UNIVERSIDADE FEDERAL DO PAMPA

KELVIN CLOVIS MONTOLI DE SOUZA

NEURAL NETWORKS TO ESTIMATE THE PROBABILITY OF RESIDUAL SYNTAX ELEMENTS FOR THE AV1 CODEC

Bagé
February 2023

KELVIN CLOVIS MONTOLI DE SOUZA

NEURAL NETWORKS TO ESTIMATE THE PROBABILITY OF RESIDUAL SYNTAX ELEMENTS FOR THE AV1 CODEC

Graduation Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering

Bagé
February 2023

Ficha catalográfica elaborada automaticamente com os dados fornecidos pelo(a) autor(a) através do Módulo de Biblioteca do Sistema GURI (Gestão Unificada de Recursos Institucionais).

K29N Souza, Kelvin Clovis Montoli de

Neural Networks to estimate the probability of residual syntax elements for the av1 codec / Kelvin Clovis Montoli de Souza.

98 f.: il.

Advisor: Fábio Luís Livi Ramos

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal do Pampa, Engenharia de Computação, 2023.

1. AV1. 2. Video codification. 3. Neural Networks. 4. Syntax Elements. I. Título.

SERVIÇO PÚBLICO FEDERAL

MINISTÉRIO DA EDUCAÇÃO

Universidade Federal do Pampa

KELVIN CLOVIS MONTOLI DE SOUZA

NEURAL NETWORKS TO ESTIMATE THE PROBABILITY OF RESIDUAL SYNTAX ELEMENTS FOR THE AV1 CODEC

Graduation Work presented in partial fulfillmentof the requirements for the degree of Bachelor in Computer Engineering.

Graduation Work defended and approved in: 01 of February of 2023.

Judging committee:

____

Prof. Dr. Fábio Luís Livi Ramos
Advisor
UNIPAMPA

____

Prof. Dr. Bruno Silveira Neves

UNIPAMPA

____

Prof. Dr. Gerson Alberto Leiria Nunes

UNIPAMPA

Assinado eletronicamente por **GERSON ALBERTO LEIRIA NUNES**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 07/02/2023, às 07:26, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **FABIO LUIS LIVI RAMOS**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 07/02/2023, às 16:45, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

Assinado eletronicamente por **BRUNO SILVEIRA NEVES**, **PROFESSOR DO MAGISTERIO SUPERIOR**, em 07/02/2023, às 21:57, conforme horário oficial de Brasília, de acordo com as normativas legais aplicáveis.

A autenticidade deste documento pode ser conferida no site https://sei.unipampa.edu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **1049193** e o código CRC **E9A845DA**.

Referência: Processo nº 23100.002573/2023-40 SEI nº 1049193

ACKNOWLEDGEMENTS

First and foremost, I would like to thank God for blessing me with all the experiences and people that I've encountered throughout my graduation.

Although it may sound kinda stupid and cliché, I would like to sincerely thank every single person that helped me during the graduation by citing their names below at some point of this section. Starting with my family, I would like to thank my mom Lisiane and my dad Clovis for the continuous support that they both gave me, asking every me single day if I was ok or if I needed any kind of help, besides that, they have been always supporting me financially, so I'm really thankful for that as well. Staying on the topic of "Family", I would also like to thank my aunts, Maria and Miriam, they both have been wonderful for me throughout the graduation, always driving me back to Bagé when required and as my parents were doing, always asking if I was ok or needed any kind of help. In general, I couldn't be more thankful for my family that I am now, my parents, sister, brothers, grandparents, aunts, and uncles, THANK YOU, each one of them had an immense impact on my graduation and this graduation work and I couldn't be more thankful for having them.

Changing topics a bit and talking about my friends, I would like to thank Artur, Ignacio (the man destined to save humanity from cancer), Thiago (big thigas), Michel (michelzindacomp), Eric (CR7), Guilherme, Maria Elizabeth, Matheus, Thamires, Giuliana, Rafael(robótica), Tomás, Gleyson, Indra, Rafael(hardware), Fábio, Bruno, Leonardo, Gerson, Leandro, Luan, Luigi, José, Túlio, Lucas (emo), Adriele, Mateus, Estefanie, Vitor, Matheus (ghost), Marco, Gabriele, Fernando, Igor, Oesley, Douglas, DiFranco, Samuel and Thales for the laughs and all the moments that we've had together, I know that I've said a lot of names, but I really appreciate every single one of them for sticking with me for so long, although I may not speak with every single one of them on a regular basis, all of them had a positive impact on my graduation at some point, and for that I would like to say: THANK YOU.

Changing subjects again, I would like to thank all the teachers from the course, but specially the following ones: Fábio, Bruno, Leonardo, Gerson, Sandro, Betemps, Wladimir, Júlio, Danrlei, Sandra, Peterson, Luciano, Everson, Leandro, Milton, Gabriel, Denise, Gustavo and Caroline, for helping me at some point of my graduation. I know that I can be terribly slow and dumb at sometimes, but I really appreciate the patience that everyone had. In general, thanks for always listening to whatever I had to say and not giving up on me, this really meant a lot to me and helped me to stay on the course.

Next topic that I wanted to cover was the University, I don't think that I have a lot to say about the university, but thanks for providing me food every business day for free, for giving me a place to learn what I was always wanted to, for helping me financially through the entire course and providing me a way to become a computing engineering student, thank you for all the people that are part of UNIPAMPA and supported me during my graduation.

Last but not least, I would like to thank Jiovana, Fábio, Tulio, Kelly, Denise and CNPq for helping me with the development of this graduation work.

As a final note, I know that this section is meant to be an acknowledgment for the graduation work and not necessarily the graduation itself, but I thought that would be nice to transform this section to a kind of "memorial" for my entire graduation. As previously mentioned, I wrote a lot of names, because I think that writing every name is a nice way of remembering each one of them as I get old. This may sound a bit dramatic, but, for the last time in this section and from the bottom of my heart, I sincerely say: THANKS to each one of these names, thanks for helping me, thanks for making me laugh, thanks for cheering me up when I was feeling sad, thanks for teaching me everything that I should learn, thanks for inspiring me, thanks for existing and being a massive positive impact at this important stage of my life, it really meant a lot to me =).

“At the end of the day, all the effort that you put is just for having a chance”

— Chica Umino

ABSTRACT

This work aims to showcase an Artificial Neural Network model to estimate the probability of residual syntax elements for the AV1 standard. Since the beginning of the 21st century, the usage of platforms that utilize video resources has been increasing. With the arrival of YouTube in 2005, it became possible to notice a growing usage of videos for entertainment and other mundane tasks. Over time, with the increase of social media and streaming applications, video traffic has become a large portion of global data traffic. In this context, it is recognized that there is a need for a video coding solution that can achieve a high level of compression. Codecs are a set of tools that are designed to significantly reduce the size of a video sequence. Although codecs that achieve a high compression rate are already available, many of them are locked down through royalty fees, which limits their usage. In 2018, the Alliance for Open Media (AOM) consortium created the codec AOMedia Video 1 (AV1) with the goal of providing a high-compression, royalty-free, open-source video format. As AV1 is a relatively new codec, finding new ways to improve its compression rate is a desired goal. In the AV1 context, residual syntax elements make up most of the entropy coded syntax elements, with entropy encoding being the last stage in the codec workflow. Therefore, accurately estimating their occurrence probabilities is crucial for obtaining a highly compressed bitstream (i.e., entropy encoding relies on these probabilities to compress the input data). To achieve this requisite, a total of 26 Artificial Neural Network models were evaluated, resulting in a final model that was capable of estimating the CDF of the DC_sign Syntax element, with an error of 0.11% when compared to the actual CDF.

Keywords: AV1. Video codification. Neural Networks. Syntax Elements.

RESUMO

Este trabalho tem como objetivo demonstrar os ganhos e perdas de compressão que podem ser obtidos usando uma Rede Neural para estimar a probabilidade de elementos residuais de sintaxe para o padrão AV1. Desde o início do século XXI, o uso de plataformas que realizam transmissões de vídeo e que se utilizam de recursos visuais tem aumentado. Com a chegada do YouTube em 2005 foi possível notar uma crescente utilização de vídeos em âmbitos mais mundanos, como por exemplo entretenimento. Avançando alguns anos, com o aumento de aplicações relacionadas a redes sociais, o tráfego de vídeo compõe uma grande parcela do tráfego global de dados. Com este cenário em mente, se vê a necessidade de codecs que atinjam um alto grau de compactação de vídeo, codecs são ferramentais que possibilitam a codificação de vídeo, reduzindo consideravelmente o tamanho de uma sequência de vídeo qualquer. Ainda que já existam codecs que atinjam um alto grau de compactação de vídeo, eles são normalmente protegidos por patentes. Em 2018, pelo consórcio Alliance for Open Media (AOM), o codec AOMEDIA Video 1 (AV1) foi lançado, tendo por objetivo fornecer um codec que atingisse um alto grau de compactação de vídeo que fosse livre de royalties e open source. Com isto, devido a recência deste codec, formas que amplifiquem o grau de compactação são desejadas. No escopo do AV1, os elementos sintáticos residuais compõem a maioria dos elementos sintáticos que são codificados durante a entropia, sendo a codificação de entropia o último estágio no fluxo do codec. Portanto, uma estimativa adequada das probabilidades dos elementos sintáticos residuais é vital para obter um fluxo de bits o mais compactado possível, visto que a codificação de entropia depende das probabilidades estarem o mais precisas possíveis para que possa gerar o menor bitstream possível. Para atingir este requisito, um total de 26 modelos de redes neurais artificiais foram avaliadas, resultando em um modelo final que foi capaz de estimar o CDF do elemento sintático DC_sign, com uma margem de erro de 0.11%, quando comparado ao valor original do CDF.

Palavras-chave: AV1, Codificação de vídeo, Redes Neurais, Elementos sintáticos.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ADAM	Adaptative Moment Estimation
ADST	Asymmetric Discrete Sine Transform
AI	Artificial Intelligence
ANN	Artificial Neural Network
AOMedia	Alliance for Open Media
ASIC	Application-Specific Integrated Circuit
AV1	AOMedia Video 1
BR	Base-range
CABAC	Context-Adaptive Binary Arithmetic Coding
CC	Computational Cost
CDF	Cumulative Distribution Function
CDEF	Constrained Directional Enhancement Filter
CIF	Common Intermediate Format
CNN	Convolutional Neural Network
codec	COder DECoder
CPU	Central Processing Unit
CQ	Constant Quality
CRFPM	Compressed Reference Frame Prediction Mode
DC	Discrete Cosine
EOB	End of block
FFNN	Feed Forward Neural Network
FHD	Full High Definition
FPGA	Field-Programmable Gate Array
FPS	Frames per second

FTRL	Follow The Regularized Leader
GPU	Graphic processing unit
HEVC	High-Efficiency Video Coding
HR	High-range
IDTX	Identity Transform
LR	Low-range
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
MC	Motion Compensation
ME	Motion Estimation
MPM	Most Probable Mode
MSE	Mean Squared Error
MSLE	Mean Squared Logarithmic Error
MV	Motion Vector
NADAM	Nesterov-accelerated Adaptive Moment Estimation
NL	Number of Layers
NN	Neural Network
NNL	Number of Neurons per Layer
NM	Number of multiplications
QP	Quantization Parameter
ReLU	Rectified Linear Activation Function
RMSpop	Root Mean Squared Propagation
RNN	Recurrent Neural Network
RSE	Residual Syntax Element(s)
SB	SuperBlock
SE	Syntax Element(s)

SGD	Stochastic Gradient Descent
SVT-AV1	Scalable Video Technology AOMedia Video 1
TU	Transform Unit
UNIPAMPA	Universidade Federal do Pampa
YCbCr	Luminance (Y), Chrominance Blue (Cb), Chrominance Red (Cr)
VVC	Versatile Video Coding
WM	Weight for Multiplications
WS	Weight for sums or subtractions

CONTENTS

1 INTRODUCTION

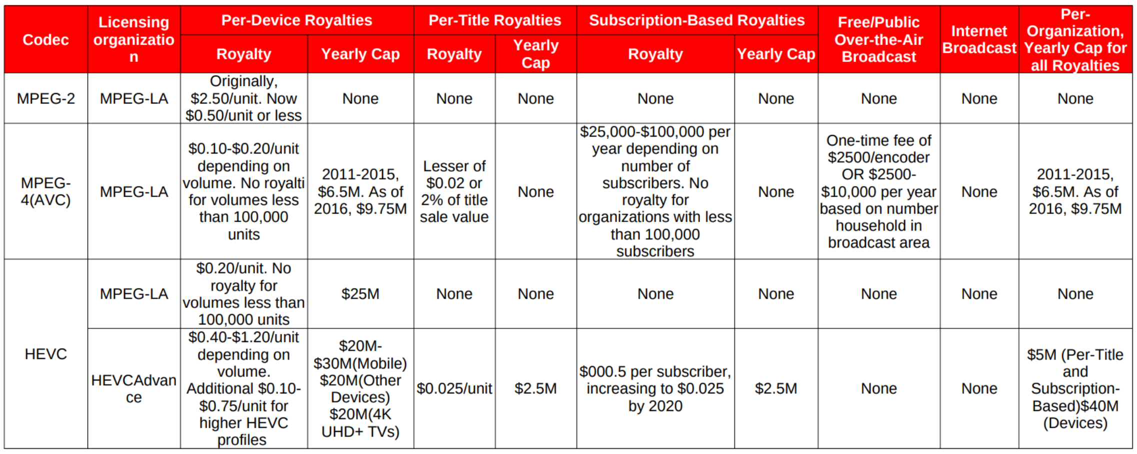
Since the beginning of the XXI century, the amount of streaming applications have been increasing. If we took YouTube for example, the streaming provider was launched in 2005, and after six years it had 0.8 billion active users (STATISTA, 2022). Eleven years later, YouTube is the most popular streaming platform in the world, averaging over 2.8 billion daily active users (STATISTA, 2022). If we considered only the most viewed video on YouTube (i.e., Baby Shark Dance), which has an average of 200,000 views per hour and a duration of 2:18 minutes, we would have a raw video file of 11,7Gb at a FHD resolution (1920×1080), and, therefore, would require a bandwidth of approximately 160,04 Mbits/s per user (according to table 2 proportion), which is an absurd amount for a single 2-minute video.

Further on, if we consider that YouTube has an average duration of 11:40 per video and approximately 5 billion daily views (STATISTA, 2018), it becomes possible to notice the huge amount of bandwidth that would be needed to transmit these videos. Fortunately, we have the so-called video **codecs** (short for *COder DECoder*) or standards that are the set of required tool to compress, for instance, that 11,7Gb file to something much smaller and feasible for our current network.

Although codecs that can achieve a high compression rate already exist, for example HEVC (ITU-T, 2013) and VVC (BROSS et al., 2020), the biggest issue that they present is that most of them have limited usage due to their expensive and burdensome royalties. If we took the royalties of HEVC as an example, it could get pretty expensive, as we can see on Table 1. Besides that, if we would consider streaming platforms that profit with monthly subscriptions, the price of the subscription would be directly affected by the price of the royalties payment, making it less appealing to the consumer.

Taking the royalties into account, plus the need of a video coding standard/format that could achieve a high compression rate, resulted in the foundation of the Alliance for Open Media (**AOM**) group in 2015, a consortium composed of multiple technology organizations, among these: *Google*, *Amazon*, $Netflix$, and $Microsoft$. One of the main objectives of the AOM was to develop a codec that would be open-source, free of royalties, and capable of achieving a high compression rate. Three years after its foundation, in 2018, the AOMedia Video 1 (**AV1**) came to life, being able to achieve up to 30% higher compression rate than HEVC and VP9, while maintaining the same quality, at the cost of having a decoding time 17% slower (LIU, 2021). To illustrate AV1s efficiency and usage, since 2021 Netflix's have been using AV1 on some Smart TV's, it is said that in some Smart TV's drops in video quality were reduced by as much as 38% (OZER, 2021).

Table 1 – Royalty prices



Codec	Licensing organizatio n	Per-Device Royalties		Per-Title Royalties		Subscription-Based Royalties		Free/Public Over-the-Air Broadcast	Internet Broadcast	Per-Organization, Yearly Cap for all Royalties
		Royalty	Yearly Cap	Royalty	Yearly Cap	Royalty	Yearly Cap			
MPEG-2	MPEG-LA	Originally, $2.50/unit. Now $0.50/unit or less	None	None	None	None	None	None	None	None
MPEG-4(AVC)	MPEG-LA	$0.10-$0.20/unit depending on volume. No royalti for volumes less than 100,000 units	2011-2015, $6.5M. As of 2016, $9.75M	Lesser of $0.02 or 2% of title sale value	None	$25,000-$100,000 per year depending on number of subscribers. No royalty for organizations with less than 100,000 subscribers	None	One-time fee of $2500/encoder OR $2500-$10,000 per year based on number household in broadcast area	None	2011-2015, $6.5M. As of 2016, $9.75M
HEVC	MPEG-LA	$0.20/unit. No royalty for volumes less than 100,000 units	$25M	None	None	None	None	None	None	None
	HEVCAdvan ce	$0.40-$1.20/unit depending on volume. Additional $0.10-$0.75/unit for higher HEVC profiles	$20M-$30M(Mobile) $20M(Other Devices) $20M(4K UHD+ TVs)	$0.025/unit	$2.5M	$000.5 per subscriber, increasing to $0.025 by 2020	$2.5M	None	None	$5M (Per-Title and Subscription-Based)$40M (Devices)

Source: Rutz (2016).

Considering that AOM is a group of multiple big tech companies that reunited a lot of professionals of the area to work together, AV1 development and implementation considered three base codecs, being then: Mozilla's Daala, Cisco's Thor, And Google's VP10. AV1 had its bitstream specification released in 2018, as already mentioned, along with a reference implementation written in C language. Because of its open source nature, since its launch, a lot of third-party versions of AV1 have come to life. For example, **SVT-AV1**[1]and **rav1e**[2]. Although these versions may not change the workflow of AV1, they are made to fulfill a more specific niche. One may take SVT-AV1 as an example, which initially was developed by Intel in partnership with Netflix, where this variation of AV1 meant to aid Intel's CPU's and, in 2020, was adopted by the AOM as one of their current projects.

Considering AV1s kinship earlier mentioned, its workflow had to be similar to its relatives (e.g., Daala, VP9 and Thor), having its heritage from codecs that followed the so-called hybrid process (e.g., HEVC). Regarding the hybrid process, it is a workflow that is followed by most of the current codecs, and evolves around four steps, being:

1. The Prediction process, which looks to remove redundancies on similar pixels. This step is achievable by looking at a specific frame, then looking for similarities at multiple pixels block. After that, a residual block is generated by subtracting the

[1]https://github.com/AOMediaCodec/SVT-AV1
[2]https://github.com/xiph/rav1e

current frame with the next one.

2. The Transform phase that aims to reduce the correlation from the residual block by reorganizing it in another fashion.
3. The Quantization step, which aims to reduce and decrease the residual coefficients values generated by the Transform step. Quantization can be a lossy process, depending on the selected quantization level.
4. The Entropy Coding step, whose objective is to compress the tranformed coefficients using their occurrence probability for that purpose.

Although most recent codecs follow this pattern, they usually differ from one another by making some big tweaks in these stages. If we took HEVC as an example, he uses the Context-Adaptive Binary Arithmetic Coding (CABAC) (MARPE; SCHWARZ; WIEGAND, 2003) as their entropy coding algorithm, whereas AV1 uses a Multi-Symbols Alphabet (M-ary) arithmetic encoding, being the main difference between that CABAC, as the name indicate, work only with binary symbols, whereas M-ary works with a Multi-symbol approach, as stated by Han et al. (2021).

Moreover, with the rising of **Artificial Intelligence** (AI) related works (BARROS, 2021), along with the necessity for efficient codecs, made room for academic works that would make use of both of these topics. The use of **Machine Learning** (ML) techniques alongside video codecs is something that is getting more attention recently, one of the reasons being because, as stated by (RAMACHANDRAN, 2018) the encoding problem is an optimization problem, and ML tools can be deployed for most of the optimization problems.

Another important factor for the use of machine learning tools, finds itself in the fact that video coding tends to work with losses, what this means is that usually the decoded video will not be an exact copy of the original, but it will pass through multiple filters to be as close as possible to the original. This characteristic often motivates the use of ML in video coding, since most of the imprecision given by an ML model would be dealt by the decoder, as long as the model maintain a certain level of precision.

