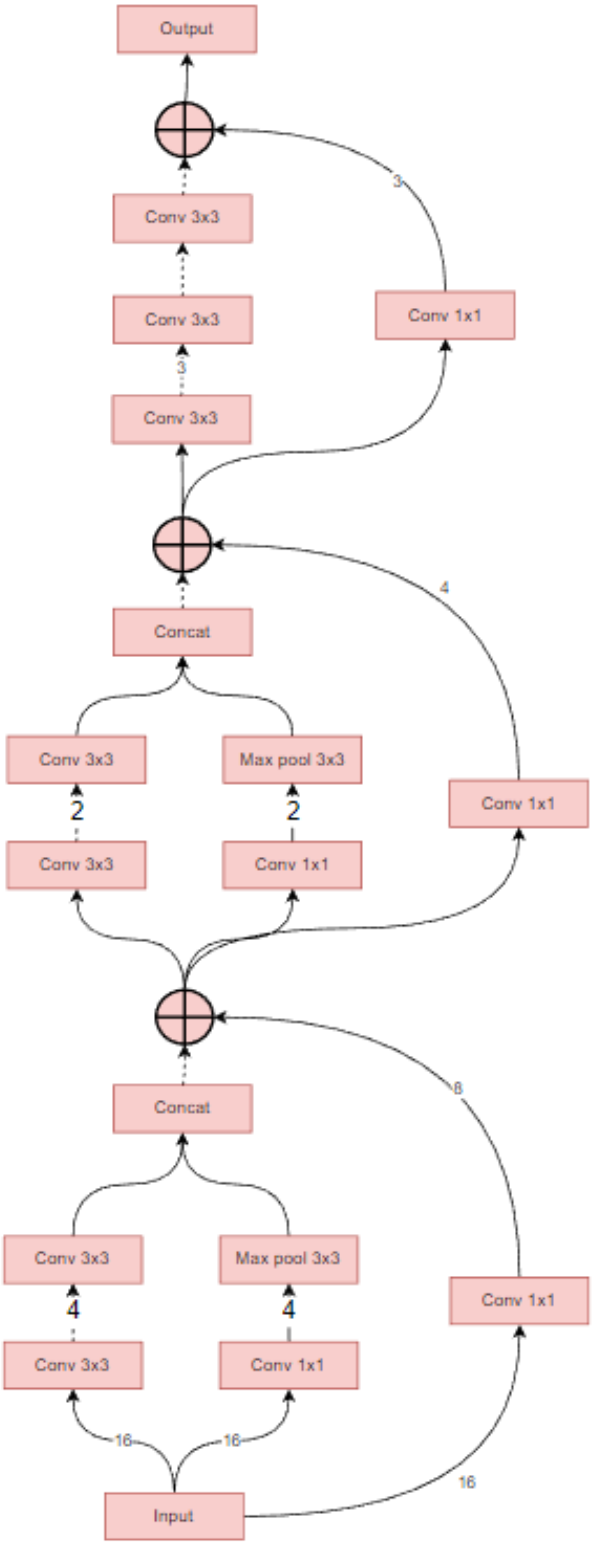


Figure 9.18: DeStem of compression model

Since we just reused the optimizations from the loss function, it is possible that some of them do not have the same beneficial effect for the compression architecture. However, due to time constraints, we did not prioritize performing the same tests again and instead assume that the methods from the loss function extrapolate well enough for the

compression architecture.

10 Implementation of JPEG and Arithmetic Encoding

10.1 Implementation choices

For the implementation of our neural network models we used python due to its large support for machine learning tasks. Therefore it would be convenient if our JPEG and arithmetic encoding implementations could interface with python. This would also be helpful for benchmarking and testing our models as loading and displaying images is quite straight forward in python. However arithmetic encoding and especially JPEG are both somewhat computationally intensive tasks and basic implementations of both would suffer from the subpar performance of python compared to other languages.
Therefore we have chosen to use the Rust programming language in combination with the python package `maturin`. Rust is a high-performance modern programming language with excellent built-in testing capabilities and `maturin` is a package that enables Rust libraries to easily interface with python by allowing the definition of PyFunctions and PyModules within Rust modules.

10.2 Structure

The JPEG and arithmetic encoding functionalities are implemented in the same Rust library, as JPEG makes use of arithmetic encoding and this simplifies the implementation process. The structure of the implementation is described below

- `lib.rs`
- `arithmetic_encoding.rs`
- `file_io.rs`
- `colorspace_transforms.rs`
- `dct.rs`
- `quantization.rs`
- `runlength_encoding.rs`
- `JPEGSteps.rs`

10.2.1 `lib.rs`

This file defines the interfacing functions between Rust and python. So it contains the functions that can be called from within python. It also contains some important data structures that are needed throughout the code.

10.2.2 `arithmetic_encoding.rs`

Includes all our arithmetic encoding functionality. Contains code for generating probability models, encoding and decoding.

10.2.3 `file_io.rs`

Contains functions for writing and reading differently encoded binary files. Holds functions that translate an arithmetic encoding, including the model and other necessary data, to a binary string.

10.2.4 `colorspace_transform.rs`
A small file containing simple functions that allows for transformation between RGB and YCbCr values.

10.2.5 `dct.rs`
Contains functions for performing the DCT and the inverse DCT.

10.2.6 `quantization.rs`
Contains functions for block quantization and generating quantization matrices based on the chosen Qf.

10.2.7 `runlength_encoding.rs`
Contains functions for performing run-length encoding and decoding.

10.2.8 `JPEGSteps.rs`
Collects several functionalities into more compact steps of the JPEG process and applies them to an image.

During this chapter we will not be describing the code in detail but rather highlight some important implementation details as the specific code details can simply be found in the code. The code has been written with focus on readability and therefore there are little to no abbreviations throughout the code.

10.3 Arithmetic Encoding
When implementing arithmetic encoding the algorithm described in chapter 4 is not an option. The interval very quickly becomes so small that floating point numbers don't have enough precision to correctly represent the interval. To solve this problem is not an easy task. It requires the introduction of several new concepts and the combination of the interval finding and binary encoding step.

10.3.1 The Finite Precision algorithm
The first step towards the finite precision algorithm is the switch to integers instead of floating point numbers. So now u and l are represented by integers interpreted as binary fractions instead of binary numbers. As result we can operate only on relevant bits of the fraction and shift out irrelevant bits.

When does a bit become irrelevant?
To understand why we can shift irrelevant bits we need to understand the behaviour of u and l. The key insight is to realize that u only ever decreases and l only ever increases and that l is always smaller than u. So if $u \leq 0.5$ ever becomes true, then both $u < 0.5$ and $l < 0.5$ are true. As a result the most significant bit (msb) of both u and l is 0 and will never change and is no longer relevant for our calculations. When $l \geq 0.5$ then the msb of both u and l is 1 and will never change.
Since our binary encoding must be between u and l we know that it must match all the bits that are fixed in either two of the above-mentioned cases.

Rescaling operations
Instead of simply shifting out the bits it is more intuitive to understand what is happening as a rescaling of u and l in order to keep the interval from becoming too small. To do this we must define the following constants:

- `PRECISION`: The number of bits available to represent u and l.
- `WHOLE`: $2^{\texttt{PRECISION}}$
- `HALF`: $\frac{\texttt{WHOLE}}{2}$

The rescaling operations then are as follows:

while $u < \texttt{HALF}$ or $l \geq \texttt{HALF}$ **do**
  if $u < \texttt{HALF}$ **then**
    $l = 2 \cdot l$
    $u = 2 \cdot u$
    output 0
  else if $l \geq \texttt{HALF}$ **then**
    $l = 2 \cdot (l - \texttt{HALF})$
    $u = 2 \cdot (u - \texttt{HALF})$
    output 1
  end if
end while

This way each rescaling operation doubles the size of interval. This process is illustrated in figure 10.1 and figure 10.2

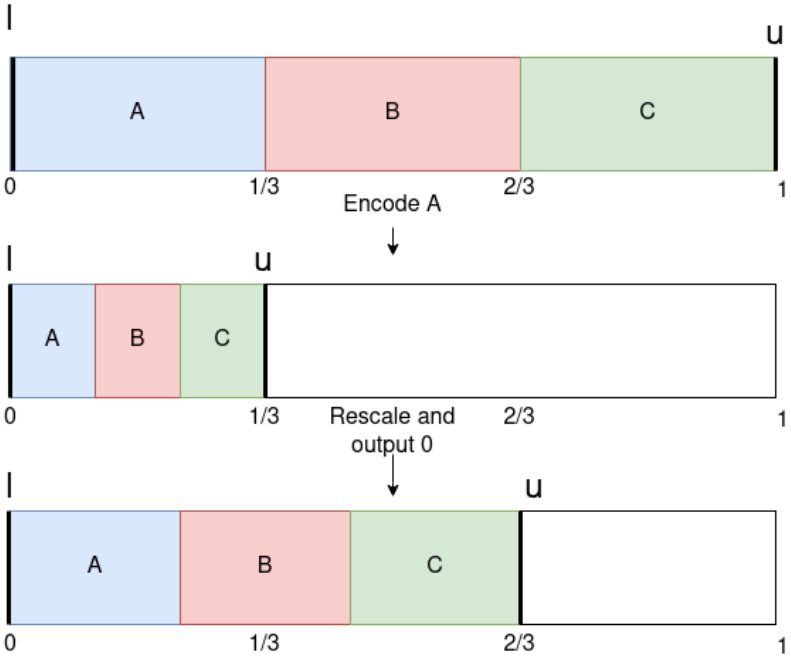


Figure 10.1: Rescale operation when $u < \texttt{HALF}$

However the algorithm is still not ready. The current rescaling operations rely on the fact that either u or l converge and cross past`HALF`. This is in no way guaranteed to happen. It is not difficult to imagine a scenario where the final values end up being $u = 0.5 + 10^{-100}$ and $l = 0.5 - 10^{-100}$. In this case rescaling must happen to not lose precision, but it wont happen with the current operations. So it is clear we must introduce at least one more rescaling operation.
When this problematic near-convergence state is encountered it must mean that both u and l are centered around `HALF`. In this state the leading two bits of u and l must be 10 and 01 respectively. If they continue to converge the leading bits become $100\ldots0$ and $011\ldots1$. There two possibilities when in this near-convergence state. Either u or l crosses `HALF` or the message has no symbols to encode. We start by covering what happens in the first case.

When $u < \texttt{HALF}$ happens then the two msb's of u must now be 01. In order for u to

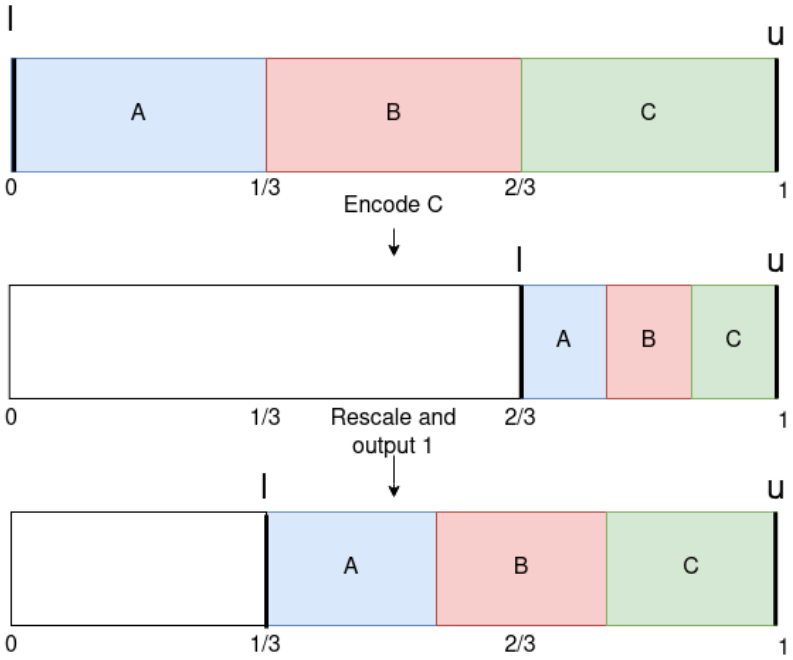


Figure 10.2: Rescale operation when $l \geq \texttt{HALF}$

remain larger than l it must have the same number of leading 1's after the first 0. When $l \geq \texttt{HALF}$ happens the two msb's must be 10 and it must have the same number of leading 0's after the first 1 to remain smaller than u. So as soon as we determine the value of the msb we know that the values of the other leading bits that are near convergence must be the opposite[57]. So we introduce a rescaling operation that essentially shifts out the second msb. It is important that we note how many times we perform this rescaling in a row as the value of these bits are pending until we determine the value of the msb. For this rescaling we introduce a new constant $\texttt{QUARTER} = \frac{\texttt{WHOLE}}{4}$. With this operation added our rescaling operations become:

while $u < \texttt{HALF}$ or $l \geq \texttt{HALF}$ **do**
  if $u < \texttt{HALF}$ **then**
    $l = 2 \cdot l$
    $u = 2 \cdot u$
    output_and_pending(0,*pending*)
    $pending = 0$
  else if $l \geq \texttt{HALF}$ **then**
    $l = 2 \cdot (l - \texttt{HALF})$
    $u = 2 \cdot (u - \texttt{HALF})$
    output_and_pending(1,*pending*)
    $pending = 0$
  end if
end while
while $u < 3 \cdot \texttt{QUARTER}$ and $l > \texttt{QUARTER}$ **do**
  $u = 2 \cdot (u - \texttt{QUARTER})$
  $l = 2 \cdot (l - \texttt{QUARTER})$
  $pending \mathrel{+}= 1$
end while