Finally, one may note that one of the most important steps of multiple codecs is the probability distribution of its Syntax Elements (SE) (i.e., the data generated throughout the encoding process prior to the entropy encoding) (RAMOS, 2019). To shed some light in this process, it is a step where the syntax elements are organized according to their probability and then used for context modeling at the entropy stage, where the final bitstream is generated. Because of that, this is a step where an accurate probability calculation may directly impact the final bitstream, since the final length of the bitstream will be related to how accurate the probabilities were estimated.

1.1 Motivation

An approach that utilizes neural networks in video encoding has the potential to achieve higher compression gains when compared to the native solution, as demonstrated in the work of (MA et al., 2019). However, further investigation is needed as the application of neural networks in the context of the AV1 format is relatively new and there are few works that have established a correlation between the two while also demonstrating the benefits and drawbacks of using an ML approach.

Another relevant factor for this monograph is that, as reported by (OZER, 2021), even though AV1 is able to achieve a compression degree that is greater or equivalent to the VVC and HEVC families, its major disadvantage lies in having a high encoding time. However, at each new version of the codec, this time is expected to be reduced.

Hence, this monograph will apply a ML approach to more accurately calculate the probability occurrence of the AV1's residual SEs. The main reason to consider specifically the residual SEs is because, as stated by (RAMOS et al., 2020) for HEVC, they are usually the majority of the SEs processed.

1.2 Objectives

The main objective of this work was to implement a neural network to estimate the probability of residual SEs (RSEs) of the AV1 format.

1.2.1 Specific objectives

1. Study and understand the AV1 video coding process.
2. Study on how the generation of probabilities of residuals SEs occurs in the AV1 codec.
3. Develop the dataset that will be used to train the neural network from video sequences encoded with AV1 software of reference.

4. Select the most appropriate method to train the neural network
5. Analyze the precision obtained by the model, as well as the architecture of the model.

1.3 Outline

The text is organized as follows: **Chapter 2** presents the main concepts related to video coding and neural networks.**Chapter 3** covers the employed methodology for this manuscript. **Chapter 4** brings a more detailed discussion on video coding syntax elements, arithmetic coding as well as some correlated works at the end of it. **Chapter 5** shows the development process as well as the final results at the end of the chapter. Finally, **Chapter 6** addresses the final considerations of this work along with some possible future works at the end of it.

2 THEORETICAL BACKGROUND

The beginning of this chapter aims to explain basic video concepts, such as frames, pixels, color spacing and sub-sampling. Next, an introductory section to the workflow of the most popular codecs, and later on, a section specifically made to talk about the AV1 format and its particularities. Furthermore, the end of this chapter will briefly talk about Neural Networks.

2.1 Basic video concepts

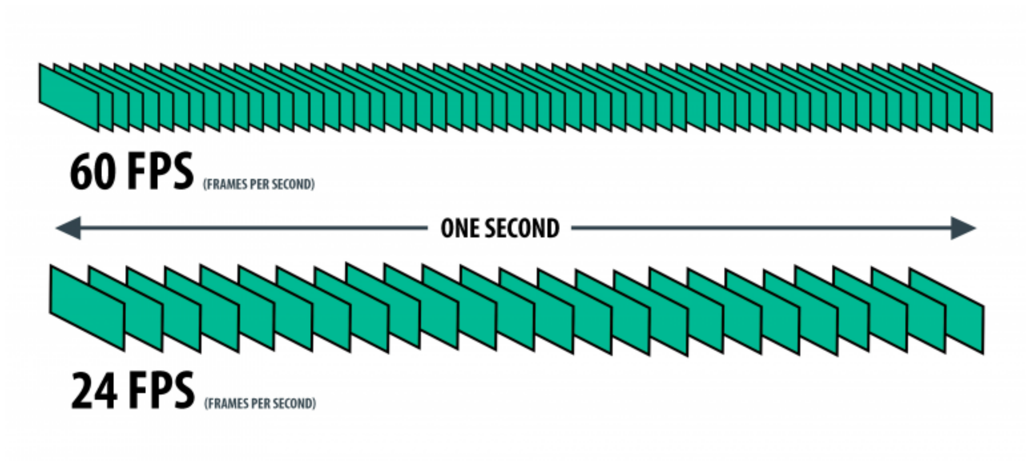
2.1.1 Frames

A video is a sequence of static images displayed in a quick succession after one another, where these images, in a video, are called **frames**. The sensation of movement given by a video is achieved by a rapid succession of frames. Due to the high speed that each image is shown, the human eye is unable to notice that there is a time between each image, giving the sensation of movement that we see in a video (RAMACHANDRAN, 2018).

Since a video cannot let the viewer notice that there is time between each frame, to be able to perceive something as a video, we need to establish the minimum time required between each frame, the metric responsible for this is called **FPS** (i.e., short for Frames Per Second). The minimum amount of FPS needed to achieve this sensation of movement is usually 24 FPS.

The frequency on which a video is changing frames gets the name of **temporal resolution**, and it is measured in FPS. Figure 1 shows the difference between two different temporal resolutions, in one second of a video, measured in FPS. To summarize, the general rule of thumb is, the smoother the video needs to get, the higher fps it will need. (RICHARDSON, 2010).

Figure 1 – Temporal Resolution

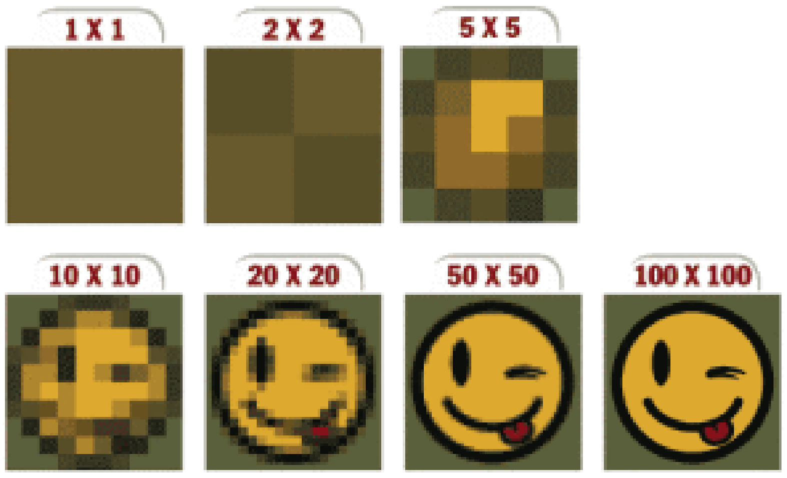


Source: Animotica (2020).

2.1.2 Pixels

Since a video is said as a sequence of frames, then, what is a frame? A frame is a matrix of pixels that determine the **Spatial Resolution** of a video. The spatial resolution could be seemed as what we usually call **resolution** for any given video. For example, if we have a resolution of 1920×1080, then it means that the matrix has 2,073,600 pixels, with 1920 columns and 1080 rows. Each pixel is described by color and light components that constitute a **Color Space**, thus we could see each pixel as a minuscule part of a frame, and when we put each pixel at his designate place, thus generating a frame. Figure 2 illustrates this concept for multiple resolutions.

Figure 2 – Multiple Resolutions

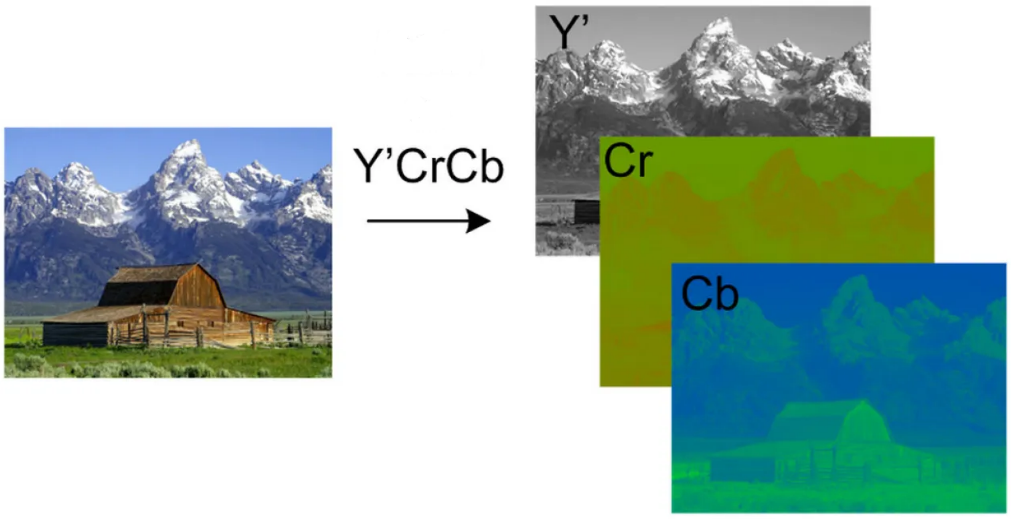


Source: Emporio (2020).

2.1.3 Color spacing

According to Ramachandran (2018) a color space maps real-world colors to discrete values, and it is a specific implementation of a color model. In other words, is a way to "color" a pixel in a meaningful way for the frame. Essentially, each color component of the pixel will have an assigned value to it, which will later on be processed and represented on the frame as a color. The most common color space in video coding is the **YCbCr** (Luminance, Chrominance blue, and Chrominance red), where Y refers to the light intensity of the pixel (or Luma for short), while Cb and Cr are both color components also referred as Chroma. Figure 3 illustrates the separation of a frame on each one of its color components.

Figure 3 – Color Components



Source: Adapted from Hisour (2021).

The YCbCr is the most common Color Spacing because of some of its characteristics that can help to reduce the amount of bits required to represent a pixel. One of these characteristics is in the fact that the Luma and Chroma components are completely independent of each other. To show why this is a characteristic that can achieve compression gains, we first need to refer to the physiognomy of the human eye, which, as stated by (RICHARDSON, 2010), is more sensitive towards Luma components than it is to Chroma. Simply put, we are more sensitive to variation in light than we are for variations in the color itself. This latter characteristic allows something called “Chroma Sub-sampling” which is, basically, a process to remove some of the chroma components without having subjective losses on the frame itself, at the same time being able to obtain compression gains.

2.1.4 Bit-Depth and Subsampling

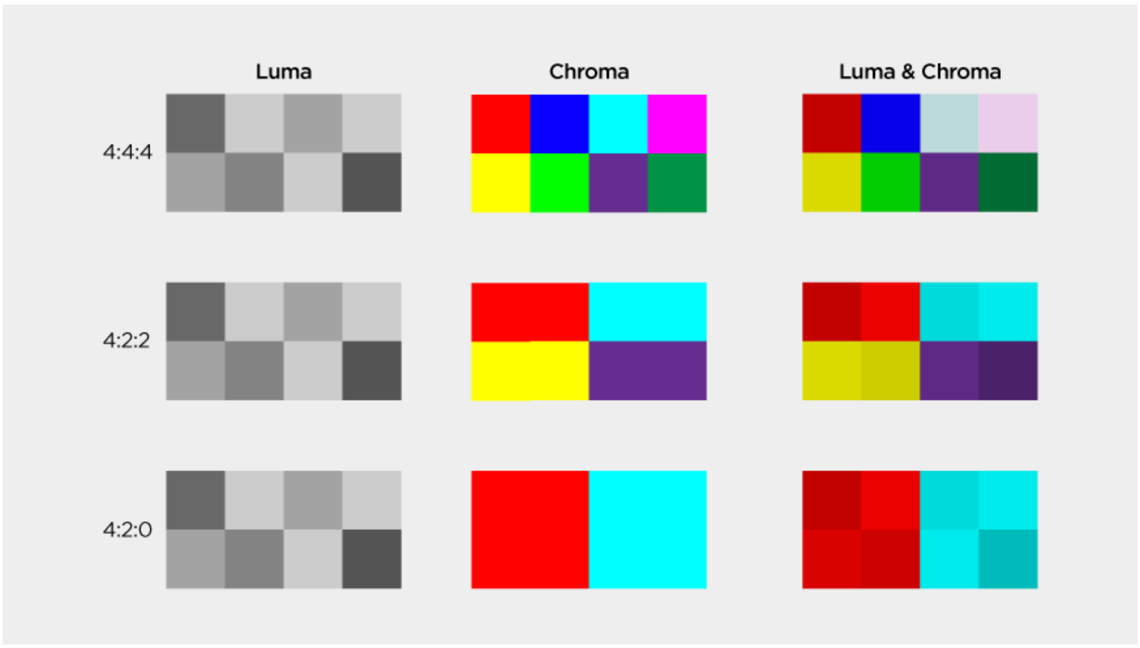
Since pixels contain color components that are described as numbers in the discrete world, we have to consider something called **Bit-depht**. One may consider a pixel that is described in terms of YCbCr components, and that each one of these components has a Bit-depht of 8-bits wide. Hence, we would notice that the amount of colors that this pixel could represent is $2^8x2^8x2^8$ or 16 million colors. Bit-depht is what dictates how many colors a given pixel can express, whereas the higher he is, the closer the pixel’s color will be to reality.

Although it may seem desirable to always be as close as possible to reality, sometimes, this is not necessary, and, in these cases, the process of Chroma Sub-sampling, for instance, is applied. There are basically three Chroma Sub-sampling methods that are commonly used, being then:

- 4:4:4: For each four Luma (Y) samples, also four Chroma blue samples (Cb) and four chroma red (Cr) samples. This is also known as a “No sub-sampling” format. It is applied when the objective for the pixel is to be as close as possible to what was captured, because of this characteristic it is not possible to obtain compression gains, since the color scheme is meant to be as close as possible to what was captured in the frame.
- 4:2:2: For each four horizontal Luma (Y) samples, also two Chroma blue (Cb) samples and two Chroma red (Cr) samples. This sub-sampling method promotes 33.3% of data reduction.
- 4:2:0: For each four Luma (Y) samples, only one chroma blue (Cb) sample and one Chroma red (Cr) sample are processed. this method promotes 50% of data reduction. This approach is mostly used when a high compression rate needs to be achieved, at some loss of subjective quality not perceived by human sight, e.g., video coding in general.

Figure 4 depicts examples of 4:4:4, 4:2:2 and 4:2:0 sub-sampling processes.

Figure 4 – Chroma subsampling



Source: HAIVISION (2020).

2.2 Video coding basics

Video coding, or video compression, is the process on which a video sequence is compressed into a **bitstream** that describes it. Video compression tends to be extremely important for any video-related application. The reason for it being that video files in their raw format are absurdly expensive in terms of bandwidth and storage space. Table 2 shows different sizes of a raw video file of ten minutes length and the bandwidth required for each resolution, while considering 12-bits for each color component.

Table 2 – Raw videos size and required bandwidth

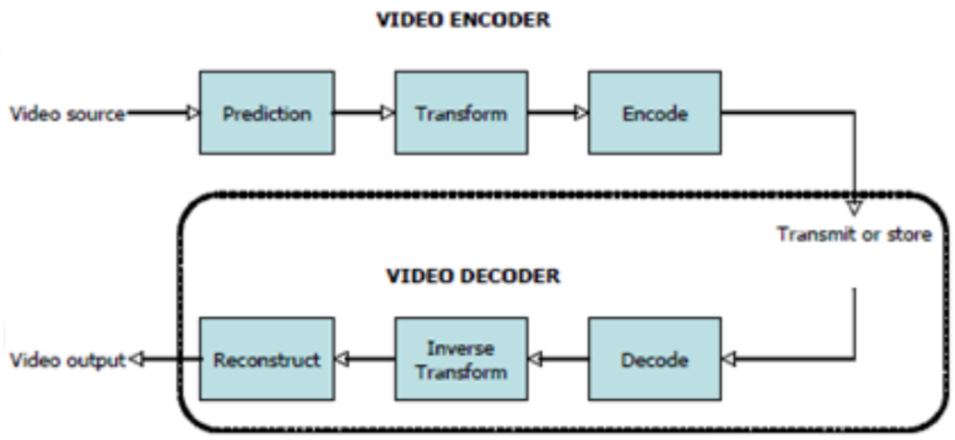
Resolution	*Size (GBytes)*	*Bandwidth (Mbytes/s)*
834x480	10	17.13
1920x1080	52	88.98
2560x1600	103	175.78
3480x2160	189	322.58

Source: Monteiro (2017)

The compression and decompression is usually done by a **codec**, where the encoder receives a raw video file, apply a set of tools intended for coding purposes, and then generates a bitstream for that video sequence. This encoded bitstream can be decoded back to the original video with the help of a **decoder**. The reason why we call both of them alongside as codecs is that one does not work without the other. In other words, we cannot compress a video using the AV1 encoder and, later on, try to decode it with the HEVC decoder, since this scenario would not work. The reason for this incompatibility among formats/standards being that each of them will have his own bitstream specification.

If we took a closer look at the workflow of the most common standards/formats (HEVC, AV1, VVC, etc.), we would notice that they all follow the so called hybrid-process. The hybrid-process is mainly composed of a sequence of predictions, transforms and additional steps, as depicted by Figure 5.

Figure 5 – Basic workflow



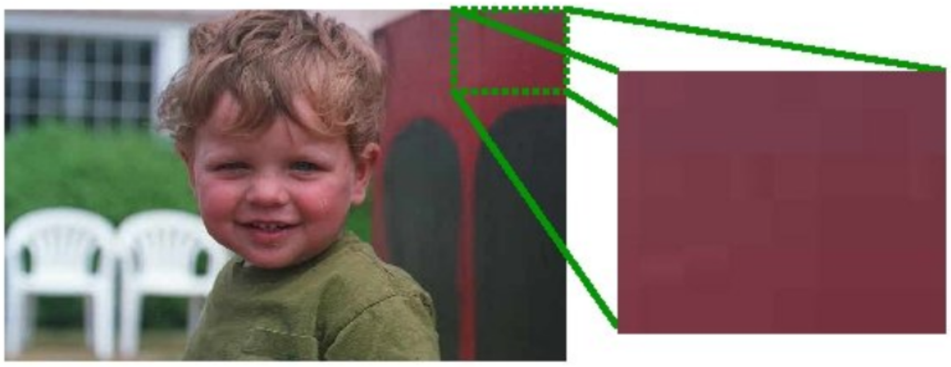
Source: Adapted from VCODEX (2022).

Another important characteristic to mention is that a compression can be lossy or lossless, what it means is that lossy is a type of compression that is going to have definitive changes in the pixels values of the decoded video (i.e., on the quality), at the exchange of compression gains, whereas lossless, as the name indicates, is a compression that decoded video will be exactly the same as the original one.

As mentioned earlier, codecs look for redundancies in a video file to obtain compression gains, these redundancies can be divided into 3 types, being then:

- Spatial Redundancy: Spatial redundancy intends to find compression gains by analyzing the similarities between pixel blocks of the same frame This type of redundancy is also known as intra-frame redundancy and is used, for instance, in the first frame of the video, since there is no other frame to use as reference. Figure 6 shows an example of similar pixels able to be classified as Spatial redundancy.

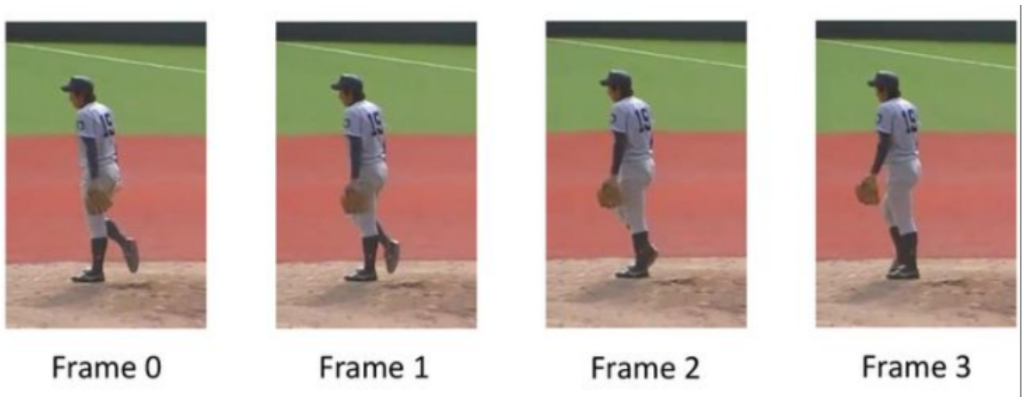
Figure 6 – Spatial Redundancy



Source: Adapted from Jaewhon (2015).

- Temporal Redundancy: Temporal redundancy is the one that exists between different frames. For instance, if there is a group of pixel that is an exact copy of its predecessor frame, or just shifted a bit, or even only with slightly changed values, then it will be considered as Temporal Redundancy. This type of redundancy is also known as Inter-frame redundancy, and according to (ZATT, 2012) is the main source of compression gain for video coding. Nevertheless, it tends to be the main bottleneck of the execution time to encode a video. In Figure 7 we can see that a lot of pixel blocks do not change between frames, indicating the presence of Temporal Redundancy.

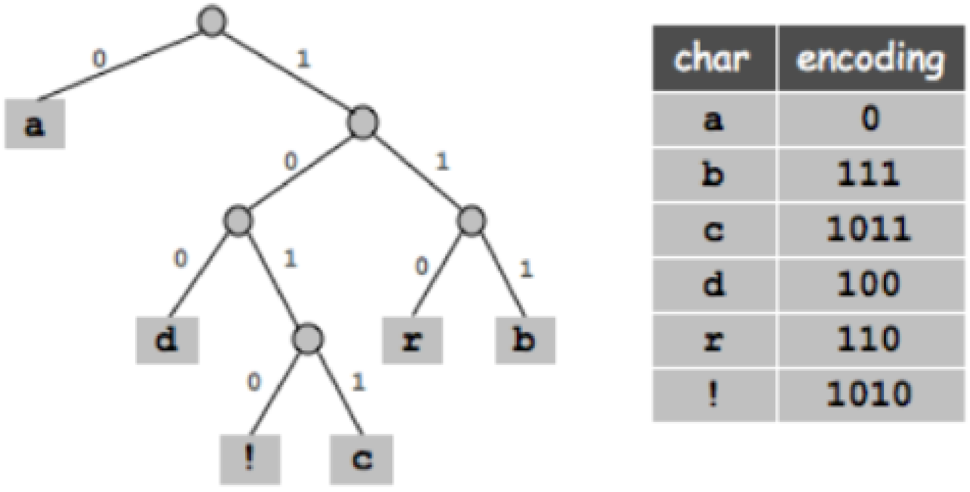
Figure 7 – Temporal Redundancy



Source: Adapted from ISCA (2018).

- Entropic Redundancy: This type of redundancy comes from the concept of information theory presented by (SHANNON, 1948). The basic idea is that as the likelihood of a given event increases, the less information it carries and vice-versa. What this concept implies is that the information that is more likely to occur can be represented with fewer bits. Figure 8 represents this idea by using a Huffman tree, it

Figure 8 – Huffman Tree



char	encoding
a	0
b	111
c	1011
d	100
r	110
!	1010

Source: Adapted from Princeton (2004).

is possible to notice that 'a' is the most common char, while '!' is the least common one (although a Huffman tree is not applied in video coding, it demonstrates a similar concept).

2.2.1 Predictions

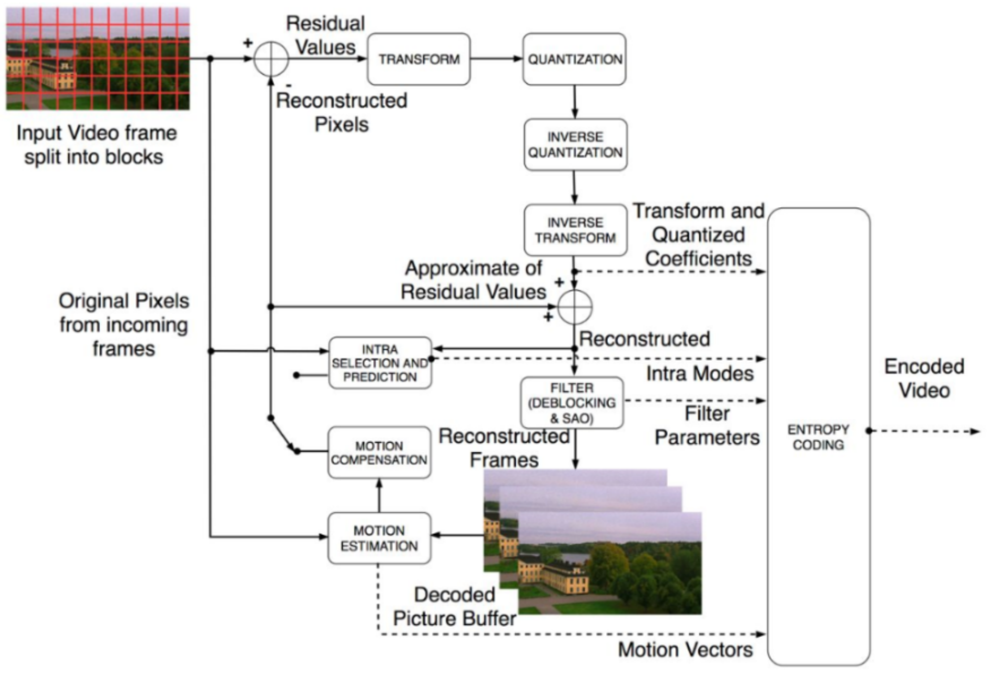
During the encoding process of a video, the default procedure is to process pixels as a group, instead of going them one by one. Usually, they are organized as squares or rectangles to make more efficient use of the coding techniques. These pixels organization receives the name of **blocks**. If we took HEVC as an example, the respective size of each of these blocks can go to as big as 64×64 pixels or as small as 16×16 pixels (RAMOS, 2019). To summarize it, blocks are a fashion to process pixel groups of multiple sizes in the most efficient way, where the size of the block usually varies from one codec to another.

The way that codecs usually explore the previously mentioned redundancies is by making **predictions**. Predictions are made, as one may notice, by using pixel blocks, and essentially, what a prediction is doing, is to use a given pixel block and tries to recreate it using as few bits as possible. Intra-frame predictions try to recreate the pixel block by using spatial redundancy, whereas Inter-frame predictions try to recreate the frame by temporal redundancy. As previously mentioned, Inter-frame prediction is the process responsible for most of the compression gains.

After building a prediction block, a subtraction is made with the original block, resulting in the difference between each block, where these differences are the **residues**, and are used to generate a residual block. The residual block is used on the next coding stages, generating an even greater data reduction. Finally, the **bitstream** is generated, where it will be a binary file that obeys a structure described by the codec. This behavior means that the bitstream will be the final piece of the codification process, and to be later on decoded, it needs to follow a specific pattern. This pattern is described by a document called bitstream description, a document responsible to describe how the bitstream is organized. By standard, when a codec is developed, two artifacts are expected to be produced, the first one being the bitstream description and the other one being the software of reference, both of them are necessary to use and understand the produced codec.

As briefly mentioned earlier, most of the codecs follow the Hybrid-process, a process composed of four steps, being then: predictions, transforms, quantization, and entropy coding. This process appears in Figure 9, exemplifying this process from the

Figure 9 – Hybrid Workflow



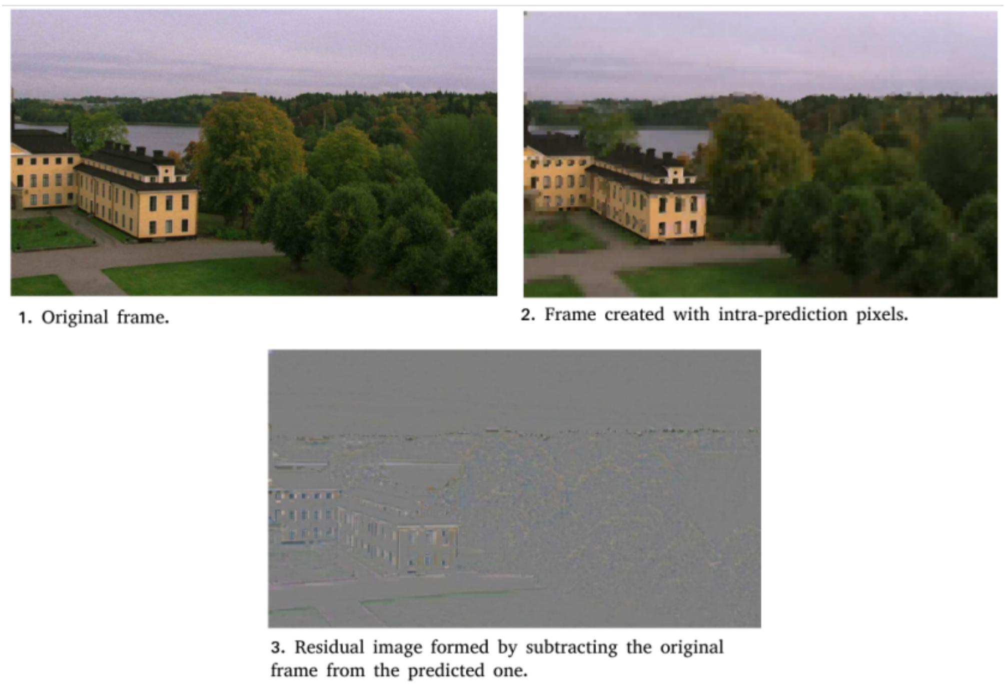
Source: Ramachandran (2018).

original input video to the bitstream output. If we took a closer look at the aforementioned Figure 9, we would notice the existence of Inverse Quantization, Inverse transform, and the application of filters. The reason for that is that since the quantization step insert losses in the coding process, all of these steps are necessary for the decoder to rebuild the frame as close as possible to what was originally hold.

2.2.2 Intra-prediction & Inter-prediction

The intra-frame prediction uses a group of pixels to obtain gains with spatial redundancy. These predictions can be directional, where pixels from neighboring blocks are used to create a trajectory. On the other hand, the predictions can also be non-directional, where the neighboring pixels are combined in various ways (BORGES, 2019). Usually, intra-prediction uses pixels from already-encoded adjacent blocks and derives the predicted pixels by filling entire blocks with pixels extrapolated from the above row and from the column to the left of the ongoing block (RAMACHANDRAN, 2018). Summarizing it, by utilizing each pixel group of a given frame, a predicted frame will be created, and after its creation, it will be subtracted with the original frame, thus creating a Residual Frame. This residual frame is what will be sent for the next coding stages. Figure 10 shows an intra-prediction being made.

Figure 10 – Intra-prediction


1. Original frame.
2. Frame created with intra-prediction pixels.
3. Residual image formed by subtracting the original frame from the predicted one.

Source: Adapted From Ramachandran (2018).

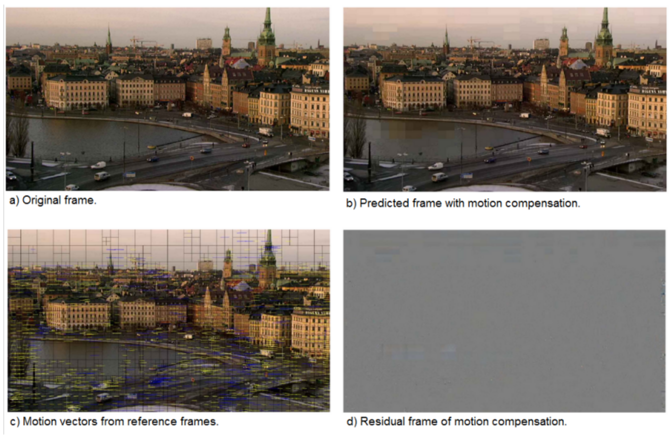
Intra-prediction frames tend to be very precise with areas without many details, but it suffers a lot when it has to predict places with a higher detail level, where these losses will be encoded as residual values on the bitstream. Another important factor about intra-frame predictions is that they are mostly used when there is no other frame of reference (i.e., when it is the first frame of a video).

Changing the subject to inter-predictions, they are the type of predictions that are mostly use throughout the coding process, they make use of temporal redundancy to obtain expressive gains in compression. The way how these predictions works is by selecting zones from previously encoded **reference frames** to find a candidate block that fits better the original block.

This search for the frames containing the best-predicted block is going to be called **Motion Estimation** (ME). Essentially, this process is looking to discover how much each pixel block is moving and selecting ones with the least amount of movement to be used as reference (RAMOS, 2019). Later on, the movement between the predicted block and the one used as reference will be called as **Motion Vector** (MV), expressed as an 2D array of pixel coordinates. The process that will employ the aforementioned MVs to create the predicted block is named Motion Compensation (MC), and, as stated by (RAMACHANDRAN, 2018) this stage, in terms of computational resources, is one of the most expensive stages in video coding.

At Figure 11 we are able to see an example of motion compensation. Something

Figure 11 – Motion Compensation

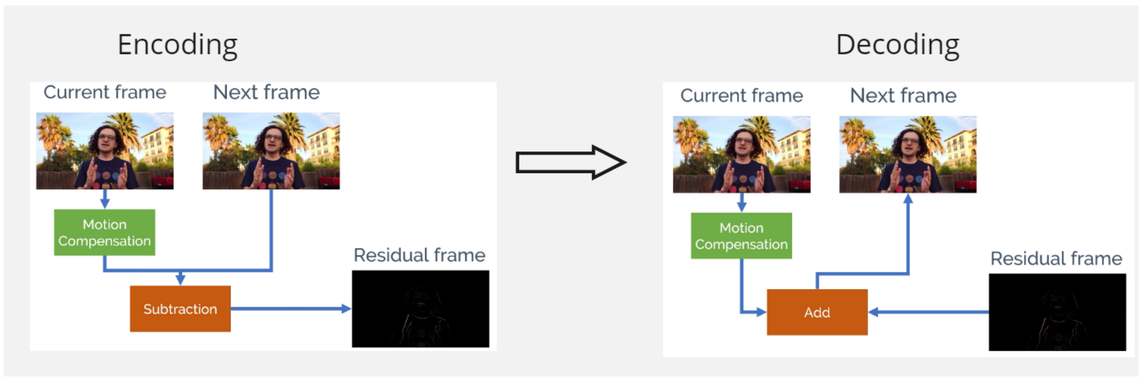

a) Original frame.
b) Predicted frame with motion compensation.
c) Motion vectors from reference frames.
d) Residual frame of motion compensation.

Source: Adapted From Ramachandran (2018).

important to note is the algorithm responsible for the ME, it is extremely important for how much loss will be passed to the next stages. The reason being that, if ME finds a bad fit, it will need more bits to represent this residual error, leading to loss in compression.

Since inter-predictions make use of previously predicted blocks in given reference frames that may contain losses, the consecutive codification of multiples frames using this type of prediction will result in the propagation of residual errors and, consequently, reducing the compressed video quality. To solve the mentioned problem, an intra-frame prediction occurs at regulars intervals, allowing the visual quality of the video sequence to be maintained. Figure 12 represents the general flow of execution to encode and decode a given frame using an inter-prediction.

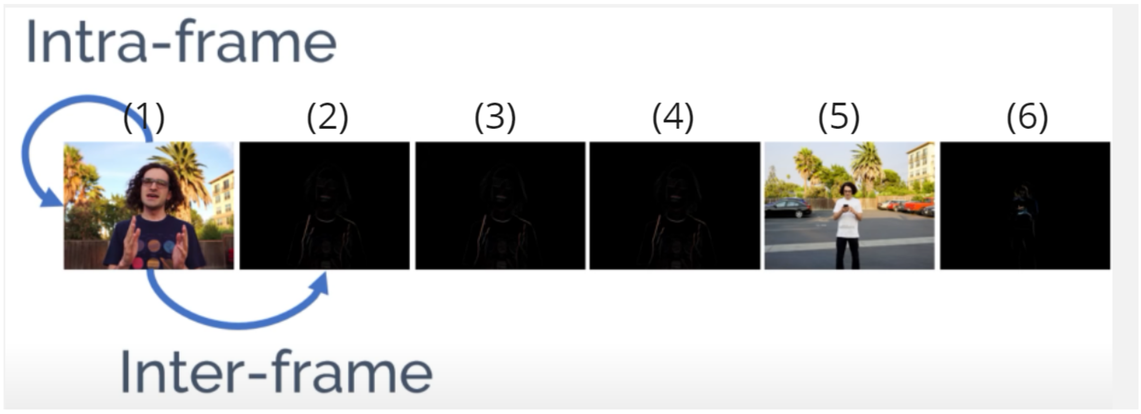
Figure 12 – Inter-frame workflow



Source: Adapted from Isikdogan (2018).

Another point worth mentioning is that each time there is a drastic change in the next frame, an intra-frame prediction will be used. This can be seen in 13, where the first frame is encoded using intra-frame prediction. After that, since there are no major changes between frames, inter-prediction is used from frame 2 to 4. Finally, at frame 5 we can see another use of intra-prediction due to the significant change between frames.

Figure 13 – Sequence of frames



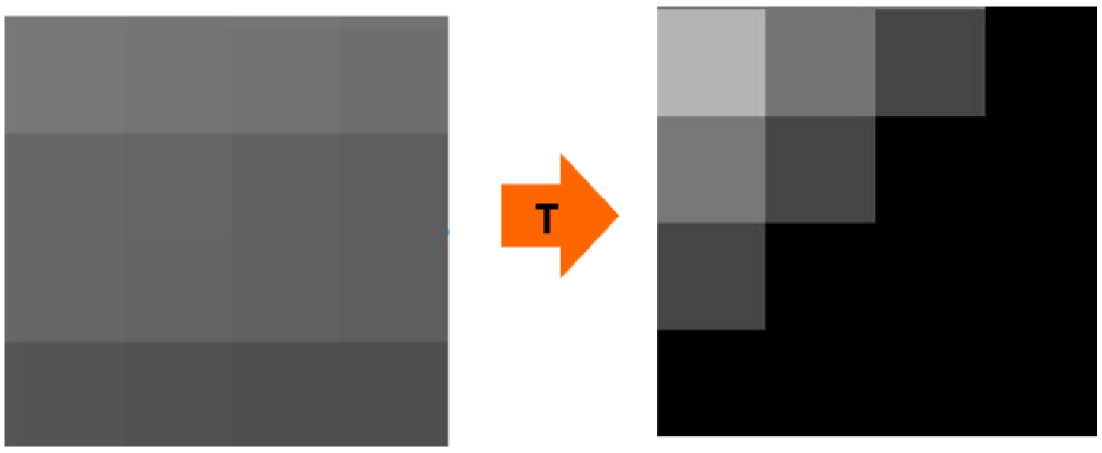
Source: Adapted from Isikdogan (2018).

To summarize, Figure 13 also presents the general concepts that both predictions are following. As one may observe, in the intra-frame prediction, the frame is encoded without the use of its neighbors, whereas the inter-frame prediction considers only the movement between each encoded frame.

2.2.3 Transform & Quantization

The transform stage begins after the aforementioned prediction stage, and its main focus is to rearrange our residual frame in a way that only some few coefficients remain significant (RAMOS, 2019). This process occur with the use of **Transform Units** (TUs), where TUs are responsible for partitioning a residual frame into multiple pixel blocks that will be internally rehanged using a given transform type, e.g., the **Discrete Cosine Transform** (DCT). Figure 14 shows an example of a 4×4 pixel block after the transform step, where we can visualize the effects of the transform by comparing the block before the transform and after the transform. After the process, the block is rearranged in a fashion that the smaller values, i.e., near zero, (seemed as black blocks) always get at the bottom-right of the block. The quantization step begins after the transform

Figure 14 – Transform Stage

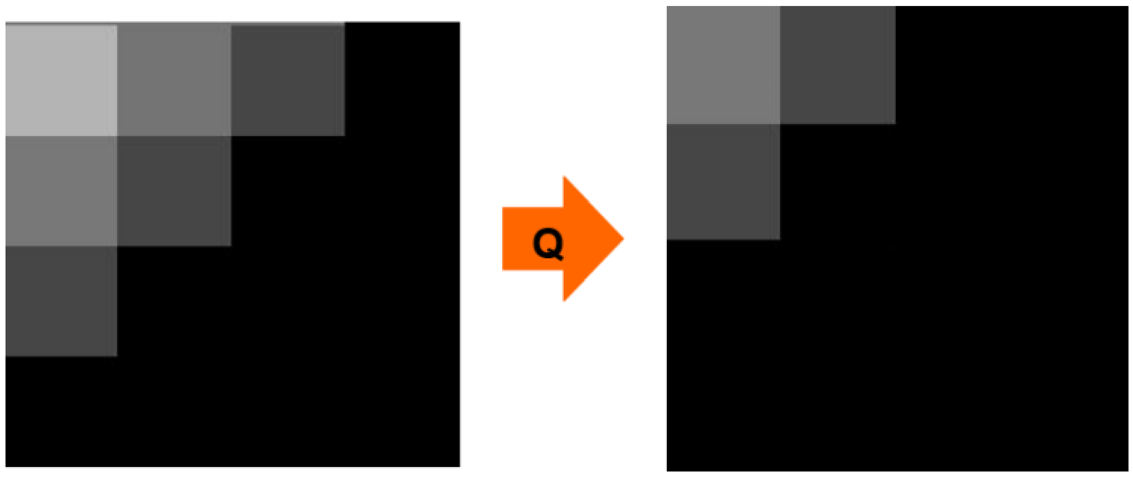


Source: Ramos (2019).

step and is responsible for the losses in this compression stage. The quantization will perform the equivalent of an integer division on the coefficient block that was created after the transform step. As previously mentioned, this stage will generate losses on the quality of the image. The argument responsible for the amount of loss is usually called **Quantization Level**, where the higher it is, higher the losses, since the coefficients will be more likely to become zero. On the other hand, as we lose in quality, we gain in compression, since zeros are easier to compress.