Where

```
function output_and_pending(bit,pending)
    output bit
    while pending > 0 do
        if bit = 1 then
            output 0
        else
            output 1
        end if
        pending -= 1
    end while
end function
```

This final rescaling operation is illustrated in figure 10.3

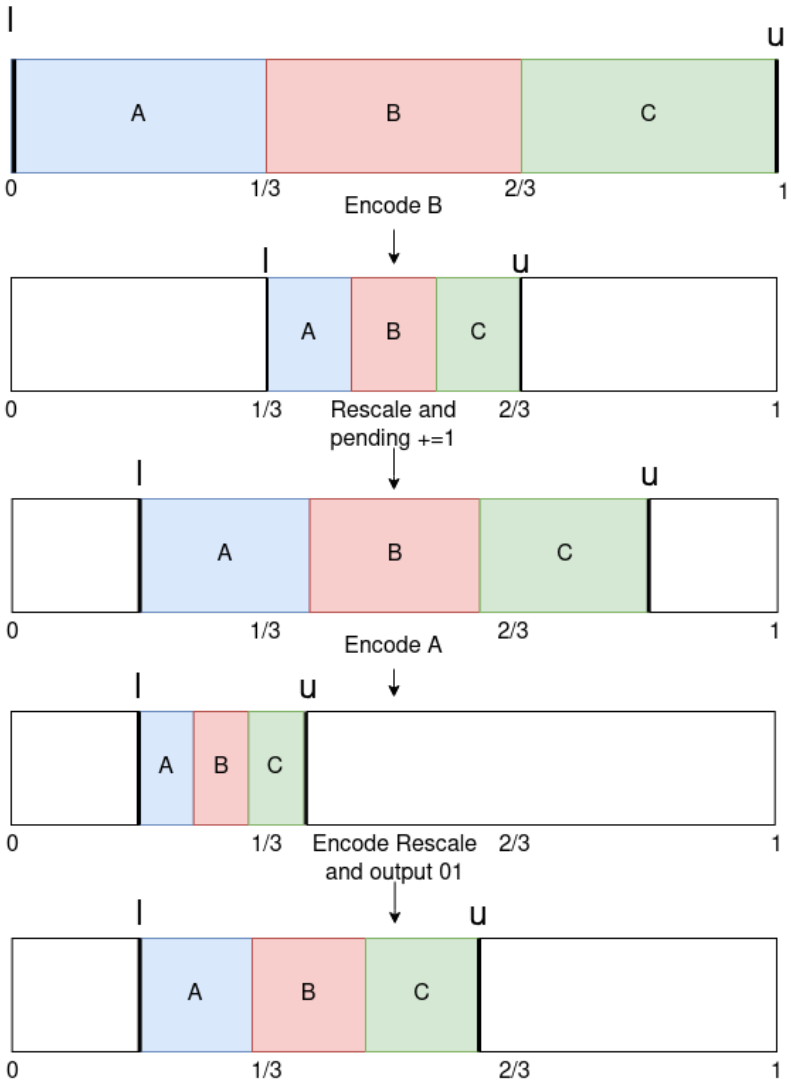


Figure 10.3: Rescale operation when $u \leq \frac{3}{4} \cdot \texttt{QUARTER} \wedge l \geq \texttt{QUARTER}$

When there are no more symbols to encode and no more rescaling operations can be performed our binary encoded string matches both u and l wherever they match each other, so finally we simply have to select the final bits so our binary encoding encodes an interval entirely between u and l. Since no rescaling can be made we know that

$u \geq \texttt{HALF} \wedge l < \texttt{HALF} \wedge (l \leq \texttt{QUARTER} \vee u \geq \texttt{QUARTER})$. First we increase *pending* by one then if $l \leq \texttt{QUARTER}$ we output the msb as 0 otherwise we output the msb as 1.

During decoding it is important that the rescaling is performed exactly in the same way as when the string was encoded. So the decoder is very similar to the encoder. Since every rescaling essentially shifts out a bit every time, this is also when a new bit is loaded from the encoded string.

Creating a model
To encode a message we need a model and since none is given to us we have to create one based on the message. Since we assume an i.i.d. model we can simply count the frequency of each symbol to get the probability for each symbol that matches our message the best. The model has to support the functionality of both the $I(s_i)$ and $S(\beta)$ mappings. The first is simply a HashMap with symbols as keys and range-pairs as values. The second is bit more tricky but is can be solved by having the symbols sorted by their ranges and then performing binary search in this array.

Time complexity
Since encoding and decoding are often performed at different points in time and rarely right after one another we shall analyse their time complexity separately. For analysis we will define

- $N =$ the number of symbols in the message
- $S =$ the number of different symbols in the message

During encoding we have to update u and l N times. Since we use a HashMap to find the ranges for each symbol this takes $\mathrm{O}(N)$ time. But we also have to perform rescaling operations for basically the entire length of the encoded result. This is dependent on the entropy of our resulting model. So the complexity becomes $\mathrm{O}(N + H(\texttt{model}) \cdot N)$.

To find an upper bound we will assume each of the S symbols appear equally often as this will maximize the entropy of the message and as a result also maximize the length of the encoded message. The probability for each symbol then becomes $\frac{N}{S} \cdot \frac{1}{N} = \frac{1}{S}$ Now we rewrite the entropy term as:

$$H(\texttt{model}) = \sum_{i=1}^{S} -\frac{1}{S} \log_2 \left(\frac{1}{S}\right)$$
$$=$$
$$\sum_{i=1}^{S} \frac{1}{S} \log_2(S)$$
$$=$$
$$S \cdot \frac{1}{S} \log_2(S) = \log_2(S)$$

So the time complexity now becomes $\mathrm{O}(N + \log(s) \cdot N) = \mathrm{O}(\log(s) \cdot N)$. In general it is clear that S must be upper bounded by N and as a result the general complexity for encoding becomes $\mathrm{O}(N \cdot \log(N))$. For our purposes we know that when compressing both the latent layer of our model and compressing the run-length encoded sequence from JPEG the number of different symbols is upper bounded by a constant so the time complexity becomes even smaller at simply $\mathrm{O}(N)$.

During decoding we have to decode N symbols and update u and l. It takes $O(\log(S))$ time to find the ranges per symbol. We have to perform the same amount of rescaling operations and can do the same rewrite of the entropy and therefore the complexity for decoding in our case becomes $O(\log(S) \cdot N + \log(S) \cdot N) = O(N)$ when s is a constant and $O(N \cdot \log(N))$ in the general case.

10.4 JPEG

The code in `colorspace_transform.rs`, `dct.rs`, `quantization.rs` and `runlength_encoding.rs` is quite simple and essentially just a direct implementation of the theory covered in chapter 5.
However there are details worth highlighting in `JPEGSteps.rs`.

10.4.1 `color_transform_and_dowsample_image`

This is the first stage of the JPEG process. It combines the color transform and sub-ample step into a single function. Instead of calculating all the Cb and Cr values and then subsampling, only the values of Cb and Cr that would remain after sampling are calculated. In the same step this function also ensures that the dimensions of all three channels each are divisible by 8 so they can be divided into block for later stages.
The transformed channels and some auxiliary data are collected into the `struct JPEGContainer`. The `JPEGContainer` contains the fields as shown below and is used to pass the relevant data from step to step

```
    pub struct JPEGContainer{
    y_channel : Vec<Vec<f64>>,
    cb_channel : Vec<Vec<f64>>,
    cr_channel : Vec<Vec<f64>>,
    original_size : (usize,usize),
    Qf : f64,
    sample_type : Sampling,
}
```

The results from the color transform are kept as floating point numbers for later stages instead of casting between integers and floating point multiple times. The original size is saved as padding may need to be removed during decoding. The sample type and Qf are needed for later stages and during decoding so they are also saved.

10.4.2 `parallel_function_over_channels`

For the remaining stages of JPEG each channel can be processed completely separately. Therefore it is rather simple to make the processing concurrent as there is no shared state by any threads. The function `parallel_function_over_channels` takes a function over a channel and applies it in parallel on each of the three channels. It allows for distinguishing between lumen and color channels and collects and returns any potential results from applying the function over the channels.

Rust has strict memory ownership rules to prevent memory mismanagement and errors and these are especially strict when it comes to sharing memory and ownership between threads. Therefore messaging channels are used to pass ownership of the channels between the functions and the `JPEGContainer struct`.

10.4.3 Time complexity

The complexity analysis for JPEG is rather simple. There are two parameters N and M being the width and height of the image. As each step in the JPEG process performs a

constant amount of operation per pixel the complexity of each stage is simply $O(N \cdot M)$. Even in the worst case with $Qf = 100$ where no compression happens, the number of symbols in run-length encoded is still upper bounden by $O(N \cdot M)$ and as result the arithmetic step is as well. So the complexity of both JPEG encoding and decoding is simply $O(N \cdot M)$

10.5 The python-rust Interface: `lib.rs`

The following functions from the module are available in python.

10.5.1 `arith_encode_latent_layer` and `arith_decode_latent_layer`

These are functions that enable entropy coding of the latent layer of our neural network model. The encoding function takes the latent layer as a vector and a path to a file to write the encoded binary strings. The decoding functions simply takes a path to the encoded file.

10.5.2 `JPEGcompress_and_decompress`

This function takes an image as an 3D vector, a Qf and a sampling type and returns a 3D vector containing the resulting image after JPEG has been performed with the specified parameters.

10.5.3 `JPEG_compress_to_file` and `JPEG_decompress_from_file`

These functions compress and decompress images to and from files. The compression function has the same parameters as `JPEGcompress_and_decompress` except it also takes a path to file it can write the compressed binary string to. The decompression function simply takes a path to a compressed file and returns the reconstructed image.

11 Results

11.1 JPEG

The JPEG implementation works as supposed, making us able to precisely select compression rates. This allowed us to compare the JPEG results with our results at comparable bit rates. Examples of compression using our JPEG implementation on an image from the Kodak data set[28] can be seen below:

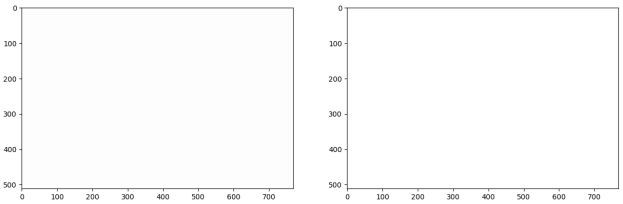
Figure 11.1: Original image

Figure 11.2: Compression factor of 6.26%, 3.55%, 1.45% and 0.29%, in the order top left, top right, bottom left and bottom right

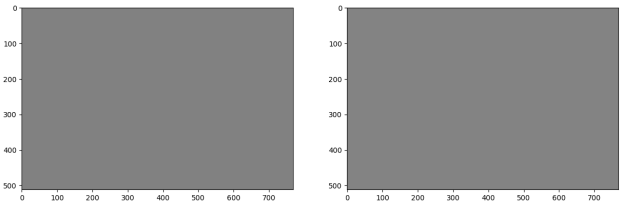
A clear quality reduction can be seen as the bit rate drops, one can also see all the expected attributes of the JPEG like blocking effects and similar.

11.2 Loss function

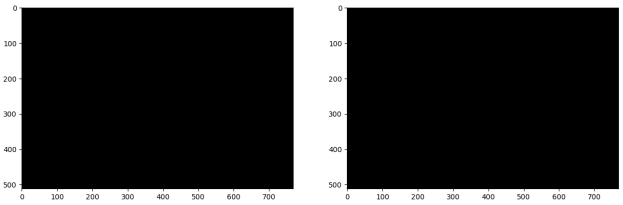
In order to see the performance of our loss function, we compare it with two of the most used loss functions in the field, the MSE and the SSIM. The pictures are passed to the loss functions in the range $[0, 1]$ not $[0, 255]$, therefor the MSE is in the range $[0, 1]$ where 0 is the best possible score, SSIM $[-1, 1]$ where 1 is the best possible score, and our loss function is from $[0, 1]$ where 0 is the best possible score. Since these loss functions do not adhere to any standard range or distribution of loss values, we will not directly compare the loss values of the different models. Instead we will compare them by indirect methods like comparing if images of similar visual appeal get a similar loss, or if we agree with the ordering of the losses. Here is a comparison on test cases presented in [19], where the loss of SSIM, MSE and our loss function respectively are shown under every image pair:



(SSIM: 1, MSE: 6.2e-5, Our: 3e-4)



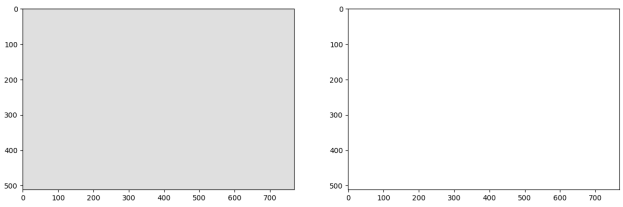
(SSIM: 1, MSE: 6.2e-5, Our: 5.09e-5)



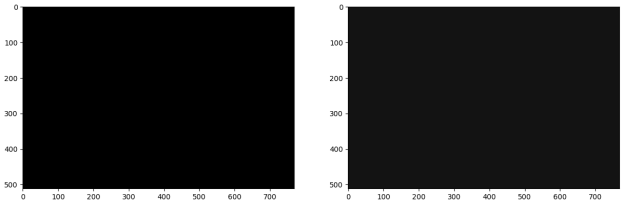
(SSIM: 0.62, MSE: 6.2e-5, Our1e-4)

Figure 11.3: From top left to bottom right, the images have all pixel values set to: 253, 255, 128, 130, 0, 2 in the scale from 0 to 255

In these examples, humans do not notice any significant difference between the images. Both the MSE and our loss function align well with the HVS in this case, while the SSIM seemingly sees a large difference between the two black images.