Figure 15 shows the general idea of the quantization step.

Figure 15 – Quantization Effect



Source: Ramos (2019).

As for both stages, they need to have an inverse process that enables the reconstruction of the block. In the case of transform stage, it is called inverse-transform, and it will reconstruct the coefficient values back to the residues. For the quantization, it gets the name of inverse-quantization and, due to the nature of an integer division, it will be impossible to reconstruct some pieces of information (i.e., the zeros generated by the quantization), therefore inserting losses in the image.

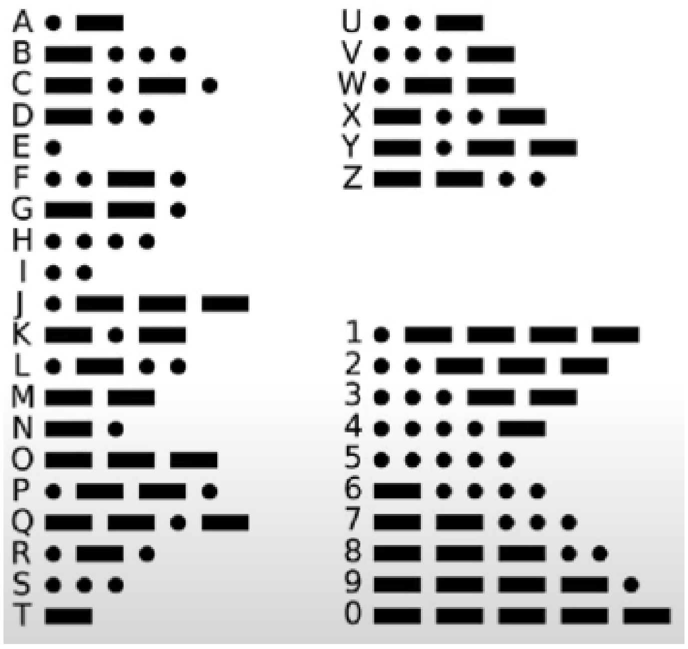
It is important to note that, after the quantization, the generated residual block will be the responsible to generate the main topic of this work, the so-called Residual SEs, and due to its importance, it will be discussed at his own chapter. Nevertheless, as a preview, they are generated between the quantization and entropy stages, and compose most of the SEs. As a final point, one may see the Transform and Quantization step as a preparatory stage for the Entropy stage, where this re-arrange will be put to use.

2.2.4 Entropy Coding

The entropy stage is the last stage of the Hybrid-workflow, where this is the stage responsible for generating the compressed bitstream. Anyways, the entropy stage will get as input what is formerly known as **Syntax Elements** (SEs) and make use of their occurrence probabilities throughout the coding steps to increase the compression gains.

The Entropy coding step is based on what we know from information theory, and as described by (RAMACHANDRAN, 2018), it is a way to measure the randomness associated with a specific content. For a more visual example, we could think of the morse alphabet, shown in Figure 16,

Figure 16 – Morse Alphabet



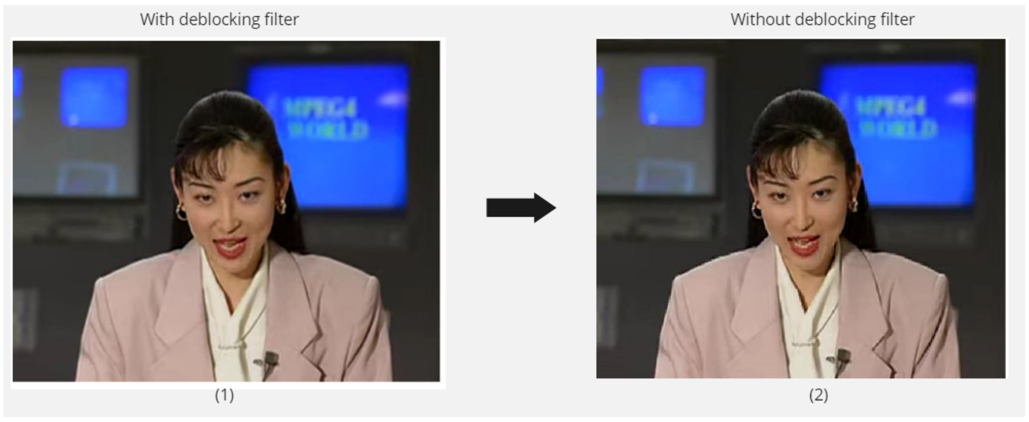
Source: Langmead (2020).

and what we have to take from this image is that the morse alphabet was built in a way that the most common letters are represented in less bits than the less used ones. If one takes the letter “E” as an example, we would notice that it is represented with a single dot. On the other hand, one of the least used letters of the alphabet “Z” is represented with two bars and two dots. The general idea is that the most used symbols of any given alphabet will be represented with fewer bits (or points/bars in the case of the Morse alphabet), where this idea translates adequately to video coding, since the final objective is to obtain a bitstream compressed as possible. All things considered, using SE and their specific probabilities, the entropy coding will generally make use of arithmetic encoding to encode each SE with a bit length respective to its frequency of use.

2.2.5 Filters

Since quantization insert losses in the coding step, it is normal to imagine that the reconstructed images sometimes will be different from what was originally compressed. To reduce these losses, we make use of filters. Filtering techniques are used during the reconstruction of the reference frames, which improves the visual quality and performance of inter-prediction. These operations get the name of in-loop filtering, because they occur during the encoding flux and can be return to be used as a reference for prediction. One filter that is commonly used is a **Deblocking Filter**, where this filter intends to smooth the edge zones of a given frame. Figure 17 shows an example of a deblocking filter being applied

Figure 17 – Deblocking Filter



Source: Adapted from Ramachandran (2018).

2.3 AV1 Format Concepts

As earlier described, AV1 aimed to bring flexibility for commercial use of codecs, while also maintaining on par with current codecs. As stated by (CHEN et al., 2018) the main focus of the AV1 standard was to reach a consistent delivery of high-quality real-time video, scalability for multiple bandwidths and hardware optimizations. AV1 allows higher bit rates, a wide range of color sub-sampling methods, while also providing higher bitrates.

AV1 follows the standard flow of execution of hybrid codecs described in previous sections. Because of that, the format operates with pixel blocks, that are further on processed in a prediction-transform scheme. The prediction comes from intra-frame reference pixels, inter-frame compensation and sometimes as a combination of both. After that, the residuals of the aforementioned stages serve as input for the transform stage, that will reduce the spatial correlation by reorganizing the block and changing the domain of the block to the frequency domain.

After that, the remaining coefficients are quantized to further increase the compression gains. Both of the SEs generated from prediction and quantization step are entropy-coded using an arithmetic coding algorithm. In parallel, there are three stages of post-processing in-loop filters, which aim to improve the quality of the reconstructed frame for it to be further utilized as a reference frame for subsequent frames. Another important point, is that the codec accept input signals in the formats 4:0:0 (monochrome), 4:2:0 as well as 4:4:4. Also function with bit depths of 8-,10- or 12-bits (HAN et al., 2021).

2.3.1 Frame Partitions in AV1

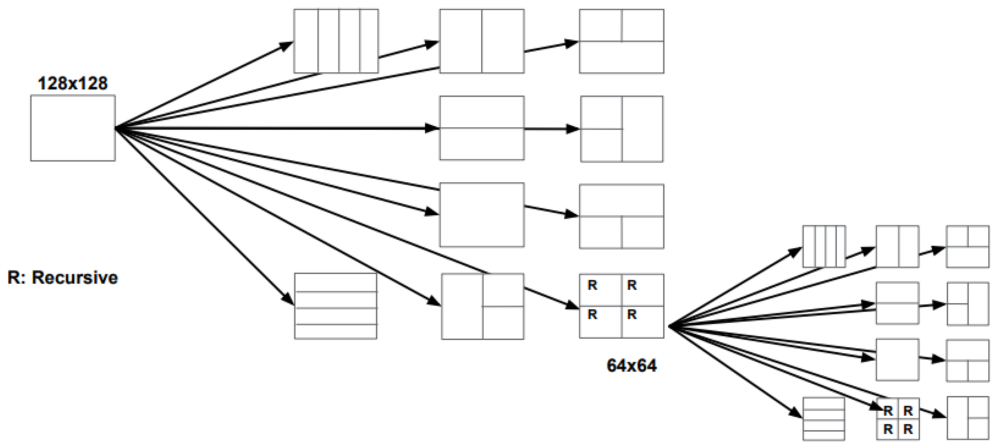
Each frame within a video sequence can be classified differently during the coding process, the chosen prediction type depends on this classification. In intra-frames prediction, the utilized frame will get the name of **KeyFrame**, and a video must have at least one frame of this kind (i.e., a frame that does not make use of temporal redundancy). The KeyFrame is also used to define subdivisions of the video into frame groups.

For the inter-frames prediction, the equivalent of the KeyFrame will be the **S-frame**, a frame that will make use of only future references to compose predictions. There is also the Golden Frame, where this is as a special type of frame that stores temporally redundant information across a subgroup of frames to which it belongs. Furthermore, for this type of prediction, each frame in AV1 can use up to eight references: One KeyFrame, one S-Frame and one GoldenFrame, up to three past frames and three future frames.

Since the AV1 follows the Hybrid-flow, a common technique applied by these formats is called frame partitioning, basically, dividing a frame into smaller parts to make the predictions more efficient and Accurate. The primary division unity of a frame of the AV1 format is known as **SuperBlock** (SB). The SB withstands two sizes, being them 128×128 or 64×64 pixels. The selected size is represented through a flag in the bitstream. An SB can be partitioned into smaller blocks, which promotes a better yield of the applied prediction and aids in the parallelism of the coding process (BORGES, 2019).

As stated by (HAN et al., 2021) The partitioning of an SB happens in a recursively quadratic and decreasing manner until some specific patterns get chosen. The subdivisions of an SB can assume three distinct formats: quadratic (1:1 (32×32 Pixels)); one-half rectangular (1:2, 8×16 Pixels); and, finally, one-quarter (1:4, 8×32 Pixels). Starting from a quadratic SB, it can divide itself to as low as 4×4 pixels, following a 10-path partition tree. Figure 18 shows the recursive structure for the partitioning block of the av1, as well as different block sizes.

Figure 18 – AV1's recursive partition tree



Source: Han et al. (2021).

AV1 also make use of tiles, which are rectangular vectors of SB's. Because of the spatial reference for them is limited to the tile borders, the tiles inside a frame can be independently encoded. This method facilitates the use of multi-threading in implementations of the AV1 standard.

2.3.2 AV1 Coding Tools

AV1 come along 56 intra directional modes, from angles between 36 and 212 degrees, enabling AV1 to target various types of spatial redundancies. AV1 also come with a variety of non-directional predictors that consider gradients, spatial correlation of samples and the coherence of the luma and chroma samples.

About inter-frame predictions, AV1 has an immense amount of tools to make use of the temporal correlation in video signals. Among these, are the adaptive filtering in translational motional compensation, affine motion compensation and highly flexible composite prediction modes. It is also important to note that the codec has two auxiliary motion vectors that inform the displacement of the current frame when comparing to the previous frame, along with the zoom effect.

The transform stage end-up requiring the block partitioning of a frame, but these can differ in shape or size when compared to the partitioning for predictions. The available partitioning for Transform blocks are: Squared, 2:1/1:2, and 4:1/1:4 in sizes ranging from 4×4 to 64×64 pixels. AV1 make use of four transform algorithms and combine them to be applied vertically and horizontally to the residual data, totalizing 16 possible transform combinations, the transform algorithms are: the DCT, ADST, flipADST and IDTX (CHEN et al., 2018).

After the transform stage, the quantization is applied in a way to reduce the coefficients, as well to remove coefficients that are not so important for the image. The quantization level is defined in the coding step by a paramater called **Constant Quality** (CQ) that vary from 0 to 63, i.e., 0 being lossless, while 63 being the maximum quality loss.

About the entropy stage, AV1 uses a multi-symbol arithmetic encoder adaptive to each one of the 16 encoded symbols. The SEs can be in an alphabet with up to N elements, and the context modeling consist of a set composed of N probabilities, where N is an integer varying from 2 to 16. The probabilities are also stored as cumulative distribution functions (also called CDF's), with a 15-bit precision.

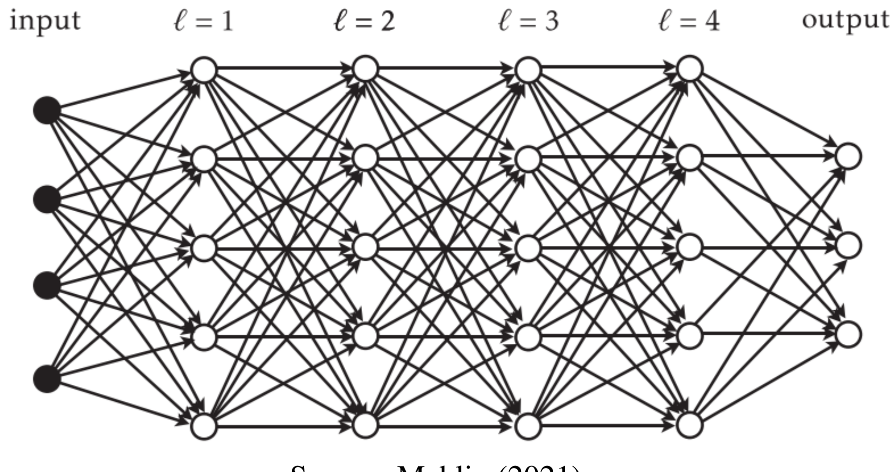
The last stage worth mentioning is the filtering stage. Up to this stage, losses have been inserted because of the quantization, color sub-sampling, etc. To soften these errors, multiple filtering techniques are applied. Some filters of the AV1 are inherited from VP9. Essentially, the AV1 permits three optional stages for in-loop filtering stage: deblocking filter, Constrained Directional Enhancement Filter (CDEF), and finally, a loop-restoration filter. After the filtering process, the output frame is then utilized as a reference for the encoding of future frames.

2.4 Neural Networks

2.4.1 Basic Structure of Neural Networks

To be able to understand a Neural Network, we first need to question, what even is a Neural Network? Neural networks, as said by (BRASPENNING F. THUIJSMAN, 1995) are computational models with particular properties, and they can have the ability to adapt, to learn, to generalize, or to cluster and organize data. The first wave of interest in Neural networks comes in 1943, after the introduction of simplified neurons by the work of (MCCULLOCH; PITTS, 1943). These neurons were presented as models of biological neurons and as conceptual components for circuits that could perform computational tasks (BRASPENNING F. THUIJSMAN, 1995). Get these neurons, link them, valuate each one of its inter-connections, and you will get an **Artificial Neural Network** (ANN), also known as **Feed-Forward Neural Network** (FFNN) (GOODFELLOW; BENGIO; COURVILLE, 2016). Figure 19 present a fully connected FFNN.

Figure 19 – Fully connected feed-forward neural network

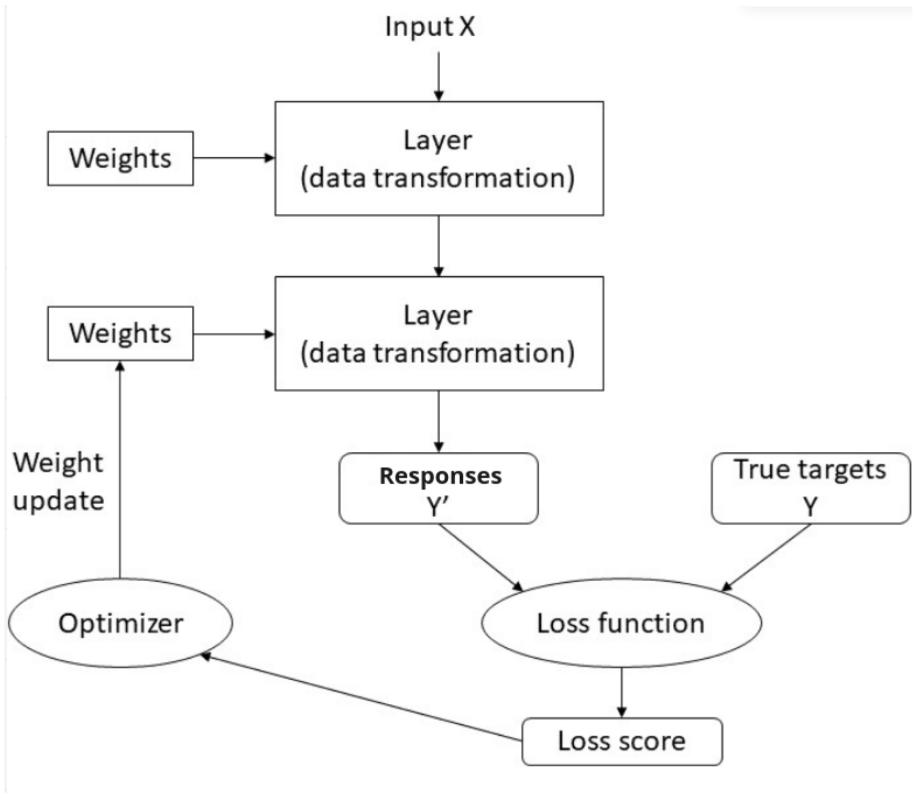


Source: Mehlig (2021).

The dots on the image represent neurons, each layer is composed of four neurons, each arrow will have a specific weight (i.e., a digit of the set of the Real numbers) attached to it and each neuron will have an activation function. The layers between the input and output are called hidden layers. The input is what the network will receive to be able to achieve its proposed functionality. To be functional, the model in the image will be trained through a predefined dataset along with backpropagation techniques.

Besides that, the amount of layers that a network has, the amount of neurons that each layer has, the function that each layer is applying, the dataset, all the aforementioned characteristics, will be extremely important to achieve a high precision rate. Generated Models are usually evaluated with the use of a cost function responsible to determinate how far the output neurons were to the correct answer. Figure 20 illustrate this workflow.

Figure 20 – FFNN general workflow



Source: Adapted from Dan (2019).

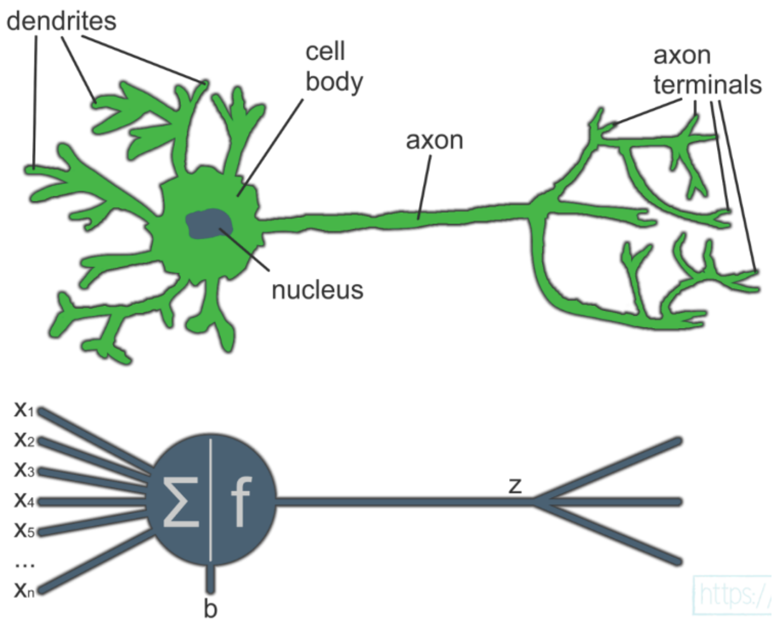
The FFNN architecture was something like a predecessor for the **Convolutional Neural Network** (CNN) and **Recurrent Neural Network** (RNN) architectures. Speaking of those, they follow a similar workflow, but with some tweaks between the execution or training stage. CNNs are frequently used in image processing, while RNNs are mostly used for temporal series (i.e., tasks with multiple steps that are extremely correlated to one another).

Essentially, a network is a tool that is broadly used to detect trends that are usually difficult to discern. The idea here is to use a known architecture to create a model that with proper training will be able to say if something is X or Y with a certain level of precision. The inputs will be respective of the task, as well as the activation function used in the output layer. The next section will cover most of the components mentioned earlier (i.e., neurons, layers, weights, etc.)

2.4.2 Neurons, Weights & Biases

Neurons are the most fundamental component of a Neural Network and the one that originated all of that. Firstly known as a perceptron from (ROSENBLATT, 1958) work, the neuron (and the whole network architecture) were inspired by our own neurons - Figure 21 shows a direct comparison between them - where X_1 to X_n are the inputs of the neuron, Z is the output and b is the bias. Each input will have its respective weight attached to it, then each one of the inputs will multiply its respective weights and after that, the results of each product will be summed, along with the bias. Finally, after this last sum, the result will go through an activation function and then will pass its results to the next neurons where it will be used as input, repeating this cycle until it goes to the output layer.

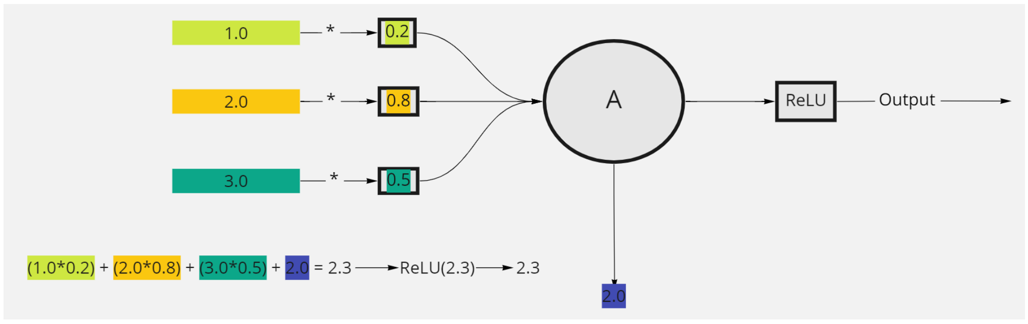
Figure 21 – Neuron comparison



Source: Kinsley (2020).

One may see how this artificial neuron will function through an example. Let A be an artificial neuron, X the set of inputs = $\{1.0, 2.0, 3.0\}$, W the set of weights = $\{0.2, 0.8, -0.5\}$ and p the bias = 2.0. The output of this neuron would be 2.3, as depicted in Figure 21. One may notice the existence of a **ReLU** (Rectified Linear Unit) function at the output of the neuron, where ReLU is an **activation function** (activation functions will be briefly covered on the following section). A ReLU function is one of the simplest activation functions, if the output is negative, then it gets passed as 0, otherwise, the output just passes through unchanged, to become an input for the next layer. In the case of the example, the final output of A will be 2.3. Summarizing, this whole process can be

Figure 22 – Neuron Example


1.0
*
0.2
2.0
*
0.8
3.0
*
0.5
A
ReLU
Output
(1.0*0.2) + (2.0*0.8) + (3.0*0.5) + 2.0 = 2.3 → ReLU(2.3) → 2.3
2.0

Source: Author (2022).

described by equation 1:

$$\bar{y} = ReLU(\sum_{j=1}^{d} w_j x_j + b) \quad (1)$$

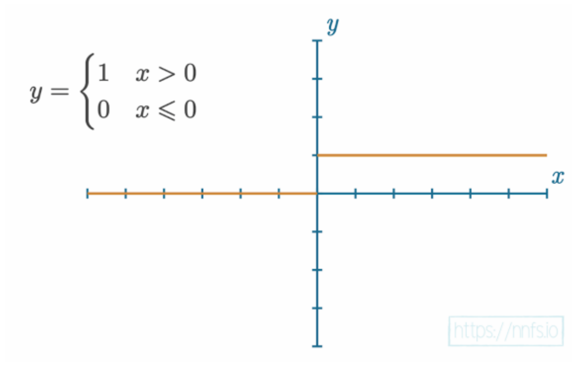
y is the output of the neuron, while d is the number of neurons that were present in the previous layer.

If we took this process that occurs for a single neuron, and increase its scale, we would end-up in a FFNN architecture that was earlier described. What we really have to grasp from this section is that this process of a single neuron is replicated to every single neuron of an entire neural network. The biggest difference that we will have between each neuron is the activation function, that can vary between each layer. Neurons of the same layer will have the same number of inputs and the same activation function.

2.4.3 Activation Functions

Activation functions are what will usually regulate the "objective" of the Neural Network model, for example, an FFNN can be a regression or a classification model, depending on the activation function used in the output layer (BURKOV, 2019). Since the neuron was based on an artificial neuron, the activation function was not different, it was initially built to emulate a neuron firing or not, where this function got the name of **Step Activation Function** (KINSLEY, 2020). Figure 23 shows this function *x* is the output without the use of an activation function.

Figure 23 – Step Activation function

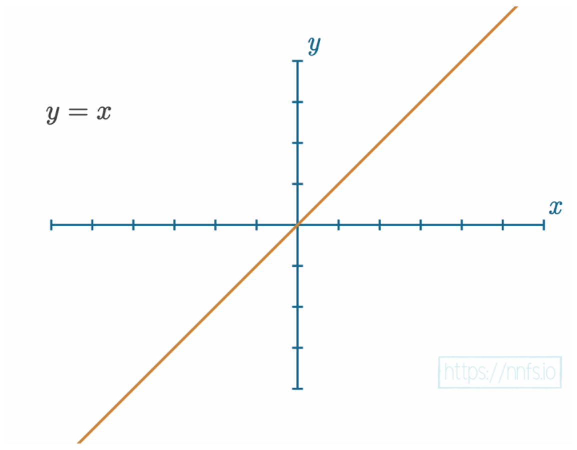


Source: Kinsley (2020).

Since the utilization of activation functions is contingent on the objective of the Neural Network, a wide range of activation functions exist to suit specific contexts. For the purpose of illustration, we will examine three of them as follows:

- **Linear Activation Function**: as stated by (KINSLEY, 2020) This activation function is usually applied to the last layer's output in the case of a regression task. Figure 24 depicts the output of a linear function

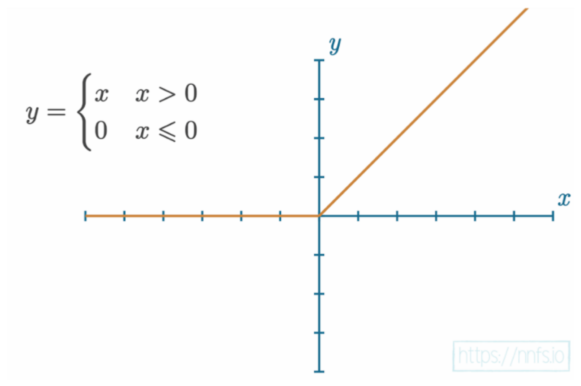
Figure 24 – Linear activation function



Source: Kinsley (2020).

- **Rectified Linear Activation Function** (ReLU): The ReLU is essentially a linear function, but without negative values, it is often used in hidden layers, because of her simplicity and speed, and as stated by (MEHLIG, 2021) the network of active neurons (non-zero output) is sparsely connected. Sparse representations of a classification problem tend to be easier to learn because they are more likely to be linearly separable. Figure 25 shows the output of a ReLU function

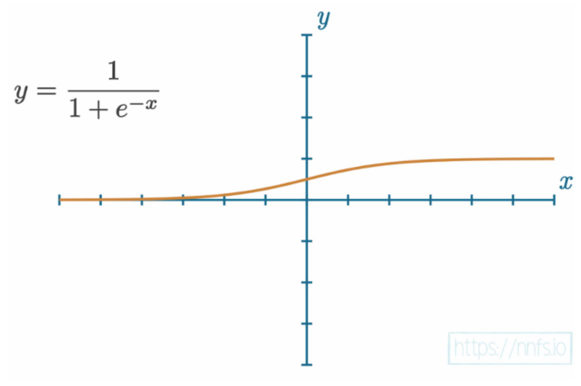
Figure 25 – ReLU activation function



Source: Kinsley (2020).

- **Sigmoid Activation Function**: this is an activation function that is often used for classification. This function returns a value in the range of 0 for negative infinity, through 0.5 for the input of 0, and to 1 for positive infinity. Figure 26 shows the behavior of a sigmoid function

Figure 26 – Sigmoid activation function



Source: Kinsley (2020).

- **Softmax Activation Function**: Last, but not least, the Softmax activation function is used in classification tasks, at the output layer, its main strength is to be able to represent the output in terms of its probabilities (MEHLIG, 2021). It is easier to represent its purpose with a short example. Imagine that you have built a model to identify whether an image is a cat or a dog, this model will have two output neurons, one for each possibility. The value of both neurons will be a probability, for example, if you had the image of a dog as an input and the model was not so sure about it, neuron A would be represented as 0.6% and neuron B would be 0.4%. Softmax functions are extremely useful for multiclass classification, where each one of the neurons at the output layer represents the probability of a given class. The following equation describes this process

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

Where:

- z is a vector of arbitrary real values
- n is the number of elements in the vector
- $\sigma(z_i)$ is the softmax activation for each element z_i

To conclude, activation functions are extremely important for the training process of an NN model, as well as its objectives. Usually, the activation function used at the output layer dictates the objective of the whole model. Besides the activation function, the number of neurons at the output layer is also important for the purpose of the network, for instance, in a neural network that was built for a classification task, each neuron at the output layer will be the equivalent for one class, on the other hand, for a regression model, it is mandatory that the output layer is composed of a single neuron.

2.4.4 Loss functions

Loss functions are used to measure the performance of a Neural Network when predicting a set of inputs during its training process (GOODFELLOW; BENGIO; COURVILLE, 2016). These functions are chosen based on the task that the Neural Network is attempting to perform. For example, the cross-entropy loss function is often used for classification tasks, as it measures the difference between the predicted probability distribution and the true probability distribution. This function is frequently used with the softmax activation function, which converts the outputs of the model into a probability distribution across the classes. There are many different loss functions available, and their usage depends on the specific task that the Neural Network is trying to complete. Some commonly used loss functions will be listed below.

- **Mean Squared Error** (MSE): The MSE is mostly used on regression tasks, The MSE loss function measures the average squared difference between the predicted and true values, its biggest strength is found in its ability to penalize large errors more heavily than small errors. (GOODFELLOW; BENGIO; COURVILLE, 2016)
- **Mean Absolute Error** (MAE): Is a loss function commonly used in regression tasks. As its name suggests, it calculates the mean absolute error between the predicted value and the true value. The smaller the MAE, the more precise the model is at predicting an exact value. It is similar to the MSE loss function, but it does not penalize large errors as heavily and is more robust to outliers in the data. The MAE loss function is generally easier to interpret than the MSE loss function because it is expressed in the same units as the data being used (GOODFELLOW; BENGIO; COURVILLE, 2016).
- **Mean Absolute Percentage Error** (MAPE): The MAPE loss function measures the average absolute percentage difference between the predicted and true values. It is often used when it is important to accurately predict the relative size of the error, rather than the absolute size. The output of this function is usually expressed in terms of percentages, which eases the process of evaluating the model in some scenarios (MEADE; ISLAM, 2005).
- **Mean Squared Logarithmic Error** (MSLE): The MSLE loss function measures the average squared difference between the logarithms of the predicted and true values. It is often used when the errors are multiplicative rather than additive, and it is particularly useful for tasks where the target values are positive and skewed (GOODFELLOW; BENGIO; COURVILLE, 2016).
- **Huber**: The Huber loss function is a combination of the (MSE) loss function and the (MAE) loss function. It has the properties of both of these loss functions, with the MSE loss for small errors and the MAE loss for large errors. The Huber loss function is less sensitive to outliers in the data than the MSE loss function, but it still penalizes large errors more heavily than the MAE loss function (GOODFELLOW; BENGIO; COURVILLE, 2016).
- **Log-cosH**: The log-cosh loss function is a smooth approximation of the mean absolute error (MAE) loss function. It is defined as the logarithm of the hyperbolic cosine function of the difference between the predicted and true values.This loss function is a smooth approximation of the MAE loss function, and it returns nearly identical results as the MAE loss function for small values, but it is defined

for all values. Besides that, the log-cosh loss function can also be useful in cases where the absolute error is not good enough, and the data could have some outliers, this smooth approximation can help improve the performance of the model (GOODFELLOW; BENGIO; COURVILLE, 2016).

2.4.5 Optimizers

in machine learning, an optimizer is an algorithm used to adjust the parameters of a Neural Network model in order to minimize a given loss function. The optimization process typically involves iteratively adjusting the model's parameters in order to reduce the value of the loss function. However, this process can change depending on the optimizer itself, for example, some optimizers may use gradient descent, while others may use a different method such as conjugate gradient or Newton's method. Essentially, optimizers usually make use of a technique called **Backpropagation**, with this technique the optimizer calculate the gradient of the loss function of the neural network and then utilizes this information to adjust the parameters of the model to reduce the loss function output. Along with backpropagation, the optimizer also comes with a hyperparameter called **Learning Rate** that is also responsible for adjusting the hyperparameters of the model (i.e. how much the weight+biases should change).

There are a lot of optimizers and their usage may vary on the task that the model is trying to perform, some of the optimizers are:

- **Stochastic Gradient Descent** (SGD): is an iterative method that updates model parameters in the direction of the negative gradient of the loss function with respect to the parameters. At each iteration, the algorithm takes a small step (or "update") in the direction of the negative gradient, which is computed using a small random subset of the data called a "batch" or "minibatch". In other words, instead of using the entire dataset to compute the gradients, it uses only a small subset of it. (GOODFELLOW; BENGIO; COURVILLE, 2016)
- **Adagrad**: AdaGrad is an optimization algorithm that adapts the learning rates of all model parameters individually by scaling them inversely proportional to the square root of the sum of all historical squared gradients. The parameters with large partial derivatives of the loss will experience a rapid decrease in their learning rate, while parameters with small partial derivatives will have a relatively small

decrease. This results in greater progress in the directions of parameter space with a gentler slope. In the context of convex optimization, AdaGrad possesses some desirable theoretical properties. However, when training deep neural networks, the accumulation of squared gradients from the beginning of training can lead to an early and excessive decrease in the effective learning rate. (GOODFELLOW; BENGIO; COURVILLE, 2016)

- **Adadelta**: The key idea behind Adadelta is to use the running average of the past gradients, rather than the current gradient, to adjust the learning rate for each parameter. Adadelta makes use of two parameters, an accumulating factor and a decay factor, which are used to control the decay of the running average. One of the main advantages of Adadelta is that it is less sensitive to the choice of the initial learning rate, and it makes fewer assumptions about the data. Adadelta does not need to calculate the gradient of the loss function, this makes it more computationally efficient and reduces the chance of running out of memory. Adadelta is also robust to the problem of the vanishing or exploding gradient, which is when the gradients become too small or too big. (ZEILER, 2012)
- **Root Mean Squared Propagation** (RMSprop) : This approach was based on the Adagrad algorithm, but it uses a moving average of the squared gradient to scale the learning rate of each parameter. The idea behind RMSprop is to use the second moment of the gradients, that is the average of the squared gradients, to scale the learning rate of each parameter. This allows the optimizer to adapt the learning rate of each parameter individually, rather than using a fixed learning rate for all parameters. RMSprop, as SDG, is not largely used anymore, since the Adamax optimizer acts like a direct upgrade of it. The RMSprop could find its use in some scenarios, usually where the magnitude of the gradient is an important factor to the optimization task, but generally speaking, Adamax will end up achieving better results to most of the problems. (GOODFELLOW; BENGIO; COURVILLE, 2016)
- **Adaptive Moment Estimation** (ADAM): It is a variant of the stochastic gradient descent (SGD) algorithm that uses a combination of gradient information from the current and previous iterations to adaptively update the model parameters.The key idea behind Adam is to maintain an exponentially decaying average of the past gradient information, as well as an exponentially decaying average of the past squared gradient information, Adam is one of the best optimizers known to date due to the wide range of applications that he can be used while obtaining good

results, most of the recent optimizers are usually variations of this one. (KINGMA; BA, 2014)

- **Adamax**: This one is a variation of the Adam algorithm that was introduced along with Adam, this one uses something called **the infinity-norm** to scale the learning rates of different parameters. The infinity-norm is a measure of the size of a vector that is used to scale the learning rates of different parameters. This allows the algorithm to assign larger learning rates to parameters that have larger gradients, and smaller learning rates to parameters that have smaller gradients. This helps to prevent the optimizer from getting stuck in flat regions of the loss function. This optimizer is mostly used in problems where the dataset that is being used is too much noisy (i.e. the data appears to be random). (KINGMA; BA, 2014)
- **Nesterov-accelerated Adaptive Moment Estimation** (Nadam) : As Adamax, Nadam is an optimization algorithm that uses characteristics of the Adam algorithm coupled with the **Nesterov Momentum Method**, this momentum method helps to anticipate the direction of the minimum and this can make the optimization process more robust and faster. Like the Adam optimizer, Nadam uses the concept of adaptive learning rates, which means that the algorithm can adapt the learning rate of each parameter individually. Nadam is often used in problems with Long narrow valleys or saddle points (i.e. regions of the loss function that can slow down the optimization process). (DOZAT, 2016)
- **Follow The Regularized Leader** (FTRL): It is a variation of the aforementioned (SGD) algorithm that aims to improve the computational efficiency and scalability of the optimization process. in large-scale problems, the size of the dataset and the number of parameters can be very large, making it computationally expensive to update the parameters for each example. FTRL addresses this issue by using a different approach for updating the parameters. Instead of computing the gradient for each example, FTRL uses an approximation of the gradient based on a combination of the current and past gradients. The learning rate adapts to the curvature of the loss function, becoming smaller as the optimization process progresses, a regularization term is used to prevent overfitting. Essentially, this was an optimization algorithm that was developed by Google in 2013 to solve large-scale online learning problem with a high number of features and parameters. (MCMAHAN et al., 2013)

3 METHODOLOGY

This chapter aims to showcase the methodology that was used to execute this work, the methodology will be divided into technical methods and the criteria for the selection of correlated works.

3.1 Technical Methods

The methodology chapter of the monograph outlined the process that was followed to conduct the research. The first step was to conduct a bibliographic research on existing video compression formats, such as H265 and VP9. This research was carried out to gain an understanding of the execution flow of these formats.

Subsequently, the focus of the research shifted to the AV1 format, as this was the format that would be used for the work. Through this step, the aim was to gain a comprehensive understanding of codecs, with a particular emphasis on the specificities of the AV1 codec.

Next, an exploratory research was conducted using the AV1 reference software. The objective of this step was to correlate the information discovered in the previous step with the information found in the reference software, thereby enabling a deeper understanding of the codec's execution flow.

Afterwards, an experimental research was carried out where the reference software was modified to include execution logs. The goal of this step was to identify elements in the reference software that had been identified in the previous steps. With the modification in place, the software was run to perform the encoding of several recommended video sequences. The primary objective of this step was to identify the variables that would be necessary for training a Neural Network, thus enabling the creation of a relevant dataset for training.

With a thorough understanding of the AV1 operation and its reference software, a dataset was prepared for training the Neural Network. For this step, the aim was to obtain a large number of recommended video sequence samples to generate a sufficiently diverse dataset.

A study on the functioning of Neural Network architectures was conducted so that the Neural Network could be properly trained and modeled. At this point, the aim was to achieve a satisfactory level of proficiency with Artificial Neural Networks so that they could be used.

After the study, a total of 26 Artificial Neural Network models were created, trained, and executed in order to find the most appropriate model. Each one of the Neural Network models were tested on a large number of samples, after this, they were compared to each other to determine the final, optimal model for tackling the proposed task.