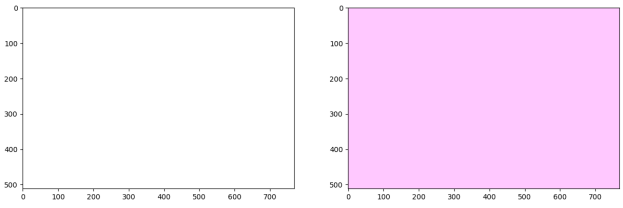
(SSIM: 0.99, MSE: 1.7e-2, Our: 0.35)



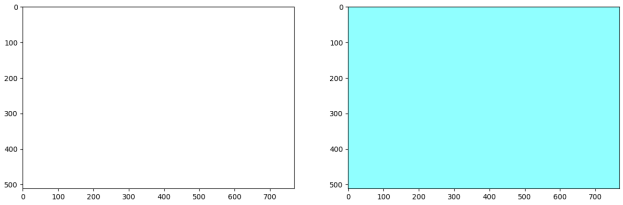
(SSIM: 9.5e-3, MSE: 1e-2, Our: 0.3)

Figure 11.4: From top left to bottom right, the images have all pixel values set to: 222, 255, 0, 26 in the scale from 0 to 255

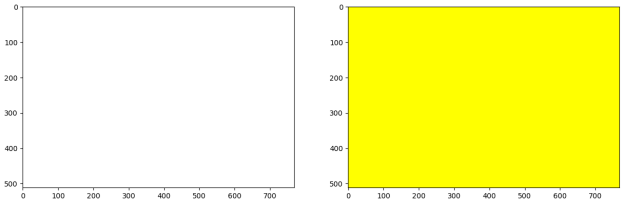
In this case humans generally prefer the difference between the two black images, while we find the difference between the two white images to be more noticeable. Again our loss function and the MSE agree with the HVS while the SSIM has the opposite opinion.



(SSIM: 0.95, MSE: 0.06, Our: 0.75)



(SSIM: 0.99, MSE: 0.02, Our0.47)

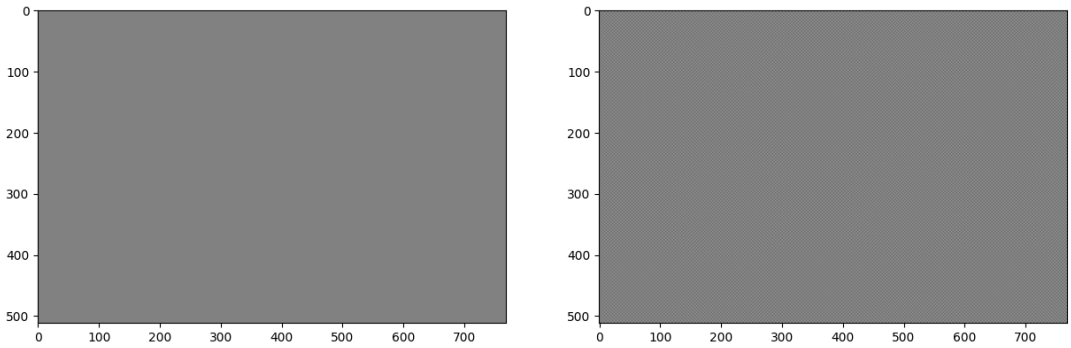


(SSIM: 0.67, MSE: 0.33, Our: 1)

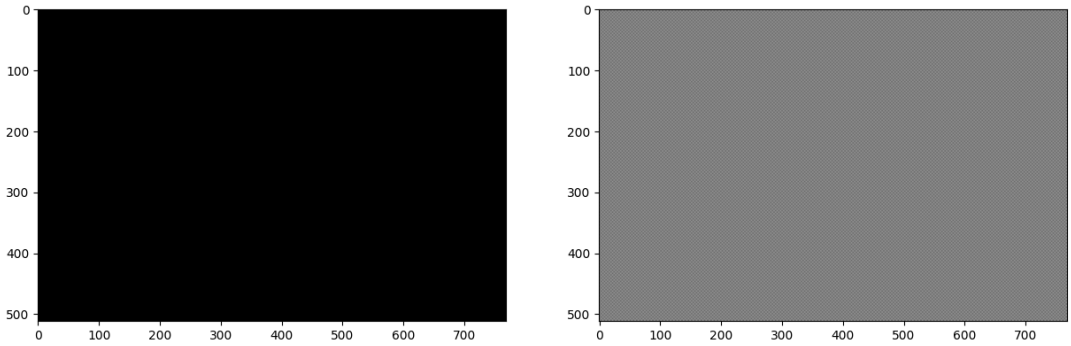
Figure 11.5: All the pixel values of the white images are 255, while the rgb for pink is (143, 255, 255), cyan is (255, 199, 255) and yellow is (255, 255, 0)

In this case humans notice a clear difference between all cases. One can argue that the yellow is especially far from the white but all are quite far. For the two first examples only our loss function agrees with the HVS and says that the images are very different. Only

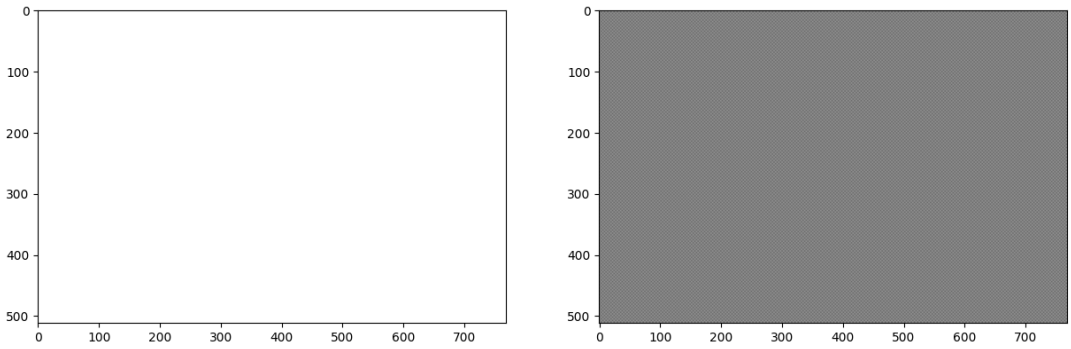
when comparing the yellow image the other loss functions catch up and say that there is a large difference.



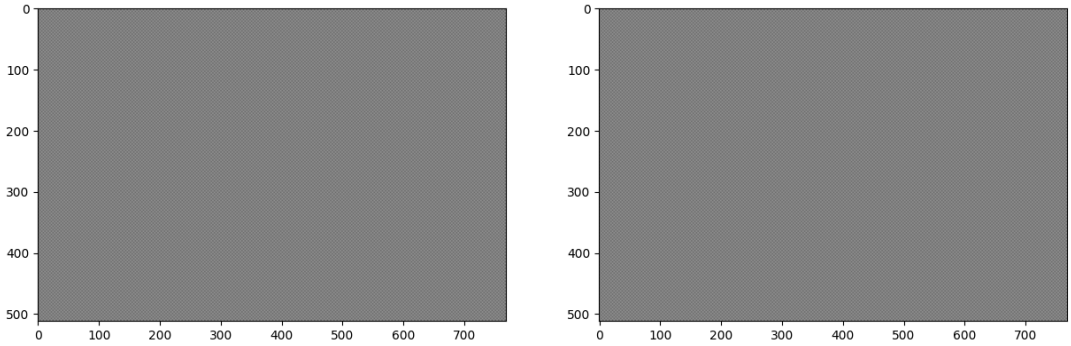
(SSIM: 3.6e-3, MSE: 0.25, Our: 0.52)



(SSIM: 1.4e-6, MSE: 0.5, Our: 0.78)



(SSIM: 2.9e-3, MSE: 0.5, Our: 0.99)

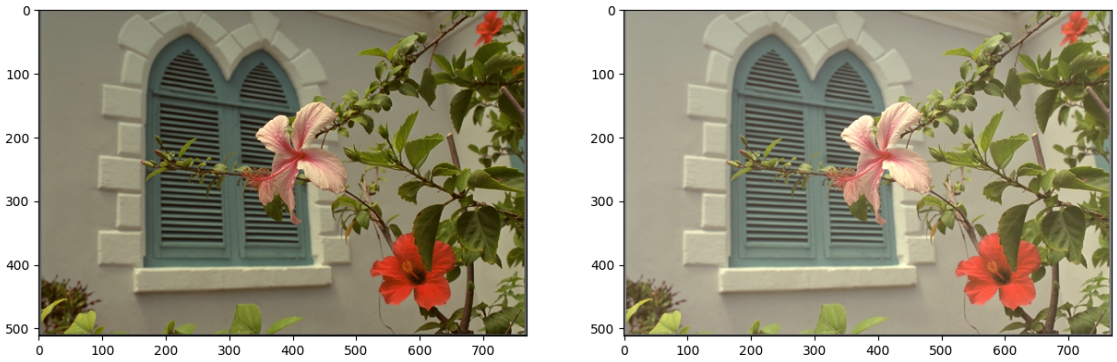


(SSIM: -1, MSE: 1, Our: 0.96)

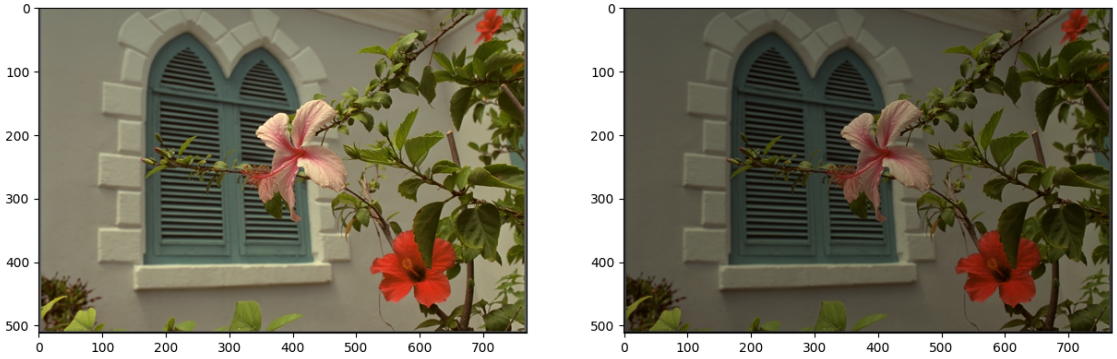
Figure 11.6: All the images on the right hand side have a chess board pattern (black and white tiles), the left hand side are gray 128, black 0, white 255 and a flipped chess board.

One can argue about how the HVS would rate these images, since it depends highly on how the images are presented. On a large screen, the black and white tiles become noticeable, but on a small screen it is indistinguishable from a continuous gray color. In the size displayed here, one should likely give the first and last images a high rating, since they all look gray, while the middle once should get a lower rating. To this extent, none of the loss functions perform well.

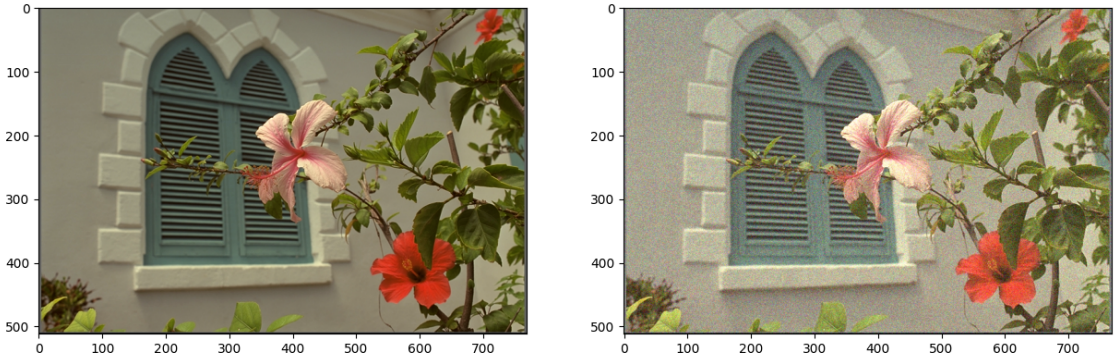
Our loss function would likely perform better on this task, if similar cases with translation or periodic patterns were added to the data set, but one can doubt how likely these artificial examples are to occur in the real world. Furthermore these examples from [19] do not contain much structure, which might explain why the structural similarity index measure SSIM performs poorly. To compare the functions on more realistic and structural images, we use the Kodak data set[28], and apply some of our augmentations on the images. All images have an approximately constant MSE in the range $0.0199 \sim 0.0205$ (the images pixel range is from 0 to 1 not 0 to 255) :



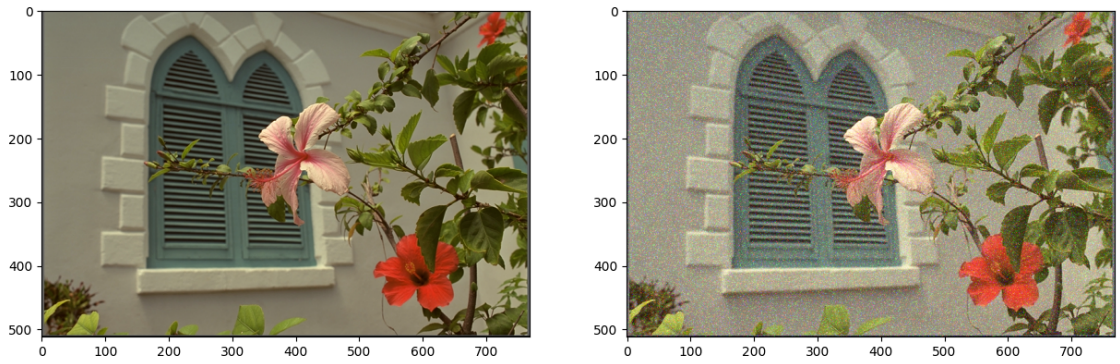
(SSIM: 0.93, MSE: 0.02, Our: 0.34)