3.2 Correlated works criteria

For the selection of works to serve as a basis for the objective of this manuscript, a bibliographical research was carried out in the IEEE Xplore database. The first works that that coupled AV1 with ML were searched through the following search strings: ("All Metadata":AV1) AND ("All Metadata":Machine Learning) OR ("All Metadata":AV1) AND ("All Metadata":Neural Network) OR ("All Metadata":AV1) AND ("All Metadata":Deep Learning) The selected keyword was AV1 and a period limit of 2015 to 2022 was select.

The search returned a total of 15 results, including the works of Chen et al. (2020), Su et al. (2019) and Kim et al. (2019) that were described at the previous section. After that, due to the low portion of related works, a new search was done, but this time, HEVC would be searched instead of AV1. the keyword was HEVC, the time period was from 2015 to 2022 and the produced search string was: ("All Metadata":HEVC) AND ("All Metadata":Machine Learning) OR ("All Metadata":HEVC) AND ("All Metadata":Neural Networks) OR ("All Metadata":HEVC) AND ("All Metadata":Convolutional Neural Networks) The string resulted on 454 works, on which 2 of those were selected, the work of Song et al. (2017) and Ma et al. (2019)

A total of 5 works were selected, the work of Chen et al. (2020), Su et al. (2019), Kim et al. (2019), Song et al. (2017) and Ma et al. (2019)

4 VIDEO CODING SYNTAX ELEMENTS & ARITHMETIC CODING

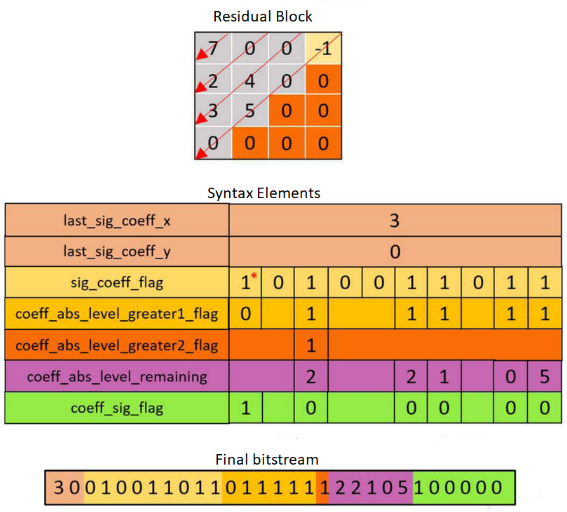
This chapter intends to give a broader view of what syntax elements are and how they are generated, focusing especially on the Residual SEs.

4.1 Video Coding Syntax Elements

One could define a **Syntax Element** (SE) as segments of information that were gathered in the stages prior to the entropy stage e.g., Motion Vector, Quantization parameters. (RAMOS et al., 2020). For instance, let us take a look at one of the HEVC SEs, called “Skip Mode”, it is an SE responsible to inform if there was motion in a given frame without any significant change in illumination (RAMACHANDRAN, 2018). Thus, this information will be saved in the bitstream as a flag, so that when the decoder sees that flag (i.e., an element that will encapsulate this information) he will be able to gather this information and make the appropriate decision.

Essentially, SEs are a way to describe characteristics or chunks of information in a summarized way, as they also serve as the input for the entropy stage. Exemplifying this concept, why should one describes an 8×8 block of pixels by each one of them when all of its values have the same green color? Instead, one could create an SE that will simply say whether this block is entirely green or not. Therefore, it is far easier and compact to try to describe the block in this fashion.

Figure 27 – Syntax elements of HEVC



last_sig_coeff_x	3									
last_sig_coeff_y	0									
sig_coeff_flag	1*	0	1	0	0	1	1	0	1	1
coeff_abs_level_greater1_flag	0		1			1	1		1	1
coeff_abs_level_greater2_flag			1							
coeff_abs_level_remaining			2			2	1		0	5
coeff_sig_flag	1		0			0	0		0	0

3 0 0 1 0 0 1 1 0 1 1 0 1 1 1 1 1 1 2 2 1 0 5 1 0 0 0 0 0

Source: Adapted from Ramos et al. (2020).

We will go through a more "practical" example by utilizing Figure 27 as reference. By looking at the figure we can notice that the block to be "interpreted" is an HEVC residual one that came after the quantization step. Below it, we see the SEs that will be generated from this residual block, and below the SEs we see the final bitstream. Now let us cover step by step of this process.

Being a residual block of transformed coefficients, it is already organized in a way that the highest valued coefficients are always in the upper-left corner while the lower or non-significant (i.e., zeroed) coefficients always stay at the bottom-right of the block. Something important to notice is that, in HEVC we read the information of a transform block in a zigzag fashion, starting from the bottom-right zero until the top-left seven.

Since SEs work as flags or elements capable of describing certain characteristics of a block, we should investigate the meaning of each one of those SEs that are generated within the block at Figure 28.

- **last_sig_coeff_x** and **last_sig_coeff_y** (LAST): Represents the position for the axis x and y of the last significant coefficient in a given group of residues (-1 for our example) (RAMOS et al., 2020).
- **sig_coeff_flag** (SIG): Determines whether a coefficient is significant or not (i.e., if it is different from 0). If the coefficient is represented as 1, then its absolute value is different from 0; if it is represented as 1* it is considered an inferred coefficient, and therefore excluded from the bitstream. Otherwise it is considered a non-significant coefficient and marked as 0. (RAMOS et al., 2020). It is important to note that this SE only consider coefficients that begin at the zone marked by LAST.
- **coeff_abs_level_greater1_flag** (COEFF1): checks if the significant coefficients' absolute values are greater than one, if it is, coeff1 marks it as 1, otherwise 0.
- **coeff_abs_level_greater2_flag** (COEFF2): for a single coefficient before the LAST, whether it has an absolute value greater than two or not. (RAMOS et al., 2020)
- **coeff_abs_level_remaining** (REM): all significant values undergo a decrement by a base value, depending on whether or not the COEFF1 and the COEFF2 exist for that same transformed residue. Therefore, the difference is, the REM (RAMOS et al., 2020). For our particular case, if COEFF1 is marked as 1, then the value is decremented by 2, and if COEFF2 is also marked as 1, then the value is again decremented, but this time by 1.

- **coeff_sign_flag** (SIGN): the signal (i.e., positive or negative) for each significant coefficient before the LAST, 1 for negative and 0 to positive.

Finally, after the evaluation of all the coefficients, they are all organized in the sequence as presented in the figure, and later on they will be sent to the entropy encoding stage.

4.2 AV1 Residual Syntax Elements

Having in mind what SEs are, Residual Syntax Elements (RSE) are the ones generated after the quantization step, within the residual block, as depicted in Figure 28 for the HEVC standard. According to (RAMOS et al., 2020), RSE can compose up to 94% of total bins generations in HEVC, which is a substantial amount. Based on this data, one can also estimate that RSE are potentially one of the major contributors for the AV1 entropy encoding stage, which make them an essential subject of further study.

With that in mind, we should take a look at the RSEs of the AV1 standard. A lot of what we have seem from previous sections will hold true for AV1, i.e., RSEs still come after the quantization step, and still are the one of the major contributors to the input data of the entropy stage (HAN et al., 2021). The AV1, differently from HEVC, starts the codification process by transforming the 2D residual block into a 1D residual array and decomposes the array into six main elements, then being:

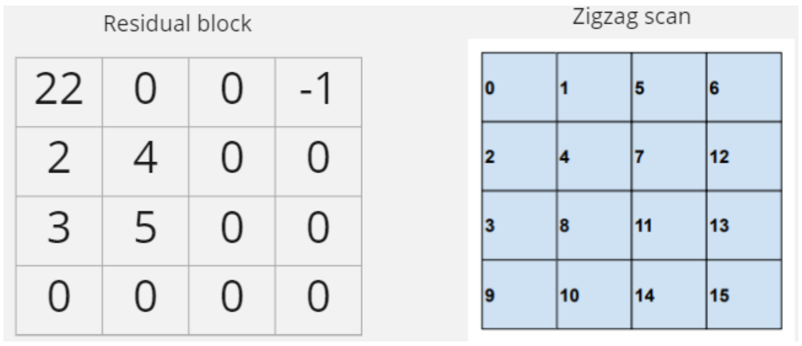
- **eob**: This element marks the index of the last non-zero coefficient plus one in the array (RIVAZ; HAUGHTON, 2018)
- **dc_sign**: it marks the signal (i.e., positive or negative) of the leftmost element in the array of coefficients. If the coefficient is negative, dc_sign will be flagged as 1, otherwise 0 (RIVAZ; HAUGHTON, 2018).
- **sign_bit**: This flag will come along each one of the significant transform coefficients (i.e., not zeroed ones) and it is responsible for carrying the signal of the coefficients, following the same pattern as dc_sign. It will be flagged as 1 if the coefficient is negative, and 0 otherwise. sign_bit will not flag the signal of the leftmost coefficient, since dc_sign would have already done that (HAN et al., 2021).
- **base_range** (BR): The symbol contains four possible outcomes $\{0, 1, 2, > 2\}$, which are the absolute values of the quantized transform coefficient, if the value of the coefficient is greater than two, it will also need a low_range flag to save its

value (HAN et al., 2021).

- **low_range** (LR): as we have seen in base_range, low_range will also contain four possible outcomes $\{0, 1, 2, > 2\}$ that correspond to the residual value over the previous element upper limit. Furthermore, each time the element goes to > 2, it will be decremented by three and will generate another low_range flag, repeating this process up to four times, effectively covering a range of 3,14. If the coefficient is bigger than 14, it will need a high_range element to save its total value Han et al. (2021).
- **high_range** (HR): high_range is a symbol that appears if and only if the quantized coefficient is in the range of [15, 2^{15}). it also corresponds to the residual value over the previous element upper limit (HAN et al., 2021).

To develop a better understanding on the subject, we are going to walk through an example, suchlike we did on the previous section for the HEVC residual block, the same is going to be done here, but this time, for the AV1 residual block. First, let us consider the 4×4 residual block depicted in Figure 28.

Figure 28 – Residual Block & Zigzag scan



Residual block

22	0	0	-1
2	4	0	0
3	5	0	0
0	0	0	0

Zigzag scan

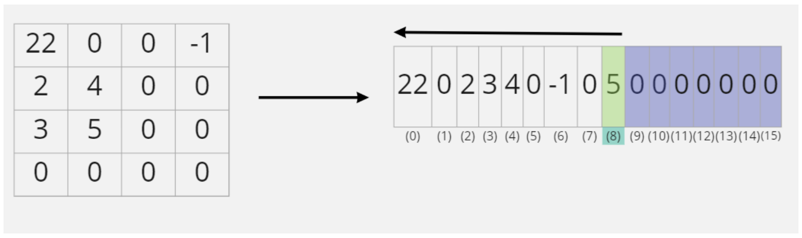
0	1	5	6
2	4	7	12
3	8	11	13
9	10	14	15

Source: Adapted from Han et al. (2021).

The first step that we have to take is to transform this whole residual block into an 1D Array. The way that we are going to do it may vary depending on which transform kernel was selected by AV1. For simplicity's sake, let us consider that the kernel was a DCT, because of that, a zigzag scan will be performed on the residual block, as showed in Figure 28, the numbers on each one of the blocks are representatives of the sequence that the block is being read scanned.

After the scan, a 1D array will be generated, where the generated array is displayed at Figure 29. One may notice that the last non zero coefficient will be the number five, at index 8, as we can see on Figure 29. Besides that, the coefficients will be processed in a right-left fashion, starting from the last non-zero coefficient until the DC_Coefficient (i.e., 22) at index 0.

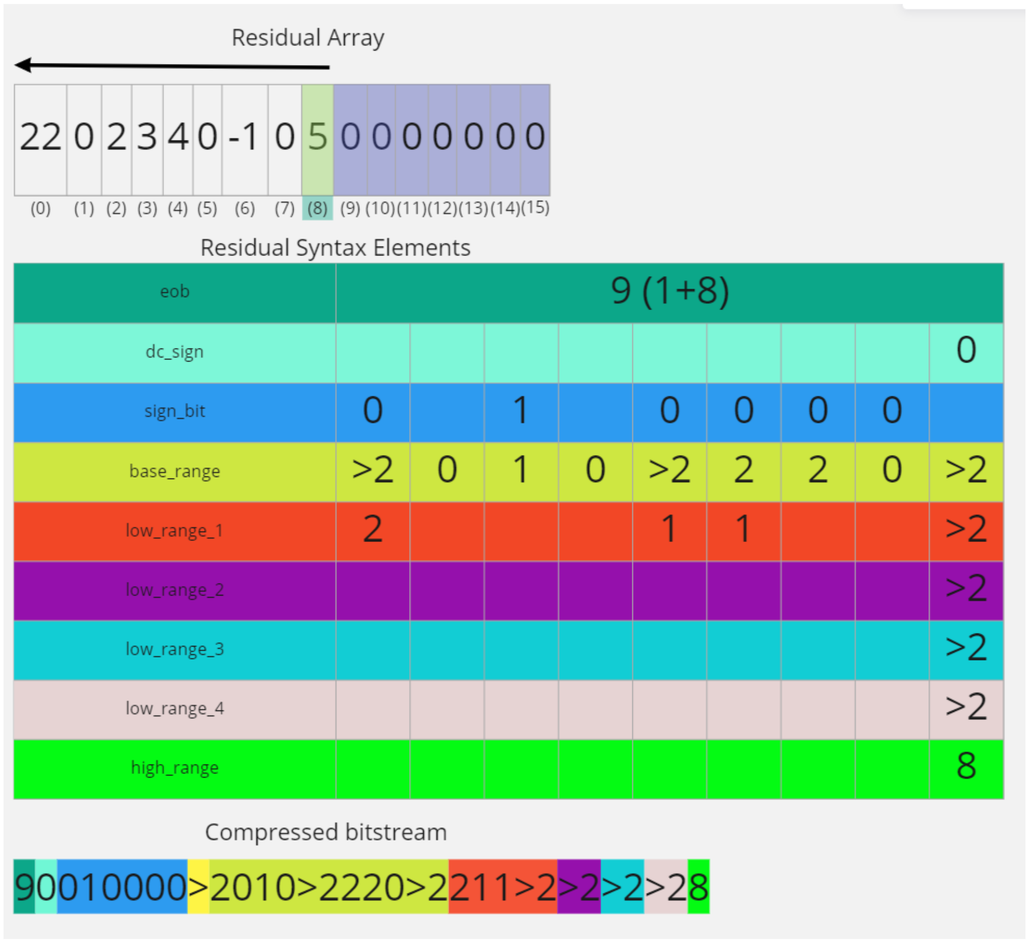
Figure 29 – Residual Matrix & Residual Array



Source: Author (2022).

After obtaining the 1D array, the next step is to properly encode each-one of the elements into the bitstream that will be fed in to the entropy coding step. Figure 30 displays all the RSEs, along with the compressed bitstream at the end.

Figure 30 – RSEs & Compressed bitstream



eob	9 (1+8)								
dc_sign									0
sign_bit	0		1		0	0	0	0	
base_range	>2	0	1	0	>2	2	2	0	>2
low_range_1	2				1	1			>2
low_range_2									>2
low_range_3									>2
low_range_4									>2
high_range									8

Source: Author (2022).

Some worth noting points about Figure 30 will be brought to light:

1. **eob** is 9 because the index of the last non-zero coefficient is 8, add 1 to it, and you will get the displayed result.

2. **base_range**, **low_range(1-4)** and **high_range** consider only the absolute value of the coefficient, leaving for the **sign_bit** or **dc_sign** to dictate its signal.
3. Only **dc_sign**, **base_range** and **low_range(1-4)** are context modeled. In consequence, only those 3 are going to be suitable for an ML Approach to estimate their probabilities.
4. **sign_bit** does not flag anything if the analyzed coefficient is 0.

4.3 Arithmetic coding

Arithmetic coding is a method for lossless data compression that represents a string of symbols as a single fractional value within a given range. Unlike other compression methods such as the previously discussed Huffman coding, which rely on fixed-length codes for each symbol, arithmetic coding assigns a unique fractional value to each symbol based on its probability of occurrence within the source data (SALOMON, 2004)

To properly define how it is possible to define a range for a symbol according to its probabilities using arithmetic coding, we must understand the method's characteristics. Arithmetic coding starts by defining an initial interval of [0, 1], and as symbols are encoded, the interval is gradually refined and subdivided based on the probabilities of each symbol. Each symbol is encoded as a fraction of the current interval, and the range of the interval represents the probability of the symbol. When encoding symbols, arithmetic coding uses a cumulative probability distribution function to determine the subinterval associated with each symbol. Specifically, the interval is split at the point corresponding to the cumulative probability of the symbol, and the subinterval representing the symbol is the one that contains the current interval. The main objective of this whole process is to get an interval between two fractional numbers that could be expressed with fewer bits as possible. The more that we subdivide each interval, more bits that are going to be required to express a value between this fractional interval. This final value is what is going to be effectively sent to the bitstream. (RAMACHANDRAN, 2018)

To properly define this process, we need to consider two equations:

$$Range_j = Range_{j-1} * p(s_j) \tag{2}$$

Where:

- $Range_j$ is the range of the interval for the current symbol j
- $Range_{j-1}$ is the range of the interval for the previous symbol j-1
- $p(s_j)$ is the probability of the current symbol j

$$Low_j = Lower\_bound + (Range_j * CDF) \tag{3}$$

Where:

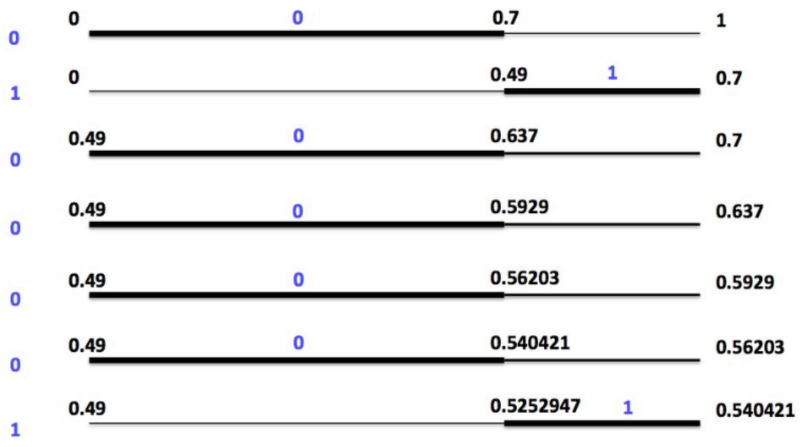
- Low_j is the lower bound of the interval for the current symbol j
- Lower_bound is the lower bound of the interval for the current symbol
- Range_j is the range of the interval for the current symbol j
- CDF: is the cumulative frequency of all the symbols that came before the current symbol

It is important to note that the range of the interval for a symbol is dependent on the probability of the current symbol, as well as the range of the interval of the previous symbol. As the probability of the current symbol is more probable, for instance, the range of the interval is updated impacting in less fractional values, and fewer bits are required to encode the decimal places into the final bitstream. However, it should be noted that there is a minimum size required to encode a sequence without losing its information. The goal of proper estimation of probabilities in arithmetic coding is to approach as closely as possible the ideal entropy proposed by Shannon (1948).

In the context of video coding, it is not feasible to have the probability of each symbol prior to the encoding of the whole sequence. To overcome this limitation, context modeling is employed, which involves considering previously encoded symbols and dynamically updating the probabilities. The initial probabilities are usually obtained from a static table of probabilistic models chosen prior to the encoding of any symbol.

To conclude this section, Figure 31 will demonstrate an example of arithmetic encoding with static probabilities. Consider that the following sequence will be encoded: [0,1,0,0,0,0,1] and the probabilities of occurrence are 0.7 to the symbol ’0’, and 0.3 to the symbol ’1’. The first step that we need to do is set an interval of [0,1]. After that, we can start the process of encoding the sequence. The first symbol to be encoded is a 0, with a probability of occurrence of 0.7, which means that we have to adjust our upper to 0.7 and our new interval is [0,0.7]. The next symbol is a 1, this time the lower boundary has to be adjusted, so the new value interval is set to [0.49,0.7]. After this, the next one is a 0, which incurs in the upper boundary being changed again, and our new interval becomes [0.49,0.637]. The fourth symbol is also a 0, meaning that our upper boundary will be updated again, after doing the calculations, our new interval is [0.49,0.56203]. As the previous symbol, the sixth symbol is also a 0, and our new interval becomes [0.49,0.54021]. Finally, the last encoded symbol is a 1, and we have to change our lower boundary, then generating our final interval [0.5252947,054021]. To deviate the

Figure 31 – Arithmetic Coding - Example



Source: Ramachandran (2018).

fraction of this interval that is going to be sent to the bitstream we just need to pick any number between the final interval, consider that the selected value was 0.54, converting it to binary would result in 0.10001, we then send the value "10001" to the bitstream. With this process we just managed to encode seven bits, by using five.

4.4 Related Works

This section will describe five recent works that tackle the video encoding processing by using a Neural Network approach. The work of Ma et al. (2019) and Song et al. (2017) make use of CNNs to estimate the probability of Syntax Elements at respectively inter and intra-prediction modes for the HEVC standard. The work of Kim et al. (2019) is related to the AV1 codec, and proposes a decision tree to decide whether or not advanced interprediction modes should be tested. The work of Su et al. (2019) develops a neural network to accelerate the transform size and kernel search stage for AV1. Finally, the last work will be from Chen et al. (2020) and it covers an Artificial Neural Network model used to replace the lookup table scheme that AV1 uses to estimate the probability of SEs.

4.4.1 Work of Ma *et al*

The work of Ma et al. (2019) covers a Convolutional Neural Network (CNN) to help in the entropy stage of inter-prediction SEs for the HEVC standard. It is stated that the manually designed binarization and context models are not flexible to estimate the probability of the syntax elements. A CNN is used to estimate the probability of the SE, and then the estimated probabilities of them are fed into the arithmetic coding step to fulfill the entropy coding stage. The syntax elements covered in this work were: merge flag, merge index, reference index, motion vector difference and motion vector prediction index. The analysis was made under low-delay P (LDP) setting and tested for 16 different video sequences that were classified in five different Quantization Parameter (QP) groups. On average, this approach managed to obtain a 0.3% reduction in BD-rate for each color channel.

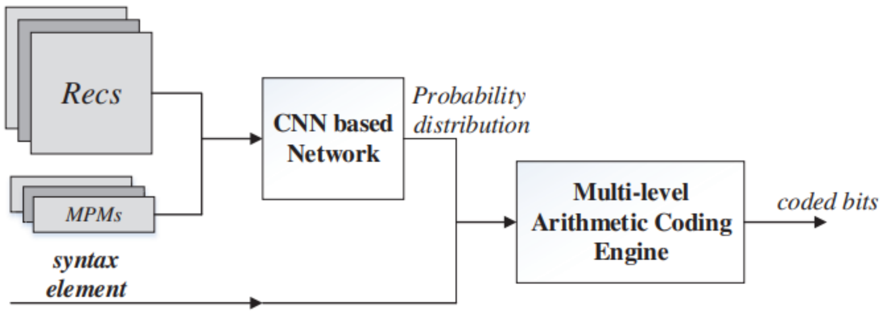
4.4.2 Work of Song *et al*

The work (SONG et al., 2017) presents a CNN coupled with a multi-level arithmetic codec to replace the CABAC from HEVC. The CNN replaces the handcrafted context models to perform the probability estimation of intra-prediciton modes that serve as input for the multi-level arithmetic codec. Figure 32 shows the aforementioned flow of execution, where intra-prediction Most Probable Modes (MPMs) along with *Recs* (reconstructed blocks made by prediction) are used as input for the CNN, then the SEs along with the probability distribution are used as input for the developed codec. This work find its importance in the manuscript by the fact that an ML approach was used in the intra-prediction modes to properly estimate the probability of Syntax Elements for the HEVC standard This approach managed to obtain gains to up to 9.9% bit rate savings when comparing to the default arithmetic encoder of HEVC (i.e., CABAC).

4.4.3 Work of Kim *et al*

The work of (KIM et al., 2019) proposes a method based on decision trees to selectively decide whether to test all advanced inter prediction modes (CRFPM) or not. This approach came from the claim that advanced inter prediction modes can be

Figure 32 – Scheme of the CNN-based arithmetic coding



Source: Song et al. (2017).

beneficial, but they are very costly in terms of computational resources, which end up limiting the range of software applications that could make use of it. This approach was tested on eight video sequences, where two of those were high resolution sequences (1920×1080) and the other six were low resolution (640×360). The experimental results of this work show that the encoding time can be reduced on average by 43.4%, with negligible impact on coding efficiency. This huge gain is achieved because not all sequences benefit from all CRFPM modes, and the decision that the tree performs is whether to skip these modes or not.

4.4.4 Work of Su *et al*

The work of (SU et al., 2019) propose a ML-based approach to accelerate the transform size and kernel search stage for the AV1 standard. The ML models use input features extracted from the prediction residue block such as standard deviation, correlation, and energy distribution. The output of the model indicates the estimated likelihood of which transform size and kernel would be selected as the optimal choice. Based on the ML models, the encoder can prune out the transform size and kernel candidates that are unlikely to be selected and save unnecessary computation to process their rate-distortion cost. the approach was tested on 110 video sequences, of which 40 of those were low resolution CIF (288p), 30 corresponded to a medium resolution (480p) and finally, the remaining 40 sequences are high resolution, ranging from (720p) to (1080p). The experimental results of this paper show that the encoding time was reduced by up to 38.2% at low resolution sequences, with a negligible loss in quality (0.13%).

4.4.5 Work of Chen *et al*

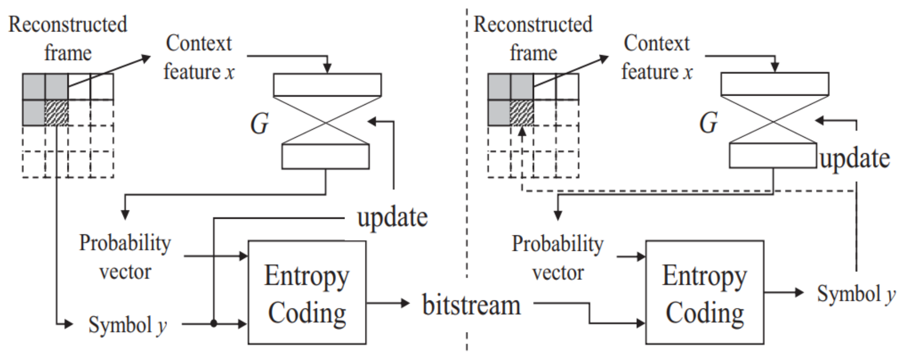
This paper proposes an ML based scheme that will perform a more accurate symbol probability prediction than the context modeling stage in the AV1 standard. This approach is implemented in the entropy coding stage for intra prediction modes only. The proposed ML model was a FFNN with a single hidden layer with sixteen neurons, and a ReLU function, whereas the output layer is composed of one neuron with a softmax activation function. The inputs of this neural network are context features, from the context modeling stage.

The choice of using an FFNN architecture instead of a CNN is justified by the fact that FFNN are much simpler than CNN's, and since the developed model will be competing with a simple lookup table, it is vital that the developed model is as quick as possible in its tasks. One may notice that the lookup table on AV1 is adaptive and updated based on the coded symbol value in each coding/decoding operation. Hence, to maintain this constant update on the developed model, after each symbol, the weights of the model are updated by the following formula 4:

$$W = W - \mu \Delta_W L_{ce}(G(x;W), y) \tag{4}$$

where **W** are the weights of the developed model, **G** is the model itself, μ is the update rate on which the weights are updated, x are the input entropy context features, y is the coded symbol, and finally, L_{ce} denotes the cross-entropy function, which is a function used to calculate the losses (i.e., how far the model was from the correct answer). Essentially, this formula is mimicking the updates on the lookup table of AV1 by enabling the model to update its weights after each symbol processing. Figure 33 depicts the proposed workflow.

Figure 33 – Proposed coding pipeline



Source: Chen et al. (2020)

The proposed model was tested on three different resolution groups, a low resolution group, composed of 40 video sequences with a 352×288 (288P) resolution, a medium resolution group, covering about 30 video sequences with a resolution of 852×480 (480P) and finally, a high resolution group, composed of about 40 sequences with resolutions ranging from 1280×720 (720P) to 1920×1080 (1080P). Since this work focused on intra prediction modes, all the video sequences were encoded with the option of “intra-only” that the software of reference disposes.

Table 3 – PSNR BD-rate gains compared to the AV1 baseline

		Low resolution	Medium resolution	High resolution
Y	W/O update	0.472%	0.445%	0.413%
	W/ update	0.495%	0.493%	0.503%
Y+UV	W/O +update	0.628%	0.512%	0.455%
	W/ update	0.698%	0.672%	0.686%

Source: Chen et al. (2020)

The results were compared with the base solution and were gathered on two different conditions, the first one is a case where the weights of the model are static, and the other case, a scenario in which the weights are updated after each encoded symbol. The aforementioned comparison was made at two different color samplings (4:0:0(luma-only) and 4:2:2). Table 3 display the results obtained for this approach It is important to note that these results are obtained over the whole coding process, and not just the entropy stage. If we made a comparison with only the entropy stage, one would get around 7% gains over the standard lookup table approach. This work act as a solid foundation for this manuscript, since the proposed approach will be quite similar to the one that this paper proposes.

Table 4 – Summary of Correlated Works

Work	*Architecture based*	*Focus*	*Codec*	*Year*	*Gain*
Ma *et al*	CNN	Inter-SE	HEVC	2019	Compression
Song *et al*	CNN	Intra-SE	HEVC	2017	Compression
Kim *et al*	Decision Tree	Prediction	AV1	2019	Time
Su *et al*	FFNN	Transform	AV1	2019	Time
Chen *et al*	FFNN	Intra-SE	AV1	2020	Compression

Source: Author (2022).

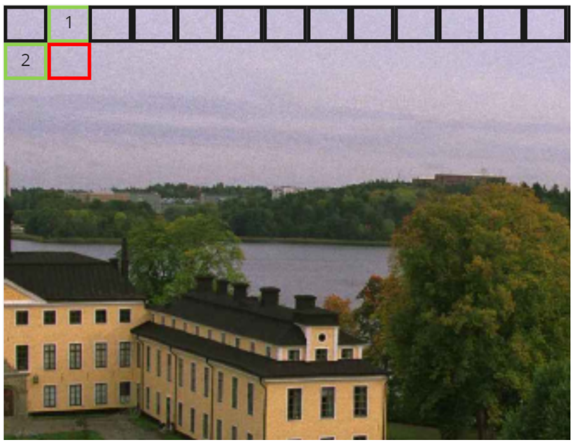
5 DEVELOPMENT OF THE ANN MODEL

This chapter aims to provide an overview of the stages of development of the proposed ANN model that was developed to estimate the probability of the AV1 Residual Syntax elements. Specifically, will be discussed the development of the dataset, the selection of the optimizer, the choice of the loss function, the determination of an appropriate size for the neural network, and an evaluation of the final model's performance.

5.1 Initial Concepts

Before delving into the development of the dataset, it is important to consider what specific information of the AV1 codec is going to be examined. As the objective of this manuscript is to create a Neural Network to estimate the probabilities of an RSE in the AV1 codec, it is crucial to first understand certain properties of these probabilities. For instance, for an RSE to be properly estimated it has to be context modeled, a context modeled RSE is an element in which the probabilities of its neighbors directly affect their own probabilities, and their own probabilities are dynamically altered throughout the encoding process, to illustrate this concept Figure 34 presents the idea. In the

Figure 34 – Context modeling



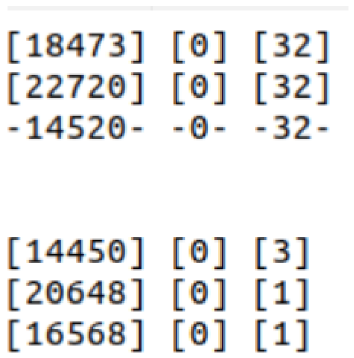
Source: Adapted from (RAMACHANDRAN, 2018).

Figure, the red block that is going to be encoded will be context modeled by both of its neighbors (block 1 and 2). It is important to note that the entropy coding can only use the probabilities of blocks that were already encoded, and since the encoding of each block always happens in a left-right top-down fashion, the only blocks that could be considered as “valid” were 1 and 2 (RAMACHANDRAN, 2018).

Another important aspect to consider is the manner in which the probabilities of each element are stored and interpreted. In the case of the AV1 codec, the probabilities are stored as CDFs, which are integers that can range from [0 to 2^{15}]. To further illustrate this concept, consider the following scenario: a DC_Sign element is being encoded. Due to the nature of this RSE it can only assume a symbol of 1 or 0. If the CDF of this element is, for example, 20157, the chance of it being a 1 is calculated as 32768 minus 20157, which is 12611. This can be interpreted as a 38% chance of the element being a 1. Because of this, we can assume that the chance of the element being a 0 is 62%. The idea of the probability being stored in a CDF is that multiple probabilities can be stored in a single integer, rather than assigning the probability of each symbol to a different variable.

Before proceeding with the development of the dataset, it is important to understand one final property of the CDFs in the software of reference. The CDFs in the AV1 codec have static probabilities that are chosen prior to the encoding of the first block of every frame. The AV1 codec has a Look-up table where it selects a set of pre-determined probabilities that are used during the encoding process, where these probabilities were chosen based on the analysis of the video sequence and the spatial and temporal correlation of the pixels, while encoding each block, these static CDFs are then dynamically altered as the blocks are encoded. This look-up table is a [2][3][3] matrix in which the X axis represents the type of plane being encoded (i.e., Luma or Chroma) and the Y axis represents each CDF according to its context prior to the encoding of the actual block, last but not least, the Z Axis stores the actual CDF along with some informations that are not relevant for this work. Figure 35 shows an example of this table.

Figure 35 – DC_Sign Probabilities - Look-up table



```
[18473] [0] [32]
[22720] [0] [32]
-14520- -0- -32-


[14450] [0] [3]
[20648] [0] [1]
[16568] [0] [1]
```

Source: Author (2023).

After considering all of the aforementioned topics, it was necessary to select an appropriate RSE to build a Neural Network to estimate its probability. For this work, the selected RSE was the DC_Sign. The reason for choosing it over the others is mainly due to its simplicity and the fact that it was the RSE that was constantly being researched during the development of this manuscript. The DC_Sign element saves the signal of the DC_Coefficient (typically the highest coefficient that is located at the top-left of the transform table). It is modeled by context because the probabilities of previously encoded DC_Signs directly affect the current on (i.e., choosing which one of the lines of the Figure 35 should be selected).

5.2 Dataset development

After selecting an RSE, the next step was to construct the dataset. To do this, it was necessary to determine which type of information would be useful to include as features in the dataset. With this in mind, seven variables were chosen to be used as features. The seven variables will be briefly described below.

- plane_type: Plane type indicates if the block that is being encoded is from a Luma or Chroma plane, 0 if it is Chroma and 1 for luma
- dc_sign_ctx: This variable is responsible for telling the context of the dc_sign element (i.e., choosing which one of the three lines it should choose based on the information of its neighbors)
- initial_cdf: As the name indicates, initial_cdf is the CDF that is conceived in the look-up table prior to the probability estimation that is done by AV1
- symbol: Symbol of the DC_sign that is being encoded, could be 0 or 1. 0 if the signal that he's encoding is positive, and 1 for negative
- nsymbols: Number of symbols that the DC_Sign element can have.
- rate: represents the rate at which the CDF of the symbol is updated.
- new_cdf: represents the updated probability of the DC_Sign Syntax element after the adjustment made by AV1.

New_cdf is what is going to be used as a target for the training of the Neural Network Model, since this is actually the probability that the model is trying to estimate. All of these features were gathered in the encodetxb.c and bitwriter.h files that are within the source code of the AV1.

Considering these informations, the next step was to choose the video sequences that would be used to build the dataset, the selected video sequences were taken from the work of Daede, Norkin e Brailovskiy (2019) [1], a collection of videos that are commonly used in video-related researches. The video sequences are shown in Table 5, All of the video sequences were encoded using the AV1 software of reference Libaom [2].

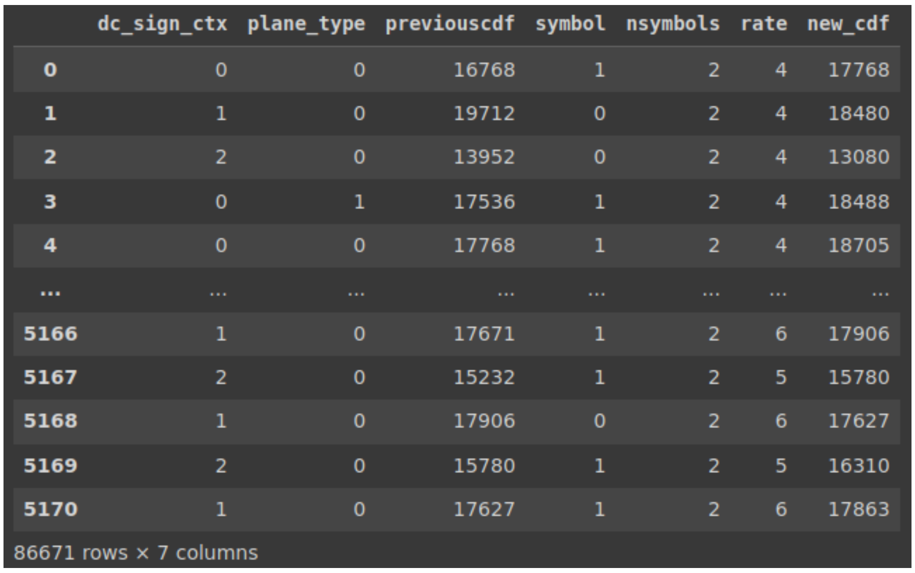
Table 5 – Video sequences

Name	*Resolution*	*FPS*	*Number of acquired samples*
Netflix_FoodMarket	1920x1080	60	9262
*Netflix_Boat_*	1920x1080	60	20933
*DOTA2_*	1290x1080	60	23679
*MINECRAFT_*	1290x1080	60	7206
Netflix_DrivingPOV	1280x720	120	20420
boat_hdr_amazon	1280x720	120	5171

Source: Author (2023).

All of the video sequences were encoded with a CQ level of 55, all of them had 8 bit-depth and a sub-sampling of 4:2:0. After merging all the sequences in one dataset, a dataset composed of 86671 entries was created, Figure 36 shows 10 entries of the dataset, aswell as its actual size at the end of the figure.

Figure 36 – Initial dataset



```
      dc_sign_ctx  plane_type  previouscdf  symbol  nsymbols  rate  new_cdf
0               0           0        16768       1         2     4    17768
1               1           0        19712       0         2     4    18480
2               2           0        13952       0         2     4    13080
3               0           1        17536       1         2     4    18488
4               0           0        17768       1         2     4    18705
...           ...         ...          ...     ...       ...   ...      ...
5166            1           0        17671       1         2     6    17906
5167            2           0        15232       1         2     5    15780
5168            1           0        17906       0         2     6    17627
5169            2           0        15780       1         2     5    16310
5170            1           0        17627       1         2     6    17863
86671 rows × 7 columns
```

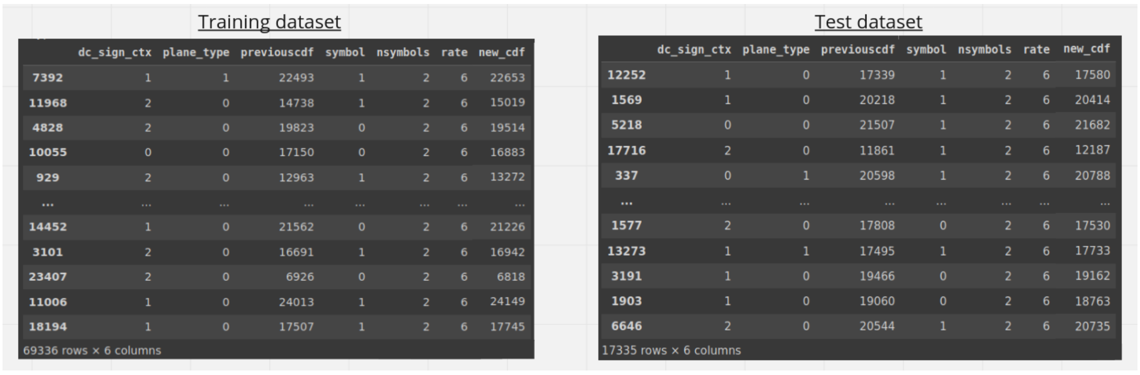
Source: Author (2023).