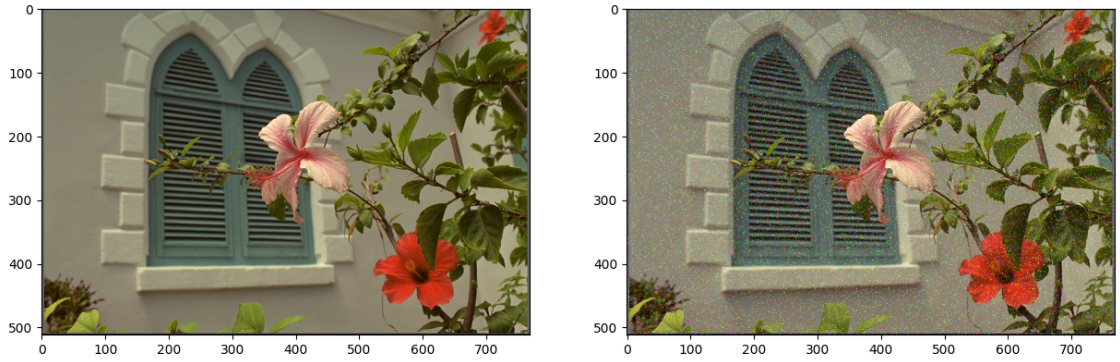
(SSIM: 0.9, MSE: 0.02, Our: 0.4)



(SSIM: 0.37, MSE: 0.02, Our:0.48)



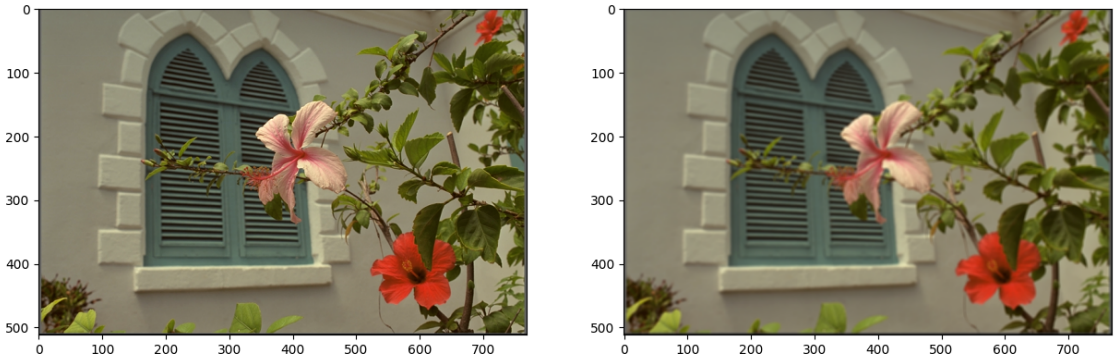
(SSIM: 0.29, MSE: 0.02, Our:0.52)



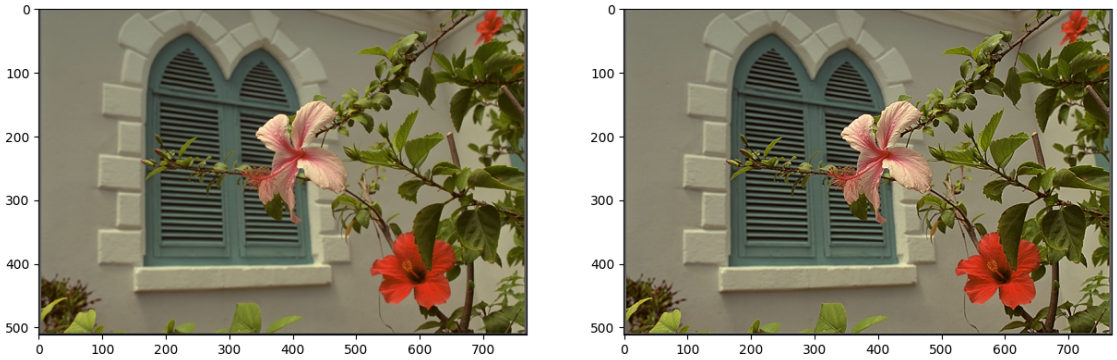
(SSIM: 0.29, MSE: 0.02, Our: 0.6)

Figure 11.7: Adding, multiplying, poison noise, exponential noise and salting, all with a constant MSE of ~ 0.02

The HVS clearly sees differences in the quality of these images (especially on a larger screen) despite them all having very similar MSE. The SSIM and our loss function agree on which images are the best (we have sorted the images in that order from good to bad), and we think that this ordering aligns well with the HVS. We have found that a clear weakness of the MSE is that it rates blurry images highly, since it only focuses on the pixel by pixel difference and does not make a more holistic review of the image. Here are two attempts to show this weakness:



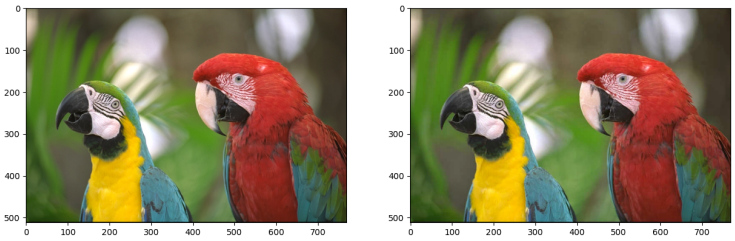
(SSIM: 0.85, MSE: 1.7e-3, Our: 0.41)



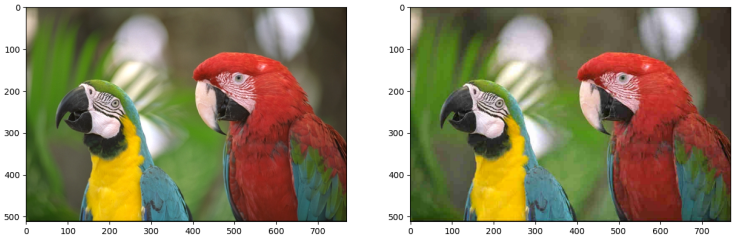
(SSIM: 0.96, MSE: 7e-4, Our: 0.03)

Figure 11.8: Blur filter and sharpening filter

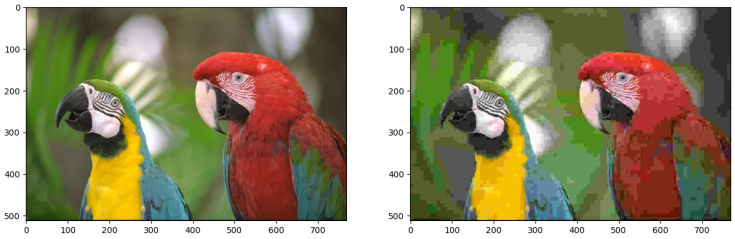
Here the MSE clearly misjudges the blurry image, while the SSIM and our loss function generally agree that the blurry image is quite poor. We personally think that the SSIM is rating the blurry picture too highly, and that our loss function aligns best with the HVS for these cases. However, this might be expected, since our loss function is trained on similar transformations. To see how well our loss function extrapolates to other transformations, we also compare the loss functions on images compressed with JPEG:



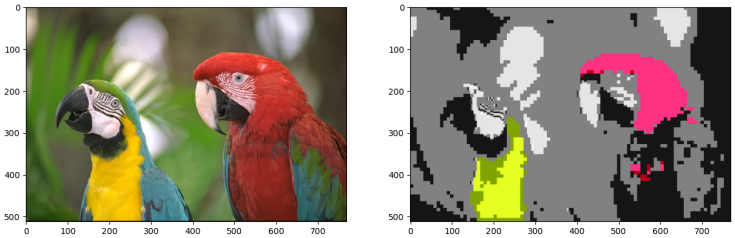
(SSIM: 0.92, MSE: 2e-4, Our: 5.3e-2)



(SSIM: 0.87, MSE: 6e-4, Our: 5.4e-2)



(SSIM: 0.73, MSE: 2.3e-3, Our: 0.15)



(SSIM: 0.59, MSE: 3.2e-2, Our: 0.56)

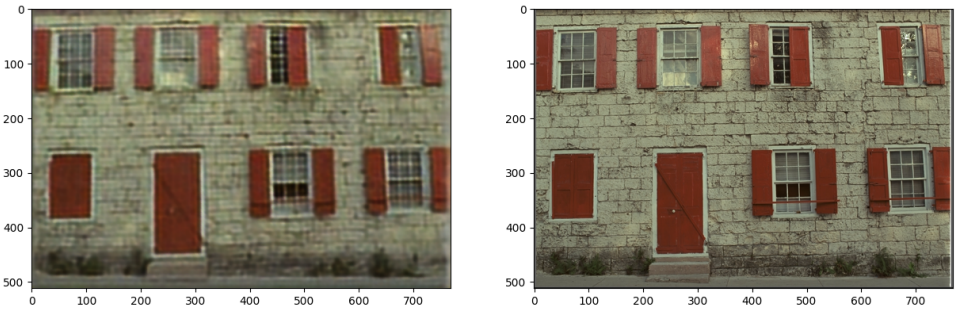
Figure 11.9: Left side is the original, right is the corresponding JPEG compression with the quality factors 50, 20, 5, 1

Where the first two images seem very similar, but the quality quickly drops on the last two images. All loss functions get the ordering right in these examples. This shows that even on a quite limited set of augmentations, our loss function does not seem to have over fitted on our augmentations and is able to extrapolate to unseen augmentations quite well. However, the loss functions do not punish the last image enough, since contrary to the evaluation of the SSIM and our loss function, it is clearly deformed more than the salted image from figure **??**.

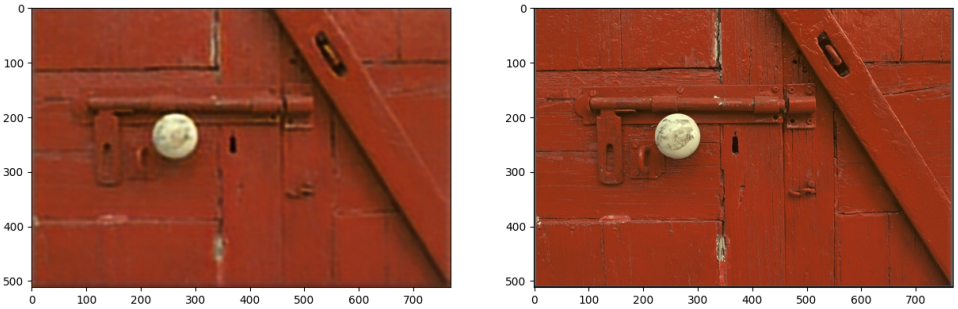
All in all our loss function seems to generally align better with the HVS compared to the MSE and the SSIM.

11.3 Neural compression

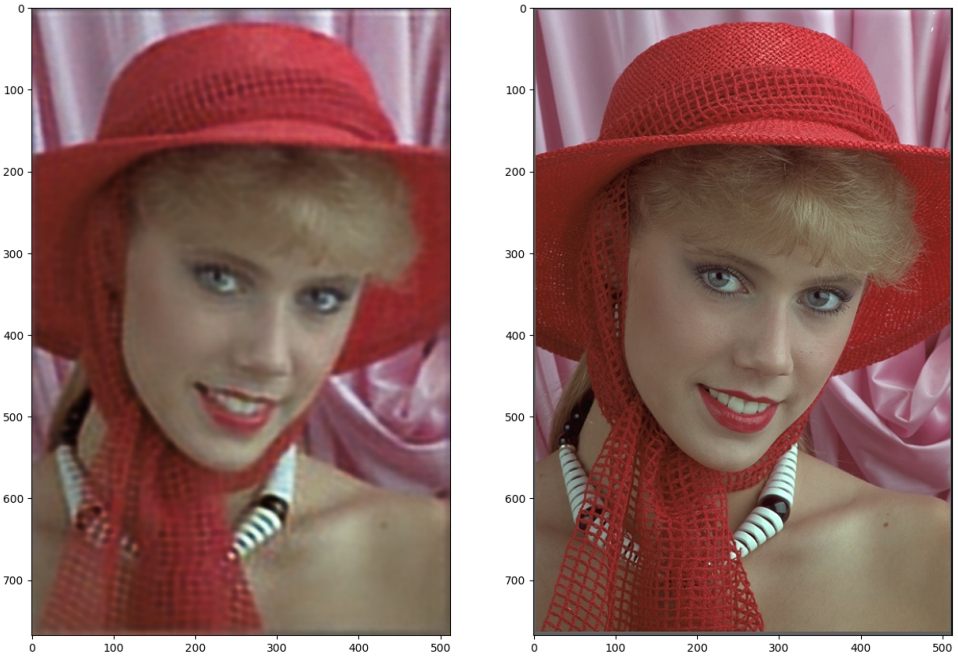
To see the comparative performance of our compression model, we compare our results with our implementation of JPEG at similar bit rates. First we show the results of the compression model that was trained with the MSE loss function:



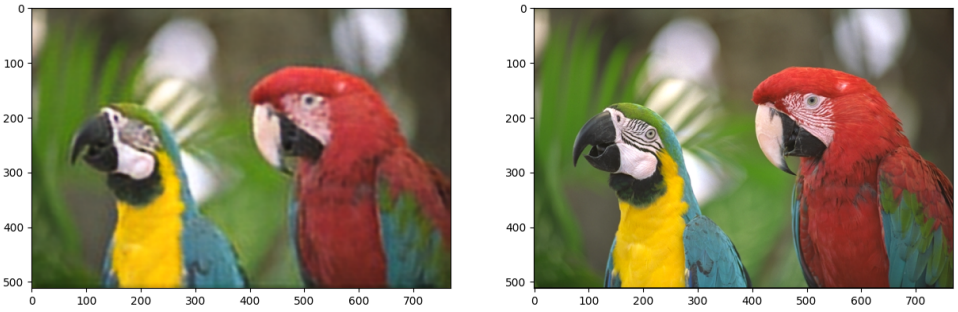
Compression factor of 8.6%



Compression factor of 8.0%



Compression factor of 8.3%



Compression factor of 8.5%

Figure 11.10: Kodak images compressed with our auto encoder model

Figure 11.11: Same images compressed with JPEG $Qf = 50$. From the top left the compression factors are (8.64%, 6.06%, 6.26% 4.98%)

As is clear, the JPEG seems visually superior. However, the main difference is that the reconstruction made by our compression model is more blurry. As mentioned in the section above 11.2, this seems to be a general problem with the MSE loss function, since the MSE does not punish blurry images much. This could lead to our compression model to favor blurry images over images that lose other information. In our attempt to solve this problem, we also trained our model with our loss function. Here we show the progression of the training when using our loss function:

Figure 11.12: Compression model training progression from top left to bottom right

The model initially seams to be learning well, since one starts to see the white building (the initial progress looks very similar to the initial MSE progress). However it is clear that the final results are not visually pleasing. The visual ”quality” is not caused by training difficulties, in fact our compression model thought it did a near perfect job, since the loss function gave losses very close to zero.

There are a few possible reasons why the results were so poor. We believe that the most likely one is that our loss function simply has not seen images similar to the ones that our compression model made, leading it to be unable to judge it correctly. By having a circular training scheme as mentioned in section 6.2 and will be covered further in subsection 12.1.2, one could make the loss function continuously learn the relevant input data space. However, since our compression model has more parameters, it is likely more powerful and flexible than our loss function. As stated by Eliezer Yudkowsky in the interview[58]:

> *”When the verifier is broken, the more powerful suggester just learns to exploit the flaws in the verifier”* -Eliezer Yudkowsky

If the task of aligning well with the HVS is too difficult a task for our loss function, then our more powerful compression network might exploit this weakness. This case would make it difficult for our proposed circular learning scheme to succeed. We think it is an unlikely case, since our compression network has a bottle neck layer that should make it less flexible compared to what the number of parameters might suggest. Furthermore, our experience has been that it was generally harder to train the compression network compared to the loss function, if this experience extrapolates well, then it should be possible to align very

well with the HVS and avoid exploitability.

12 Further work

12.1 Improvements to our method

12.1.1 Simpler model

We now believe that we were somewhat naive in thinking that we could get good results without much trial and error just by learning from successful projects. We simultaneously tried to implement a whole lot of tricks to improve performance. Even though it is possible to use these techniques to improve performance, we had difficulties knowing what impact each of them had separately and how they interacted with each other. Furthermore, it was quite difficult to tinker with the model, since there were so many different hyper parameters, making the search of possible beneficial changes difficult. We might have gotten better results by having a simpler implementation and tuning it well. This simpler approach would still leave open the opportunity for incremental future optimization. We were reminded of a quote from Donald E. Knuth [59]:

"Premature optimization is the root of all evil." -Donald E. Knuth

If we were to restart the project from scratch, we would have liked to start by having a simple model, and then perform some kind of grid search or Bayesian optimization in order to find a great architecture and the right hyper parameters. This would be particularly beneficial if there was more time, since grid searches and Bayesian optimization are time consuming. This would also lead to a better understanding of which parts really affect performance.

12.1.2 Loss function further training

The single thing we most would have liked to have done differently is the training process. As mentioned in section 6.2, we wanted to make a loss function that aligns well with the human visual system when it comes to rating a reconstruction of an image. In order to get a data set of reconstructions of images, we implemented augmenting functions that alter the images. But because of the cost and time constraints, we found it infeasible to get human evaluation of these augmentations. Instead we chose to just rate the augmenting functions ourselves, which was quick and in our opinion quite an okay approximation. But, the choice did lead to some of the images being rated in a way that does not align well to the human visual system. One such example can be seen below, where the same blur augmentation clearly does not have the same visual impact on both images:

Figure 12.1: Blur transform on sharp and blurry image

Furthermore, it is likely that the distribution of our transformations is quite far removed from the distribution of reconstructions of the compression network. This covariate shift might cause our loss function to be unable to rate the reconstructions properly. Inspired by Deep Reinforcement Learning from Human Preferences[29], we would have liked to use a more circular training structure as depicted in the figure below:

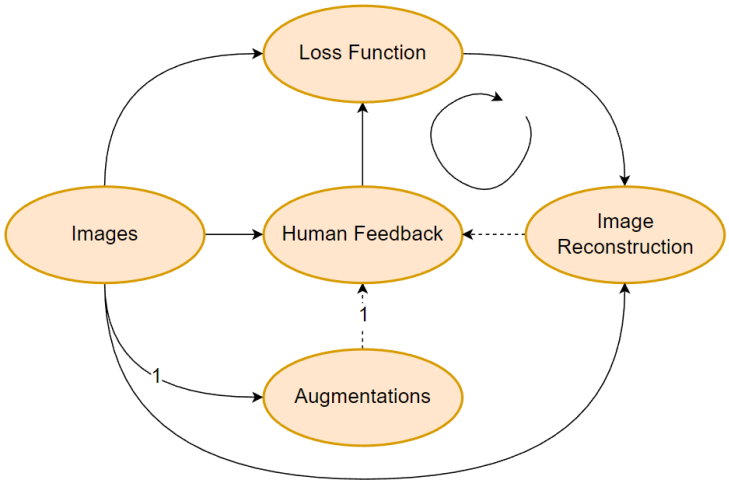


Figure 12.2: Training scheme inspired by Deep Reinforcement Learning from Human Preferences

Where the 1 means that the connection is only for round one and the dotted arrow means that the humans do not necessarily need to rate all the images in every round. The circular arrow only emphasises the cycle in the graph. In [29] they found that one only needs to

rate about 1 for every 1.000 data points, since the loss function was able to learn the human preferences by only seeing a subset of the ratings. The proportion of the images that humans need to rate does likely depend on the model, task, data and more.

Applying this circular training strategy would allow the loss function to get better at rating the output from the compression network, since it will be able to train on human feedback of the image reconstructions. According to [29], one human could in their example rate about 5.000 1-2 second video clips in 5 hours. Since we have still images and not video clips, 5 hours would likely be an upper bound for our case. Assuming that the average video was 1.5 seconds, then $\frac{5000 \cdot 1.5}{5 \cdot 60 \cdot 60} \approx 42\%$ of the time was spent on running the videos, which for our case would be reduced to 0. Thus if we wanted people to rate 50.000 images in each round for 5 rounds, it would cost at most 250 hours of labour and more likely about ~ 200 hours.

We believe that this training scheme would create a loss function that is very well aligned with how humans see images, making it possible for the compression network to exploit the intricacies of the HVS to compress images in a way that looks nice for human observers. The loss function might also become more useful in other projects that use other reconstruction techniques.

12.1.3 Jointly optimizing distortion and bit rate

In our implementation, the auto encoder only cares about the reconstruction error after rounding and not the distribution of the latent layer. It is clear from the theory covered in chapter 2 that the amount of symbols in the latent layer and their distribution has a direct impact on the bit rate of the arithmetic encoder. Thus it would be beneficial to jointly optimize the reconstruction error and the latent layer distribution as can be seen on the figure below:

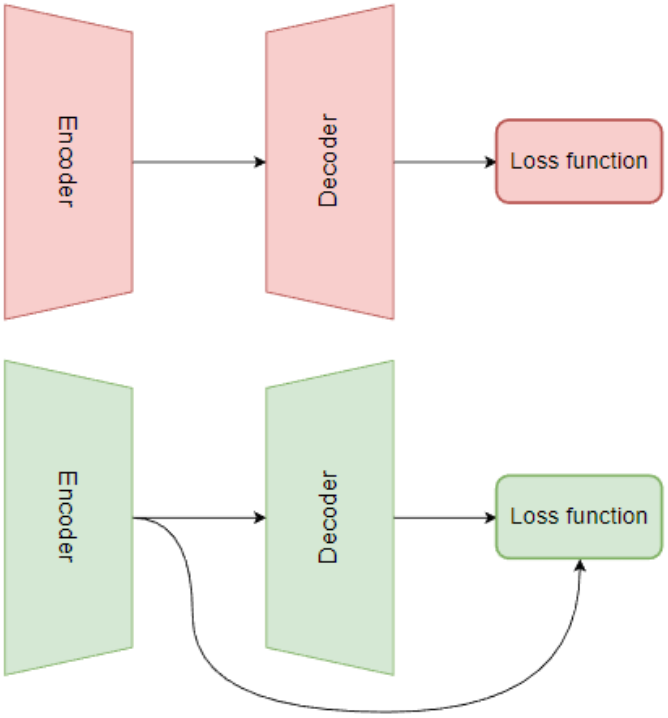


Figure 12.3: The one in red is the forwards pass in our current training scheme, while the one in green is the preferred method that includes the latent layer distribution in the loss function

Said in an other way, we would like to change our loss function from:

$$D$$

Where D is the distortion of the image after reconstruction, to:

$$R + \lambda D$$

where R is the bit rate, and λ is the trade off factor between quality and compression.

We are not sure what the best way to extract R from the latent layer distribution is, since the procedure of figuring it out directly involves rounding which would inhibit gradient decent. One could use the variance of the latent layer distribution as a proxy, since low variance would mean that the latent layer distribution is clustered closely together. One could also use the absolute range of values.

Since unevenness in the distribution leads to better compression rates, these proxies would not capture the complete picture, and it could be beneficial to find some other approach.

λ would then be a hyper parameter, which must be optimized in order to find a good trade of between quality and compression rates.

12.1.4 Neural latent layer distribution predicter

Our current implementation of the entropy encoder needs to know the distribution of the latent layer in order to compress and decompress the image. This means that the distribution is sent along with the compressed image, which increases the bit rate. It is possible to mitigate this requirement by having a model that can predict the distribution

based on the entropy encoded string. This could be done jointly with the approach from section 12.1.3, by making the optimization favour not only a shorter range of values in the distribution, but also a more fixed distribution across images. Then one would not have to send the distribution each time, since the distribution predictor could be a part of the compressor decompressor scheme.

We are not sure of the best way to implement this, neither when it comes to adding it to the loss function nor what type of model would be preferable for prediction. One way to try to solve both issues at the same time is by changing the skip connection in the latent layer to:

$$y = ELU(F(x, W_i) + x)$$

and making $R = \Sigma(e^{x+1})$ for $x \in L$ (or some other monotonically increasing function) where L is the latent layer parameters. Because the ELU activation function is used, L should only contain numbers larger than -1. This would make the loss function favor values in L to be as low as possible while the ELU guarantees that the value is no smaller than -1. One could then try to predict the distribution by counting the amount of different values in the distribution, and guess that the most populous group is likely the -1 group, the second one 0 ... since this would cause the lowest value of R.

We do not know if this change to the skip connection would affect training, but we think that since the change only takes place in one layer, it would likely not affect the training too much.

12.1.5 Layerwise training

We trained our compression network end to end, but this might not be the best method. According to [8], one can more easily train a deep auto encoder by taking a layer by layer pretraining approach. This would also allow us to take a more data oriented approach, where we choose the size of the bottle neck layer based on when we see that the auto encoder starts to have trouble reconstructing the image. The downside of this approach is that one needs to train the auto encoder multiple times (once for each layer) making it more cumbersome and time consuming to train. Furthermore we think that our careful architecture design might bring with it some benefits similar to layer by layer pretraining, especially the residual connections might make it easier for the network to converge[41].

12.2 Limitations and other methods

Using an auto encoder to compress images has some weaknesses that seem difficult to overcome. Two weaknesses that were highlighted in [60] are that one auto encoder only works on one image size and that auto encoders can not perform variable rate compression. [60] also mentioned that it is hard to ensure visual quality with auto encoders, but this seams to be a more general issue with lossy compression and not an auto encoder specific issue.

In less general cases (e.g. if one had to make a compression algorithm for a specific camera or phone), these issues do not seem to be all that relevant, since one knows the image size and one can have a few models to give a few different compression rates if necessary.

It is possible to somewhat mitigate the problems in the general case. One could have multiple auto encoders (maybe doubling the size of the image for each auto encoder) in combination with padding the images. Multiple auto encoders would take up more space, be harder to train and would likely reduce to compression rate in the case where one uses a larger auto encoder in conjunction with padding. One could also section larger images and compress these sections individually, but this would probably lead to blocking effects.

In our preproject seen in appendix A and B we managed to get variable rate compression using what we called Modular auto encoders. This is illustrated below:

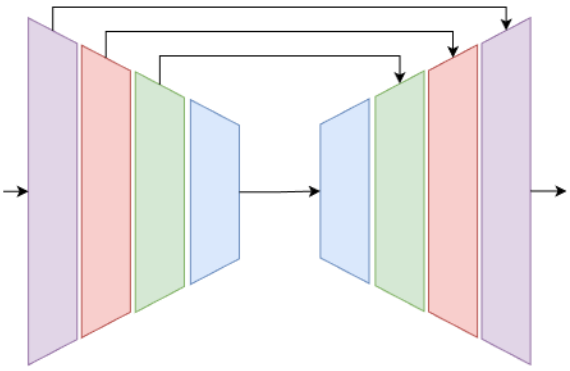


Figure 12.4: Modular auto encoder

Where the Modular auto encoder is trained to be able to follow any of the arrows on the image above, thereby getting different levels of compression depending on the path chosen. This method worked somewhat, but we were not able to get great visual results. We think that the training procedure was the main cause of the lack of quality, as we were not able to tune the model much due to time pressure, and we did not have much data (only the Kodak data set of 24 images [28]).

One could possibly also train an even more modular architecture that can take in images of different sizes and compress them to various degrees. A proposed model architecture is shown below:

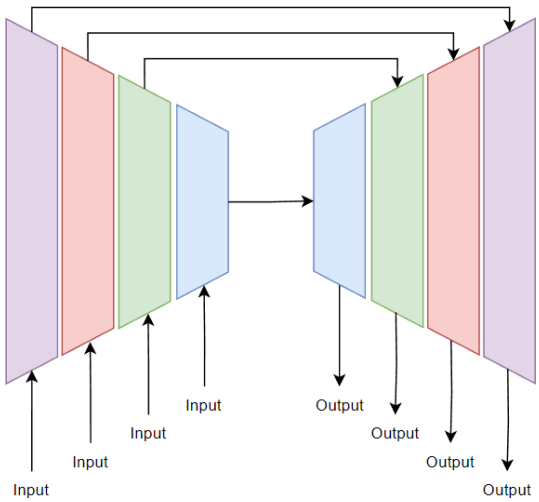


Figure 12.5: Even more modular auto encoder

Where if the model gets an image as input to the red encoder, then it tries to reconstruct the image in the red decoder.

There are other methods that solve some of these problems, but we have not found any silver bullet, as most of the methods seem to have drawbacks. One notable example is presented in [60], where they use recurrent neural networks (RNN's) with long short term

memory (LSTM). They managed to get performance that was superior to JPEG with 32x32 images. However, since the RNN LSTM based architectures have been shown to have problems remembering long sequences[61], it is not obvious that such techniques generalize well to larger image sizes.

Another exciting method is Implicit Neural Representation (INR), where one overfits a neural network to an image, and thereby encodes the image directly onto the network weights and biases[62]. One of the downsides of INR is that overfitting a neural network takes quite some time, multiple orders of magnitudes more time than JPEG[62].

In short, the field has open problems and many possible ways to try and tackle them.

13 Conclusion

Our goals for this project were to learn about image compression, especially image compression that makes use of neural networks. Furthermore we wanted to put the gained knowledge into action, by making our own compression algorithm that makes use of neural networks.

Theoretically we managed to get a good overview of the fields of information theory and neural compression. Firstly this made us able to implement entropy encoding and the well known JPEG compression algorithm from scratch. Furthermore this also made us able to come up with our own compression scheme based on neural networks. Our compression method worked well, but was not better to the currently most used method JPEG.

Our loss function seems to align better with the HVS compared to the commonly used SSIM and MSE. Given these attributes, we hoped that it would be generally useful for other people that need to measure image quality and train neural networks. However, as of now, it is not fit for training compression networks, since our compression network easily managed to exploit the weaknesses of our loss function.

We think that it is not the general idea of our compression model nor loss function model that limits performance, but rather the time constraints that made it hard to optimize the quite complex models and the financial constraints that made it infeasible to get human rated image reconstructions for circular training schemes. Funding could also speed up training by getting access to more compute.

The field of neural compression is still full with exciting challenges and open problems. We think that it is possible to build on the ideas presented here to gain performance and there are also many other avenues that might result in great compression results in the future. Hopefully the field will continue to advance, leading to better utilization of infrastructure and increased connectivity and storage space for people and companies.

Bibliography

[1] et al Hu Yueyu. "Learning end-to-end lossy image compression: A benchmark". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).

[2] et al Mark Adler. "PNG (Portable Network Graphics) Specification Version 1.0". In: *Dr. Dobb's Journal* (1995).

[3] C. E. Shannon. "A Mathematical Theory of Communication". In: *Bell System Technical Journal* (1948).

[4] *Oxford english dictionary*. URL: https://www.oed.com/. Accessed: 29-5-2023.

[5] *Redundancy (information theory)*. URL: https://en.wikipedia.org/wiki/Redundancy%5C_(information%5C_theory). Accessed: 29-5-2023.

[6] *Shannon's source coding theorem*. URL: https://en.wikipedia.org/wiki/Shannon%5C%27s%5C_source%5C_coding%5C_theorem. Accessed: 29-5-2023.

[7] et al Yassine Ouali. "An Overview of Deep Semi-Supervised Learning". In: (2020).

[8] R .R. Salakhutdinov G. E. Hinton. "Reducing the Dimensionality of Data with Neural Networks". In: *Science magazine* (2006).

[9] et al Christian Szegedy. "Rethinking the Inception Architecture for Computer Vision". In: *Computer Vision and Pattern Recognition* (2015).

[10] et al Johannes Ballé. "END-TO-END OPTIMIZED IMAGE COMPRESSION". In: *conference paper at ICLR* (2017).

[11] *Arithmetic coding*. 2023. URL: https://en.wikipedia.org/wiki/Arithmetic_coding. Accessed: 24-5-2023.

[12] *(IC 5.4) Why the interval needs to be completely contained*. URL: https://www.youtube.com/watch?v=jHS8-rmEo5k. Accessed: 29-5-2023.

[13] et al Graham Hudson. "JPEG-1 standard 25 years: past, present, and future reasons for a success". In: *Journal of Electronic Imaging* (2018).

[14] et al Mark Adler. "THE OPTIMAL QUANTIZATION MATRICES FOR JPEG IMAGE COMPRESSION FROM PSYCHOVISUAL THRESHOLD". In: *Journal of Theoretical and Applied Information Technology* (2014).

[15] *JPEG*. 2017. URL: https://da.wikipedia.org/wiki/JPEG. Accessed: 24-5-2023.

[16] B. Girod. "What's Wrong with Mean-squared Error?" In: *M.I.T. Press* (1993).

[17] et al Zhou Wang. "Image Quality Assessment: From Error Visibility to Structural Similarity". In: *IEEE Transactions on Image Processing* (2004).

[18] et al Zhou Wang. "MULTI-SCALE STRUCTURAL SIMILARITY FOR IMAGE QUALITY ASSESSMENT". In: *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers, 2003* (2003).

[19] Tomas Akenine-Möller Jim Nilsson. "Understanding SSIM". In: *arXiv:2006.13846* (2020).

[20] Gregory K. Wallace. "The JPEG Still Picture Compression Standard". In: *IEEE Transactions on Consumer Electronics* (1992).

[21] Eric Hamilton. "JPEG File Interchange Format version 1.02". In: *IEEE* (1992).

[22] et al Jiang Wang. "Learning Fine-grained Image Similarity with Deep Ranking". In: *IEEE Conference on Computer Vision and Pattern Recognition* (2014).

[23] et al Sumit Chopra. "Learning a Similarity Metric Discriminatively, with Application to Face Verification". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2005).

[24] Stephen Balaban. "Deep learning and face recognition: the state of the art". In: *Biometric and Surveillance Technology for Human and Activity Identification* (2019).

[25] et al Jingtou Liu. “Targeting Ultimate Accuracy: Face Recognition via Deep Embedding”. In: *Computer Vision and Pattern Recognition IEEE* (2015).
[26] et al Florian Schoff. “FaceNet: A Unified Embedding for Face Recognition and Clustering”. In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2015).
[27] et al Yaniv Taigman. “DeepFace: Closing the Gap to Human-Level Performance in Face Verification”. In: *IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2014).
[28] *Kodak data set*. URL: https://www.kaggle.com/datasets/sherylmehta/kodak-dataset?select=kodim04.png. Accessed: 24-5-2023.
[29] et al Paul F Christiano. “Deep Reinforcement Learning from Human Preferences”. In: (2023).
[30] et al Andrew G. Howard. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: *Computer Vision and Pattern Recognition* (2017).
[31] et al Christian Szegedy. “Going deeper with convolutions”. In: *IEEE Conference on Computer Vision and Pattern Recognition* (2015).
[32] Christian Szegedy Sergey Ioffe. “Batch Normalization: Accelerationg Deep Network Training by Reducing Internal Covatiate Shift”. In: *Computer Vision and Pattern Recognition* (2015).
[33] et al Christian Szegedy. “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning”. In: *Computer Vision and Pattern Recognition* (2016).
[34] François Chollet. “Xception: Deep Learning with Depthwise Seperable Convolutions”. In: *Computer Vision and Pattern Recognition* (2017).
[35] et al Dor Bank. “Autoencoders”. In: *Computer Vision and Pattern Recognition* (2021).
[36] et al Min Lin. “Network In Network”. In: *Neural and Evolutionary Computing* (2014).
[37] Yoshua Bengio Xavier Glorot. “Understanding the difficulty of training deep feedforward neural networks”. In: *nternational Conference on Artificial Intelligence and Statistics* (2010).
[38] et al Djork-Arné Clevert. “FAST AND ACCURATE DEEP NETWORK LEARNING BY EXPONENTIAL LINEAR UNITS (ELUs)”. In: (2016).
[39] Shaoqing Ren Kaiming He Xiangyu Zhang. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. In: *Computer Vision and Pattern Recognition* (2015).
[40] Siddharth Krishna Kumar. “On weight initialization in deep neural networks”. In: (2017).
[41] et al Kaiming He. “Deep Residual Learning for Image Recognition”. In: *Computer Vision and Pattern Recognition* (2015).
[42] Xue Ying. “An Overview of Overfitting and its Solutions”. In: *Journal of Physics: Conference Series* (2018).
[43] Leslie N Smith. “Cyclical Learning Rates for Training Neural Networks”. In: (2017).
[44] et al Kaichao You. “HOW DOES LEARNING RATE DECAY HELP MODERN NEURAL NETWORKS?” In: *Machine Learning* (2019).
[45] et al Jieun Park. “A Novel Learning Rate Schedule in Optimization for Neural Networks and It’s Convergence”. In: *Advance in Nonlinear Analysis and Optimization* (2020).
[46] Jimmy Lei Ba Diederik P. Kingma. “ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION”. In: *3rd International Conference for Learning Representations* (2017).

[47] Sebastian Ruder. “An overview of gradient decent optimization algorithms”. In: (2017).
[48] et al Amam Gupta. “Adam vs. SGD: Closing the generalization gap on image classification”. In: (2021).
[49] Mikhail Belkin Chaoyue Liu. “ACCELERATING SGD WITH MOMENTUM FOR OVERPARAMETERIZED LEARNING”. In: (2019).
[50] Yurii Nesterov. *Lectures on Convex Optimization*. Switzerland: Springer, 2018.
[51] *NVIDIA Deep Learning Performance Documentation*. 2023. URL: https://docs.nvidia.com/deeplearning/performance/index.html#optimizing-performance. Accessed: 24-5-2023.
[52] Szymon Migacz. *PERFORMANCE TUNING GUIDE*. 2023. URL: pytorch.org/tutorials/recipes/recipes/tuning_guide.html. Accessed: 24-5-2023.
[53] et al Ryosuke Okuta. “CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations”. In: (2017).
[54] et al Paulius Micikevicius. “MIXED PRECISION TRAINING”. In: *conference paper at ICLR 2018* (2018).
[55] et al Mingfeir Ma. *Accelerating PyTorch Vision Models with Channels Last on CPU*. 2023. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/pytorch-vision-models-with-channels-last-on-cpu.html. Accessed: 24-5-2023.
[56] Smitha Rao Saahil Afaq. “Significance Of Epochs On Training A Neural Network”. In: *INTERNATIONAL JOURNAL OF SCIENTIFIC TECHNOLOGY RESEARCH VOLUME 9* (2020).
[57] *Data Compression With Arithmetic Coding*. 2014. URL: https://marknelson.us/posts/2014/10/19/data-compression-with-arithmetic-coding.html. Accessed: 30-5-2023.
[58] Lex Fridman Eliezer Yudkiwsky. *Eliezer Yudkowsky: Dangers of AI and the End of Human Civilization ||LexFridmanPodcast#368*. 2023. URL: https://www.youtube.com/watch?v=AaTRHFaaPG8&t=5291s. Accessed: 24-5-2023.
[59] Donald E. Knuth. “Structured Programming with go to Statements”. In: *ACM Computing Surveys* (1974).
[60] et al George Toderic. “VARIABLE RATE IMAGE COMPRESSION WITH RECURRENT NEURAL NETWORKS”. In: *conference paper* (2016).
[61] et al Jingyu Zhao. “Do RNN and LSTM have Long Memory?” In: *ICML* (2020).
[62] et al Yannick Strumpler. “Implicit Neural Representations for Image Compression”. In: (2022).

A Pre-Project Report

EXPLORING NEURAL LOSSY IMAGE COMPRESSION

Hans Hennrik Hermansen s194042 & Rani Eydfinnsson í Bø s1940670

ABSTRACT

We explore the field of neural lossy image compression, and try out and compare different techniques. These techniques cover many areas of the learning process, like the loss function, activation function and model architecture. We also try a specialized training procedure where one inserts noise into the latent layer of an auto encoder in order to make the model robust to quantization of the latent layer, which then makes entropy coding of the latent layer possible.

We did also use this exploration to come up with our own loss functions, that applies different loss functions on lumen and chromatic components of images, and allows for weighing the importance of the lumen and chromatic components. We also present a novel training scheme and architecture of auto encoders, that allows variable rate image compression.

Index Terms— Lossy compression, perceptual loss functions, generalized divisive normalization, modular AE, quantization, entropy coding,

1. INTRODUCTION

Data compression is a central problem in computer science and engineering in general. When performing data compression, one encodes data in order to use less space. It is known that the entropy rate is a fundamental limit to the compression rate [1], and that one has to compromise the data in order to compress the data further, also known as lossy compression.

In the age of the internet with snapchat, instagram and other social media, a large amount of images are taken, sent and stored daily. This has lead to image specific data compression schemes like PNG and JPEG. Images are also increasing in quality and size, making better compression rates as important as ever, since it can help mitigate storage requirements and network load.

Many images are taken by humans for humans and thus the reconstructed images do not need to be perfect reconstructions of the original, but only good enough for some specific use case of humans. Especially when looking at images on a phone, the small screen makes small deviations from the original image acceptable if not unnoticeable. This makes it possible to surpass the fundamental limit of the compression rate, by allowing some compromise of the original image. One can state the lossy image compression problem as the minimization of:

$$R + \lambda D \tag{1}$$

where R is the bitrate, D is the distortion and λ is the tradeoff factor.

If we assume that the images are made for humans, the distortion D should ideally be measured with a measure that corresponds with the human visual system (HVS). This complicates the matter, since the HVS is complicated and not fully understood[2].

The field of neural lossy image compression is new compared to handcrafted methods like JPEG. However, neural lossy image compression was already better than JPEG in 2016 and in 2018 it was beating the best known handcrafted codecs[3]. With the field still being young, there are still a lot of open problems and ideas to explore and it is likely that further improvements in quality and compression will come. These are our reasons for wanting to explore neural lossy image compression.

2. DATA SET AND PROCESSING

For our purposes we needed a data set comprised of non-compressed images, but without the images being so large, that training the network would be too slow for us to perform experiments in reasonable time. For these reasons we chose the Kodak data set [4], a classic data set within the image processing sphere also used in [5] for benchmarking.

One problem with this data set is that it is quite small, which can make training ineffective. In order to enable better training we increased the size of our data set by performing several transformations on each image. We flipped the images and also switched all the color channels around. This gives $4 \cdot 8 = 24$ images per original image, giving us a data set with a total of $24^2 = 576$ images.

All of these transformations still leaves us with a valid data set, as we are not trying to capture anything about the Kodak data set in particular, but rather images as a whole. Therefore it can actually be to our advantage, as these transformed images might make the network more robust against images on the internet, like digital art, where the image portrays objects or color combinations not present in the physical world.

3. LOSS FUNCTION

The loss function is one of they key challenges and open problems of lossy image compression[3]. One can use standard loss functions like the mean squared error (MSE) for image compression, but the resulting compression will not align well to the HVS[6]. Because we want the images to be rated highly by humans, it is preferable to have a loss function that aligns well with the HVS[6].

There are some loss functions that are based on the HVS. Two notable examples are SSIM and MS_SSIM[7][8]:

$$
\begin{aligned}
l &= \tfrac{2\mu_x\mu_y+c_1}{\mu_x^2+\mu_y^2+c1} \\
c &= \tfrac{2\sigma_x\sigma_y+c_2}{\sigma_x^2+\sigma_y^2+c_2} \\
s &= \tfrac{\sigma_{xy}+c_3}{\sigma_x\sigma_y+c_3} \\
SSIM &= l^\alpha \cdot c^\beta \cdot s^\gamma \\
MS\_SSIM &= l_M^{\alpha_M} \cdot \textstyle\prod_{j=1}^{M} c_j^{\beta_j} s_j^{\gamma_j}
\end{aligned}
$$

These two loss functions are created under the assumption that the HVS is good at capturing structural similarity[7]. This is a somewhat one dimensional approach, ignoring other aspects of the HVS like color, but it results in visually superior images compared to more common loss functions like the MSE[7][8]. One weakness of SSIM and MS_SSIM, is that they were designed for gray scale images[7][8]. We found that by transforming the RGB picture into YCbCr like in JPEG conversion[9][10], we can exploit the strengths of SSIM or MS_SSIM on the lumen axis, while using other loss functions on the two chromatic axis. The transfomation is defined by:

$$
\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} K_R & K_G & K_B \\ -\frac{1}{2}\cdot\frac{K_R}{1-K_B} & -\frac{1}{2}\cdot\frac{K_G}{1-K_B} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2}\cdot\frac{K_G}{1-K_R} & -\frac{1}{2}\cdot\frac{K_B}{1-K_R} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \text{ [9]} \tag{2}
$$

Where R,G,B refers to the colors red, green and blue, and K_R, K_G and K_B are parameters that add up to 1 and practically scale the importance of the different colors (set to 0.299, 0.587 and 0.114 in [9])

It would be preferable to use a loss function that aligns to how the HVS sees color on the chromatic axis. We where not able to find any published loss function that is based on how the HVS sees color, and ended up using the mean absolute error (MAE) on the two chromatic axis, while using SSIM or MS_SSIM on the lumen axis.

With the YCbCr setup, it is possible to weight the axis in order to make the model prioritize color and lumen differently. We found that when using a simple auto encoder (AE) to recreate an image and using our loss functions with all axis weighted equally, the training fell into a local minimum where the structure of the images clearly looked like the original, but the color was off. By weighing color higher at about 95% and lumen at 5%, it was possible to avoid this local minimum resulting in the two loss functions:

$$
\begin{aligned}
&Our\_SSIM = \\
&\quad 0.05 \cdot SSIM(y) + 0.95 \cdot MAE(Cb, Cr)
\end{aligned} \tag{3}
$$

$$
\begin{aligned}
&Oor\_MS\_SSIM = \\
&\quad 0.05 \cdot MS\_SSIM(y) + 0.95 \cdot MAE(Cb, Cr)
\end{aligned} \tag{4}
$$

We compared these loss functions with SSIM and MS_SSIM, and two more common loss functions MSE and MAE. This is done by training a relatively simple auto encoder (AE) to reconstruct the original image. The comparison is shown below:

Fig. 1: original

Fig. 2: Our_MS_SSIM and Our_SSIM

Fig. 3: MS_SSIM and SSIM

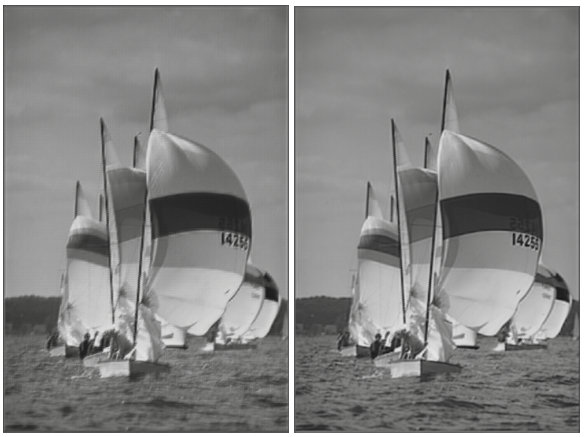


Fig. 4: MSE and MAE

It is clear that the models trained with our loss functions perform much better when it comes to color, and is only rivaled in clarity by MS_SSIM and SSIM. The model trained with SSIM seems to produce the clearest images, while our version of SSIM is the best overall. It is possible that some other training scheme or model might result in a different outcome, but the ability to split up the lumen and chromatic axis and apply different loss functions and weighting to these axis looks promising.

On a side note, we found both the SSIM and MS_SSIM to be somewhat unstable. Especially when using a larger learning rate on larger models, the loss functions would sometimes suddenly blow up totally ruining the training. The loss functions did also have problems converging when using larger models. Looking back to the formula for SSIM and MS_SSIM:

$$l = \frac{2\mu_x\mu_y+c_1}{\mu_x^2+\mu_y^2+c1}$$
$$c = \frac{2\sigma_x\sigma_y+c_2}{\sigma_x^2+\sigma_y^2+c_2}$$
$$s = \frac{\sigma_{xy}+c_3}{\sigma_x\sigma_y+c_3}$$

$$SSIM = l^\alpha \cdot c^\beta \cdot s^\gamma$$
$$MS\_SSIM = l_M^{\alpha_M} \cdot \prod_{j=1}^{M} c_j^{\beta_j} s_j^{\gamma_j}$$

It is likely that one of l, c or s comes close to being 0. For instance if the covariance σ_{xy} comes close to c_3, s would come close to 0.

It would be ideal if the loss functions were more well behaved, but it was possible to somewhat mitigate the problems by using a more well behaved loss function like the MAE or MSE in the first few epochs and reducing the learning rate. Such measures were necessary to get the results shown later in this report.

Furthermore [11] shows that there are many other problems with the SSIM loss function, and all derivations of it. One problem is that SSIM is less sensitive to errors between two white images, compared to two black images, another is that resolution has a big effect on SSIM. These problems combined with the problems we faced and there being questions of the validity of the claim that SSIM aligns with the HVS [11] suggests that further work on the development of loss functions that align well with the HVS could be beneficial.

4. ACTIVATION FUNCTION

When designing a network, the choice of activation function is an important one. However it is often difficult to know the best option for the task ahead of time, so experimentation is often needed. In the case of neural compression, articles like [12] clearly present the generalized divisive normalization function as the superior option.

The generalized divisive normalization (GDN) [13]:

$$y_i = \frac{z_i}{(\beta_i+\sum_j \gamma_{ij}|z_j|^{a_{ij}})^{\epsilon_i}}$$
$$\mathbf{z} = \mathbf{Hx}$$

is an activation function with trainable parameters. This way certain features can have a larger or smaller impact when normalizing a single feature, depending on what the network discovers to work best. It is a generalization of divisive normalization, which is a model originally proposed to describe the firing rate of neurons in the visual cortex [14]. As such it also makes intuitive sense to try this activation function when doing image compression, which greatly relies on our visual perception of an image. GDN Gaussianizes the data more effectively than more common activation functions[13][5]. It also achieves smaller mutual information between components than alternative Gaussianization methods[13]. When using a grid search to compare GDN with the common activation functions sigmoid, tanh, ReLU, leaky ReLU and Elu on a simple AE, GDN and Elu performed the best.

As such we used ELU for most of our experimentation with our own model as it is simpler to implement and less computationally intensive. We did however use GDN for the Quantization based model described in section 5, and will continue including GDN in our further testing.

5. QUANTIZATION BASED MODEL

Most articles named in the benchmark [12] and the architectures presented in [12] and [5] use quantization as a method of increasing the compression rate. Quantization is simply a way of taking a continuous signal and transforming it into a discrete one [15]. This is then applied to the latent layer in order to apply further compression with lossless methods like arithmetic encoding [16]. This is a method we want to try and apply to our model in the future, so to gain better understanding of quantization and the challenges of training such a model, we trained the model presented in [5] on our data set, as that model is quite simple and still showed quite good results.

In this model the quantization scheme is quite simple. Simply round to the nearest whole number. However issues arise when training a model with this quantization. If rounding is performed during training most gradients simply zero out and no process can be made by gradient descent. When rounding, the largest difference between the original value and the rounded value that can occur is ± 0.5. In order to make the model robust towards the quantization step, only one key insight is needed. If the model is robust towards random noise in the range $[-0.5; 0.5]$ added to the latent space, it will also be robust towards rounding. As such we can replace the quantization step during training by adding random noise to the latent representation.

A thing to notice about this model is that since the quantization step compresses the latent space, one can allow for a larger latent space, and as such greatly increase the amount of channels in the latent space. As our own model was rather difficult to train, we did not manage to attempt quantization on it, but it is a goal for the future. The results from this model are shown in figure 5.

Fig. 5: original and compressed 92%

These results are not quite as good as in [5]. This might be due to superior training, or the fact that they used a larger data set.

6. MODULAR AE

The quantization and entropy coding method can clearly work well[5], but it did not seem intuitive to let most of the compression happen outside of the neural network and to insert noise into the latent layer to mitigate the effects of quantization. Furthermore the compression rate depends on the range of values in the encoded image, since this range decides the amount of values left after quantization and therefor the compression rate after entropy coding. Furthermore, experience has shown that it is often preferable to give the neural network flexibility and let it find its own way of solving the task, rather then forcing it to solve the task in a specific way. This lead us to try to use a normal AE with a small bottleneck layer for the entire encoding and decoding procedure.

Using an AE was somewhat successful, but as stated in[17], the main things that make AE unfit as a drop in replacement for standard image codecs, are that:

- AE does not result in variable-rate encoding
- It is hard to ensure visual quality
- AE does not work well on variable image sizes

We found that by structuring an AE such that it can skip over layers and still produce a good output image, one can achieve variable rate compression with one AE. A sketch of the architecture is shown in figure bellow:

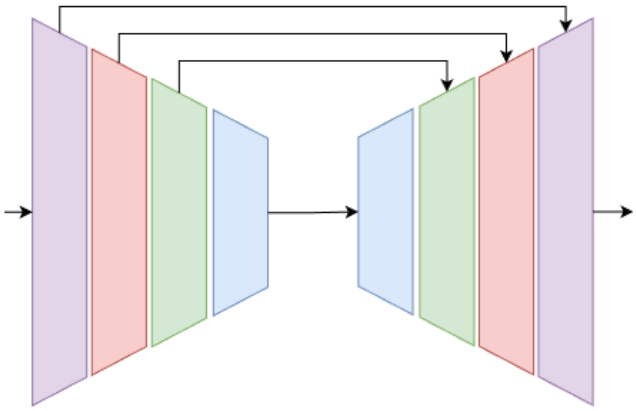


Fig. 6: Modular AE architecture

This structures allows variable compression, since the model can skip layers. For instance the lowest compression is where the image is encoded and decoded only by the purple sections, while the highest compression is when the image is encoded by going trough the entire left half of the AE, and is decoded by the entire right half of the AE.

This method worked, but it was very hard to train. During training, we had to implement a temporary AE for each compression rate, in order to make sure that the modular AE could skip layers. These temporary AE's can be seen in the image above, with the lowest compression being the purple part of the image, the second lowest compression being the purple and red parts, the third lowest being the purple, red and green parts etc.

The first training scheme we tried, started by training the temporary AE with the lowest compression rate. Afterwards the temporary AE with the second lowest compression gets its purple regions initialized to the learned weights of the AE with the lowest compression. These weights were also frozen in order to make sure that the final modular AE can skip from the purple encoder to the purple decoder and still produce a nice image. Then the these learned weights were used to initialize the AE with a higher compression factor and so on until the full modular AE was trained.

This approach worked somewhat, but the model had sudden drops in quality with just a little more compression. It was as if the information could not reach the deepest layers of the encoder half. In order to mitigate this, we tried to first train the full AE, then to initialize all the encoders of the temporary AE's with the weights learned, and freeze the encoders. This makes sure that information can come to the deepest layers of the encoder, while still leaving flexibility in the decoders. Then the model was trained with the same scheme as described before, but with the encoder being frozen.

This idea resulted in our best results and can be seen below:

Fig. 7: original (24 bits/pixel)

Fig. 8: compressed (16 bits/pixel) and (8 bits/pixel)

Fig. 9: compressed (4bits/pixel) and (2 bits/pixel)

But it was not without problems and modifications. Firstly the model had problems converging and the loss did sometimes blow up when using our modified version of the SSIM loss function. This was somewhat mitigated, mostly by pretraining the model with the MSE loss function for a few epochs and reducing the learning rate. The freezing of layers did also seem to disrupt the final image, since we got better reconstruction when training a normal AE with similar structure. It seemed like the segmentation of the model made it more difficult for the model to train, and that the freezing of the encoding half of all AE's made the model less flexible.

Our results with this model are clearly inferior compared to the quantization and entropy coding model shown in[5]. We had a lot of problems during training and it felt like we were not able to get the best performance out of the architecture. However, we still got some alright results and it seems likely that better training and further optimization has potential to get this method working on par with the quantization and entropy coding method. This would make some intuitive sense, since there is no quantization of the latent layer, giving the model freedom to chose how the compression is performed. Unlike many other common methods, modular AE does not partition the images into sections, which should result in fewer blocking effects[12][17].

Thus the modular AE solves one of the problems with AE's as an drop in replacement for lossy image compression, namely variable rate compression, with the caveat that the possible compression rates are discrete. One could somewhat ensure visual quality, by checking the loss at each compression rate, and stopping before the loss gets to big. Even if this could remedy some of the problems AE have, one would likely need to train multiple AE's in order to effectively compress images of different sizes, since too much padding would likely lead to inefficient compression.

7. CONCLUSION

We managed to get an overview of the field of neural lossy image compression and to understand specialized ideas and terms that are not widely used in other areas of machine learning. We were able to test and compare cutting edge ideas, and come up with our own ideas, inspired by our exploration.

We found that the loss functions we came up with worked well, further testing might show it to be definitively superior to other methods like the normal SSIM. We suspect that further work based on the HVS, especially how the HVS perceives color, could result in a loss function that aligns closer to human perception.

We also tested our idea of a novel architecture and training scheme, which we call modular AE. We were not able to get it to perform well compared to the model proposed in[5], but it seems promising and further work could lead to better results.

The field of neural lossy image compression is still young and methods will likely come with many changes and improvements in the coming years.

We did originally want to test the methods we found on larger images. Being interested in photography, we had access to a collection of 100GB of large images, which could easily make a training set of 100.000+ images with a few transformations. However, due to time constraints and the unexpected amount of ideas to test, this was not prioritized.

Link to GitHub :
https://github.com/raniiboe/DeepLearningProject

8. REFERENCES

[1] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, 1948.

[2] Richard H. Masland1 and Paul R. Martin, "The unsolved mystery of vision," *Current Biology : CB*, 2007.

[3] Yibo Yang et al, "An introduction to neural datacompression," .

[4] "Kodak data set," https://www.kaggle.com/datasets/sherylmehta/kodak-dataset?select=kodim04.png, note = Accessed: 2-1-2022.

[5] Johannes Ballé et al, "End-to-end optimized image compression," 2017.

[6] B. Girod, "What's wrong with mean-squared error?," *M.I.T. Press*, 2016.

[7] Zhou Wang et al, "Image quality assessment: From error visibility to structural similarity," *IEEE*, 2004.

[8] Zhou Wang et al, "Multi-scale structural similarity for image quality assessment," *IEEE*, 2003.

[9] Eric Hamilton, "Jpeg file interchange format version 1.02," *IEEE*, 2003.

[10] Gregory K. Wallace, "The jpeg still picture compression standard," *IEEE*, 1991.

[11] Tomas Akenine-Möller Jim Nilsson, "Understanding ssim," *arXiv:2006.13846*, 2020.

[12] Yueyu Hu, Wenhan Yang, Zhan Ma, and Jiaying Liu, "Learning end-to-end lossy image compression: A benchmark," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[13] Johannes Ballé et al, "Density modeling of images using a generalized normalization transformation," 2016.

[14] "Normalization," https://en.wikipedia.org/wiki/Normalization_model, Accessed: 2-1-2022.

[15] "Quantization," https://en.wikipedia.org/wiki/Quantization_(signal_processing), Accessed: 2-1-2022.

[16] "Arithmetic encoding," https://en.wikipedia.org/wiki/Arithmetic_coding, Accessed: 2-1-2022.

[17] George Toderic et al, "Variable rate image compression with recurrent neural networks," *conference paper*, 2016.

B Pre-Project Poster

Lossy Neural Image Compression

s194042 Hans Henrik Hermansen, s194067 Rani Eydfinnsson í Bø

1 Introduction

Because of the growing amount of images being sent and stored, image compression is more needed than ever. Neural networks based methods have been shown to lead to superior bit-rates compared to other methods[9]. This lead to us being interested in investigating lossy neural image compression. This is essentially an optimization problem where the objective function to minimize is:

$$R + \lambda D$$

where R is the bitrate, D is the distortion and λ is the tradeoff factor.
We explore the following techniques for lossy neural compression:

- Basic Auto Encoder
- Our own take on a modular Auto Encoder
- Auto Encoder with Quantized latent space

During our experiments we compare GDN with more common activation functions. We also make our own loss functions and compare it to SSIM, MS-SSIM, MSE and MAE.

2 Exploration and experiments

Loss function

The loss function is one of they key challenges of lossy image compression. One can use the mean squared error (MSE) for image compression, but the resulting compression will not align well to the human visual system (HVS)[7]. There are loss functions that are based on how the HVS works. Two such loss functions are SSIM and MS-SSIM[5][6]:

$$l = \frac{2\mu_x\mu_y + c_1}{\mu_x^2 + \mu_y^2 + c1}$$
$$c = \frac{2\sigma_x\sigma_y + c_2}{\sigma_x^2 + \sigma_y^2 + c_2}$$
$$s = \frac{\sigma_{xy} + c_3}{\sigma_x\sigma_y + c_3}$$
$$SSIM = l^{\alpha} \cdot c^{\beta} . ^{\gamma} s$$
$$MS - SSIM = l_M^{\alpha_M} \cdot \prod_{j=1}^{M} c_j^{\beta_j} s_j^{\gamma_j}$$

One weakness of SSIM and MS-SSIM, is that they were designed for gray scale images[5][6]. We found that by transforming the RGB picture into YCbCr like in JPEG conversion[8][10], we can attempt to exploit the strengths of SSIM or MS-SSIM on the lumen axis, while using other loss functions on the two color axis, preferably one that aligns to the HVS. Because we could not find any published loss function that is based on how the HVS sees color, we applied the mean absolute error (MAE) on the two color axis. With this setup, we found that the optimal color to lumen trade of was around 95% color and 5% lumen. We compared our two loss functions with four others on a simple auto encoder (AE) model with relatively few epochs. The comparison is shown in the QR code below:

We found that SSIM and MS-SSIM were not as stable as other loss-functions during training. We did discover some methods for dealing with this, but further testing might change our results for the loss functions.

Activation function

The generalized divisive normalization (GDN):

$$y_i = \frac{z_i}{(\beta_i + \sum_j \gamma_{ij} |z_j|^{a_{ij}})^{\epsilon_i}}$$
$$\mathbf{z} = \mathbf{Hx}$$

activation function Gaussianizes the data more efficiently than more common activation functions[2][3] it also achieves smaller mutual information between components than alternative Gaussianization methods[2]. When using a grid search to compare GDN with the common activation functions sigmoid, tanh, ReLU, leaky ReLU and Elu on a simple AE, GDN and Elu performed the best.
We further compared GDN and Elu on a model based on the model from[3], which is described further in the next section. We found that Elu and GDN where comparable, but that GDN had fewer artifacts, and was nicer overall. We also tried with different loss functions, and GDN was still superior. The comparison is shown in the QR code below:

Quantization based model

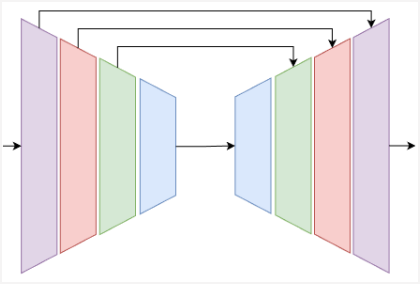
One of the most comon aproches to lossy neural image compression, is done by performing entropy coding on the quantized latent layer of an AE[9][4]. This is done because one can greatly decrease the size of the quantized representation with arithmetic encoding. During training quantization can not be utilized as it leads to 0 or undefined gradients, which causes gradient decent to not work. One way to fix the gradient problem, is to not perform quantization during training, but rather add uniform noise which changes the latent space in a similar manner. This makes the model robust to later quantization when the model is in use[3]. We base our model on the model from[3], which follows this principle, but we did not manage to get the same quality. It is possible that our training scheme is inferior, but we are not sure. We tried building different models based on Elu and GDN activation, and MSE, SSIM and our loss based on SSIM, the results can be seen in the same QR code above.
We found the one with our loss function combined with the GDN activation function the most appealing.

Modular auto encoder

According to[1], the main things that make AE unfit as a drop in replacement for standard image codecs, are that:

- No variable-rate encoding
- Hard to ensure visual quality
- One AE needed for each image size

We found that by training an AE such that it can skip over layers and still produce a good output image, one can achieve variable rate compression with one AE. This was done by training different sections of the AE while freezing other sections. A sketch of the architecture is shown in figure bellow:



This method worked, but it was very hard to train. We managed to train a model with four layers, but most of our tries ended with sudden steep declines in quality. Our best try can be seen in the pictures in the QR code below:

This method seams inferior compared to the quantization and entropy coding method shown in[3]. It is possible that better training and further optimization could get this method working on par with or better than the quantization and entropy coding method. This would make some intuitive sense, since there is no quantization of the latent layer, giving the model total freedom of how compression is performed. Unlike many other common methods, modular AR does not partition the images into sections, which also reduces the odds of blocking effects[9][1].
One could somewhat ensure visual quality, by checking the loss of each step, and stopping before the loss got to big. Even if this could remedy some of the problems AE have, one would likely need to train multiple AE in order to effectively compress images of different sizes, since too much padding would likely lead to inefficient compression.

3 Conclusion and further work

We managed to get a overview of the field of lossy neaural image compression, and to understand specialized ideas and terms that are not widely used in other areas of machine learning. We were able to test and compare cutting edge ideas, and come up with our own ideas, inspired by the work we read. We found that the loss function we came up with worked well, and we suspect that further work based on how the HVS perceives color could align the loss function closer to human perception. We could not get the modular AE to work well compared to the model proposed in[3], but it seems promising and further work could lead to better results.
Because of the ever increasing quality of images, we originally wanted to test our findings on bigger images. Given more time we would have explored the compression efficiently of larger imagesfurther.

References

[1] George Toderic et al. "VARIABLE RATE IMAGE COMPRESSION WITH RECURRENT NEURAL NETWORKS". In: *conference paper* (2016).
[2] Johannes Ballé et al. "DENSITY MODELING OF IMAGES USING A GENERALIZED NORMALIZATION TRANSFORMATION". In: (2016).
[3] Johannes Ballé et al. "END-TO-END OPTIMIZED IMAGE COMPRESSION". In: (2017).
[4] Yibo Yang et al. "An Introduction to Neural DataCompression". In: *Preprint* (2022).
[5] Zhou Wang et al. "Image Quality Assessment: From Error Visibility to Structural Similarity". In: *IEEE* (2004).
[6] Zhou Wang et al. "MULTI-SCALE STRUCTURAL SIMILARITY FOR IMAGE QUALITY ASSESSMENT". In: *IEEE* (2003).
[7] B. Girod. "What's Wrong with Mean-squared Error?" In: *M.I.T. Press* (2016).
[8] Eric Hamilton. "JPEG File Interchange Format version 1.02". In: *IEEE* (2003).
[9] Yueyu Hu et al. "Learning end-to-end lossy image compression: A benchmark". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021).
[10] Gregory K. Wallace. "The JPEG Still Picture Compression Standard". In: *IEEE* (1991).

Technical
University of
Denmark

2800 Kgs. Lyngby
Tlf. 4525 1700