With the dataset at hand, the next step was to divide it into a training set and a testing set. The dataset was divided in a proportion of 80/20, resulting in a training set

[1]https://datatracker.ietf.org/doc/html/draft-ietf-netvc-testing-07autoid-22
[2]https://aomedia.googlesource.com/aom/

of 69336 samples and a testing set of 17335 samples. Figure 37 illustrates both of the datasets.

Figure 37 – Training & Testing dataset



Training dataset

	dc_sign_ctx	plane_type	previouscdf	symbol	nsymbols	rate	new_cdf
7392	1	1	22493	1	2	6	22653
11968	2	0	14738	1	2	6	15019
4828	2	0	19823	0	2	6	19514
10055	0	0	17150	0	2	6	16883
929	2	0	12963	1	2	6	13272
...	...	...	...	...	...	...	...
14452	1	0	21562	0	2	6	21226
3101	2	0	16691	1	2	6	16942
23407	2	0	6926	0	2	6	6818
11006	1	0	24013	1	2	6	24149
18194	1	0	17507	1	2	6	17745

69336 rows × 6 columns

Test dataset

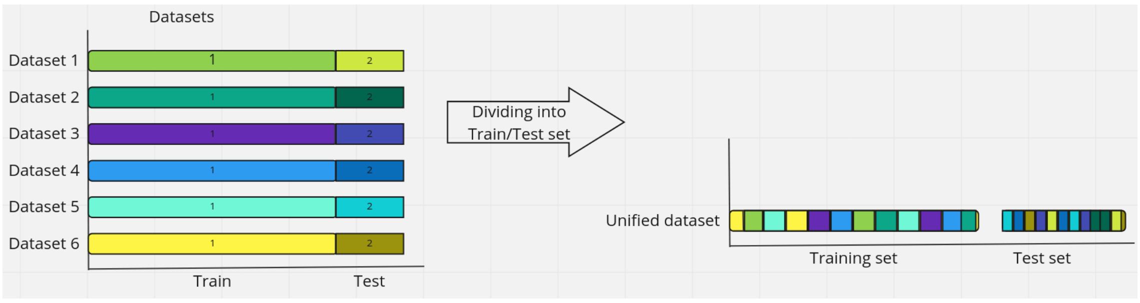
	dc_sign_ctx	plane_type	previouscdf	symbol	nsymbols	rate	new_cdf
12252	1	0	17339	1	2	6	17580
1569	1	0	20218	1	2	6	20414
5218	0	0	21507	1	2	6	21682
17716	2	0	11861	1	2	6	12187
337	0	1	20598	1	2	6	20788
...	...	...	...	...	...	...	...
1577	2	0	17808	0	2	6	17530
13273	1	1	17495	1	2	6	17733
3191	1	0	19466	0	2	6	19162
1903	1	0	19060	0	2	6	18763
6646	2	0	20544	1	2	6	20735

17335 rows × 6 columns

Source: Author (2023).

The division of the dataset into a training and testing set is crucial for the model's performance. The training set is used to specifically train the model, while the testing set is used to validate the model's accuracy. If the model were tested with the same dataset that was used to train it, it wouldn't be able to properly evaluate it. This is because the model would already know the answers, and it would not be able to perform well on new data that it had never seen before. This division is a crucial step in the model development process as it helps to ensure that the model has good generalization capabilities and is not overfitting the training data. With this in mind, the division of the dataset was performed according to Figure 38.

Figure 38 – Split into train/test set



Source: Author (2023).

This action ensures that, even though the same video sequences are used, different parts of them are used to evaluate the model. Additionally, for the training phase, the dataset is organized in a way that parts of video 1 are "mixed" with parts of video 2. This mix in the order also helps with the model's generalization and evaluation.

5.3 Metrics

To effectively evaluate the generated models, it is necessary to establish three different metrics for measuring their performance when compared to one another. Since the objective is to predict an integer value (i.e., CDF), it is necessary to establish metrics that could evaluate a linear regression model. The first established metric was called "Precision". The main objective of this metric is to determinate models that may not be exceptionally good at predicting the precise values, but instead are more precise to outlier values. This characteristic is achieved by considering the prediction to be correct if it falls within a 100-unit range of the actual value, for example, if the predicted value was 17394 and the actual value was 17494 it would count as a hit, it would also count as a hit if the predicted value was 17594. Equation 5 describes this process.

$$Precision(\%) = \frac{\sum_{i=1}^{n} [|p_i - a_i| \leq 100]}{n} \tag{5}$$

Where:

- p_i is the prediction for the i^{th} instance.
- a_i is the actual value for the i^{th} instance.
- n is the total number of instances.
- $[\cdots]$ represents an indicator function that returns 1 if the condition inside the brackets is true and 0 otherwise.

The chosen value of 100 was deemed acceptable as a margin of error due to the fact that a difference of 100 only results in a deviation of 0.3% in the actual probability, which is considered a neglible difference in terms of losses.

The next metric that will be used is the previously mentioned **Mean Absolute Error** (MAE). This metric is useful in determining the accuracy of the model. It works by calculating the average absolute difference between the predicted values and the actual values. This provides a measure of how close the predictions are to the true values. The lower this metric is, the more precise the model is at predicting the exact value that was supposed to be predicted. Equation 6 highlight the aforementioned process.

$$L_{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{6}$$

Where:

- L_{MAE} is the MAE loss
- n is the number of samples
- y_i is the true value for sample i
- $\hat{y}_i$ is the predicted value for sample i

Finally, the last metric that needs to be introduced is called "Computational Cost." Since the models have a "static" architecture, it is possible to estimate how costly (in terms of computational resources) each model is by assigning a weight to the multiplications and sums utilized by the model. For example, we could assign a cost of 3 to each multiplication that needs to be performed by the model for a prediction, while sums or subtractions could be assigned a cost of 1. With this in mind, we can consider the final architecture of the model and have a rough estimate of its computational resources usage.

The formula for this metric is listed below in equation 7

$$CC = (WM * NM) + (WS * NSS) \tag{7}$$

Where **WM** stands for the weight that is going to be given to multiplications and **WS** stands for the weight to sums and subtractions, **NM** and **NSS** are the number of multiplications, sums and subtractions that happen in the model for a single prediction, and **CC** is the Computational cost. The number of multiplications is obtained by equation 8

$$NM = \sum_{i=0}^{nl-1} (NNL_i * NNL_{i+1}) \tag{8}$$

Where **NM** stands for the Number of Multiplications, **NNL** stands for Number of Neurons at Layer and **nl** stands for the number of layers that the model has. Finally, if we assume that each layer has a ReLU activation function, and the final layer is a single neuron with a linear activation function, we could obtain the number of sums and subtractions, by following equation 9

$$NSS = \sum_{i=1}^{nl-1} ((NNL_i * 2) + ((NNL_{i-1} - 1) * NNL_i)) + (NN_{nl-1}) \tag{9}$$

Where **NNL**, stands for Number of Neurons at layer, **nl** corresponds to the total number of layers that the model has. To further explain the equation, to calculate the total number of sums in a general model, we begin by considering all of the layers, with the exception of the last layer. The last layer is not included in the calculation because it is an output layer without a ReLU activation function and is typically composed of a single neuron. We start the equation by counting the number of neurons in the current layer, and multiplying it by 2 (NNL$_i$ x 2), as each layer, with the exception of the input and output layers, have a ReLU activation function. After that, we need to calculate the number of sums that each neuron in the current layer has to perform to compute the weights attached to it. This is expressed by the term $((NNL_i - 1) * NNL_i)$ of the equation part of the equation. This idea is illustrated in Figure 21, essentially, we are extending the number of sums shown in Figure 21 to all of the neurons in the current layer. Finally, the term (NN_{nl-1}) is responsible for adding-up the number of sums that the output layer must perform, which is the number of neurons in its preceding layer.

All the assumptions that were done to establish the formulas happened because the general architecture of the proposed models will follow these patterns (i.e., will have the same activation functions and will always have a single neuron as an output layer).

It is also important to note that equation 7 does not provide a precise estimation of the model cost, to properly estimate its computational cost we would need to consider several factors such as the hardware on which it is being executed (e.g., CPU, GPU, FPGA or ASIC), the input that is going to be used, and each assigned weight to all of the connections of the model and biases. Rather than that, this formula gives a rough estimation based only on the model architecture (i.e., number of neurons, number of layers and the activation function of each layer)

It will be important to have the computational cost metric to be able to visualize the tradeoffs between the usage of computational resources and the efficiency of the model, with this information it will be possible to choose a model that is both efficient in terms of computational resources and performs on an acceptable level.

With the MAE and "Precision" metrics in mind and after running several tests, it will be possible to deviate a model that will achieve a better result according to the MAE metric and another model that will perform better according to the "precision" metric, so later in this chapter it will be possible to compare both of them and talk about which one of them would be more appropriate to be used as a possible substitute for the previously described look-up table that the AV1 uses.

As a final note from this section, it is important to consider that these metrics are not used as loss functions or as factors that impact the training of the model (the loss functions that were used to train each model will be discussed at their own section). Instead, these metrics are used to properly visualize the performance and efficiency of the developed models.

5.4 Libraries, choosen language and IDE

This section will briefly cover the language choice, along with the used libraries, as well as the IDE that was used to build the Artificial Neural Network models.

The programming language used to develop the models was Python. Python is commonly chosen for machine learning-related projects, and this work was no exception. The preference for Python over other languages was due to its libraries that make many tasks required in a machine learning project, such as organizing a dataset, training a model, evaluating a model and many other tasks, much easier.

About the libraries, pandas, numpy, matplotlib, seaborn, sklearn, and Tensorflow were used for the development of this work. All of them will be briefly covered below:

- Pandas: Pandas is a library that is used for data manipulation and some tasks related to the "well-being" of the dataset, such as checking if there's any Null value, reorganizing the dataset according to a filter and so on.
- Matplotlib: is a plotting library for Python. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK. Since Seaborn is built upon matplotlib, sometimes matplotlib is necessary for some adjustments, like for example the size of the graph.
- Seaborn: Essentially, is a plotting library for Python. This library was used to plot most of the graphs that will be seen in the following sections
- Sklearn & TensorFlow: Sklearn (with the Keras API), along with TensorFlow, were the libraries that provided all the tools that were necessary to develop the models, such as functions to evaluate the model, train the model, divide the dataset into training and testing and so on.

The IDE used for the development of the Neural Network models was Google Colab basic, a cloud-based notebook environment that provides a web-based interface for creating and running code. it is commonly used for ML tasks due to its ease of use, since it technically can be used at any place, as long as you have an equipment capable of navigating through the Internet.

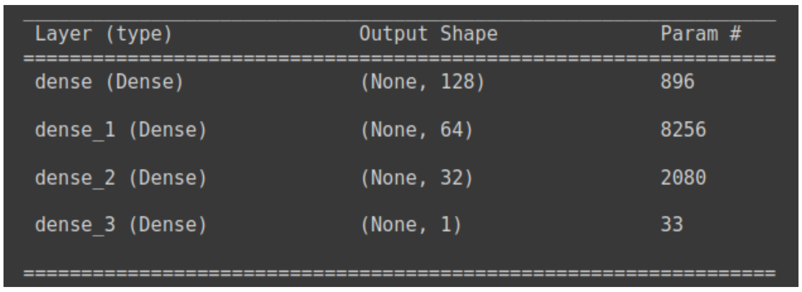
The IDE that was used for gathering the data to build the dataset was CLion, an IDE for C, that was used to debug the Libaom software and build logs throughout its execution.

5.5 Choosing a proper optimizer

One way to select an appropriate optimizer for a neural network model is to evaluate different optimizers on models with similar architectures, such as the same number of layers, neurons, activation functions, batch size, and number of training epochs. By comparing the efficiency of each optimizer under similar conditions, it is possible to determine which one is most suitable for the specific task and the model architecture. This approach can help to improve the generalization efficiency of the model, as an appropriate optimizer can help to minimize overfitting and improve the model's ability to generalize to new data

All of the optimizers provided by the Keras API were evaluated under the same model architecture and conditions to determine which performed the best. Each optimizer was trained for a total of 100 epochs, using a batch size of 32. Which is a commonly used batch size in deep learning, as it helps to control the memory usage during training. The 100 epochs were chosen as a default value, as it provides a sufficient number of iterations for the model to learn and generalize well, while also avoiding overfitting, another reason for choosing 100 epochs lies itself in the training time of each models.

Figure 39 – Model Architecture



```
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               896

 dense_1 (Dense)             (None, 64)                8256

 dense_2 (Dense)             (None, 32)                2080

 dense_3 (Dense)             (None, 1)                 33

=================================================================
```

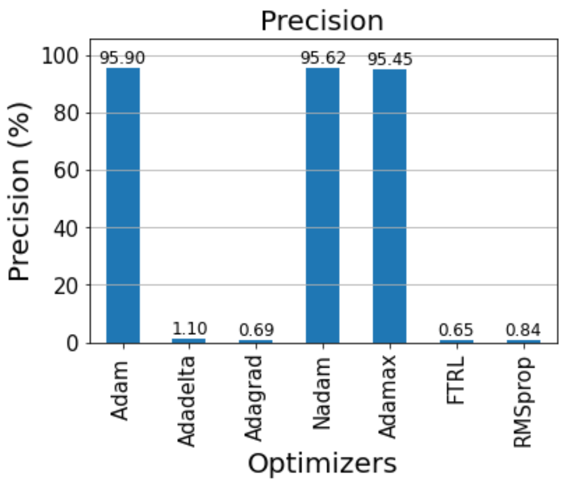
Source: Author (2023).

The model architecture, shown in Figure 39, consisted of the first three layers (128-64-32) utilizing a ReLU activation function, while the last layer employed a linear activation function, resulting in the output being the raw value (i.e., the CDF itself). The loss function that was used to train the model was the Mean-Squared Error (MSE).

After training the models and properly evaluating each one, we were able to produce the barplot shown in Figure 40.

As depicted in the image, the optimizers Adadelta, Adagrad, FTRL, and RMSprop

Figure 40 – Optimizers - Precision

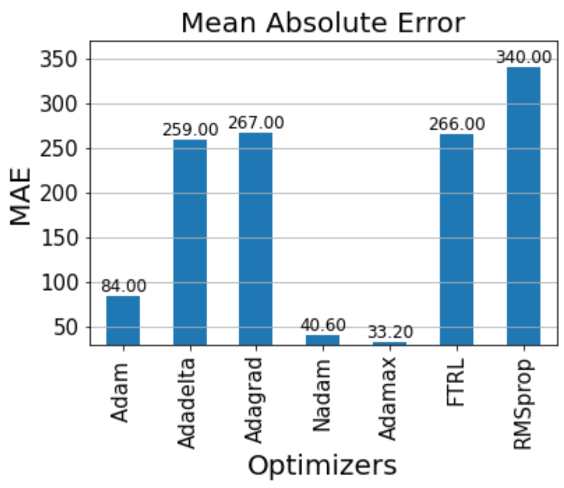


Source: Author (2023).

were not suitable for this task, as their precision remained close to 0. On the other hand, Adam, Nadam, and Adamax proved to be excellent optimizers for our task, as most of them achieved a precision above 95%. This outcome was expected, as Adadelta, Adagrad, FTRL, and RMSprop are older optimizers in comparison to Adam, Nadam, and Adamax. Additionally, Adam, Nadam, and Adamax have built upon these older optimizers and have added new features while correcting their failures.

Figure 41 shows a barplot that compares the MAE of each model. With this graph, we can infer that the mean absolute error (MAE) of the FTRL, RMSprop, Adagrad and Adadelta algorithms were exceptionally high, which justifies the poor efficiency seen in Figure 40. Additionally, it is also apparent that Nadam and Adamax were more accurate

Figure 41 – Optimizers - MAE

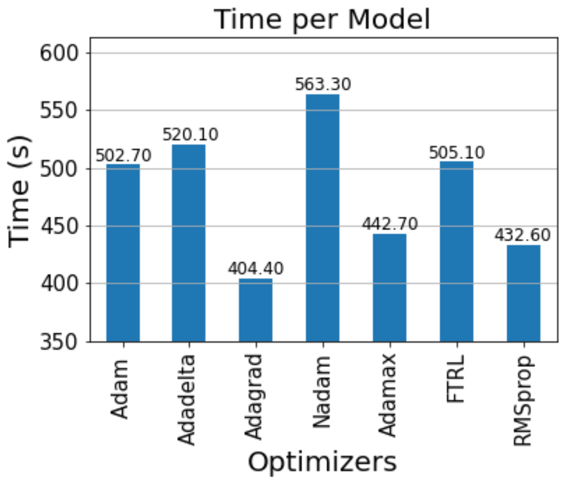


Source: Author (2023).

than Adam by a significant margin. Furthermore, this suggests that the Nadam and Adamax optimization algorithms may be more suitable for the objective of this work (i.e., estimating the CDF).

Figure 42 illustrates the duration required to train each model. As the architecture was consistent across all models, it was anticipated that the time to train each model would not vary significantly. However, the results indicate that there were variations in the training time, with some models taking longer than others. Additionally, it is noteworthy

Figure 42 – Optimizers - Training times



Source: Author (2023).

that although the Adagrad optimizer did not perform well in terms of accuracy, it was the fastest to train. This characteristic could be beneficial in situations where time is a crucial factor, such as real-time applications or large-scale datasets, where the ability to train the model quickly may outweigh the need for a high efficiency. Furthermore, this information could also be utilized to evaluate other optimization techniques and compare their time and efficiency tradeoffs

At last, Table 6 shows the values for each of the previously presented graphs in a summarized way. The first column are the optimizers that were being tested, the precision is the metric that was discussed in the metrics sections, the Test MAE is the MAE that was obtained by evaluating the model with the model.evaluate() function from the Keras API, and finally, the Time column refers to the amount of time that was necessary to train the model. By examining the precision column, we can see that the difference in precision between each model is substantial, with the closest values being seen in the Nadam, Adamax and Adam models. Similarly, when looking at the MAE metric, we can observe the significant difference between each optimizer. One noteworthy observation

Table 6 – Optimizers - Summary

Model_ID	*Optimizer*	*Precision(%)*	*Test MAE*	*Time(s)*
1	*Adam*	95.90%	84	502.7 (8m22s)
2	*Adadelta*	1.10%	259	520.1 (8m40s)
3	*Adagrad*	0.69%	267	404.4 (6m44s)
4	*Nadam*	95.62%	33.2	563.3 (9m23s)
5	*Adamax*	95.45%	40.6	442.7 (7m22s)
6	*FTRL*	0.65%	266	505.1 (8m25s)
7	*RMSprop*	0.84%	340	432.6 (7m12s)

Source: Author (2023).

from Table 6 is that the optimizers with the highest precision values, Adam, Nadam, and Adamax, also have the lowest MAE values. This suggests that these optimizers not only provide accurate predictions, but also have a lower error margin. Additionally, it is interesting to note that the optimizers with the lowest precision values, Adadelta, Adagrad, FTRL, and RMSprop, also have the highest MAE values, which make sense since the prediction metric is also affected by the MAE and not only the outlier values.

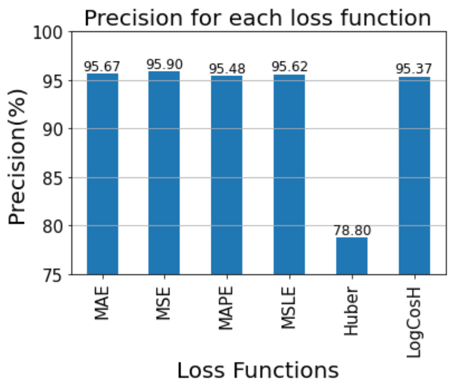
The computational cost (CC) metric was not considered in this section because it only evaluates the computational cost that is tied to the final model (i.e., the model that is acquired after the training stage), and since the model architecture was the same for each one of the optimizers (i.e, the hidden layers were the same, the number of neurons were the same and only the optimizers were changing), the CC would end up being a constant.

5.6 Choosing a proper Loss function

As was done in the previous section, it is crucial to determine the most appropriate Loss Function by testing a variety of regression losses while maintaining the same architecture, batch size, number of epochs, and optimizer. Choosing the right loss function can greatly impact the performance and efficiency of the model, as it determines the way in which the model is trained and how it will generalize to new data. The loss function is used to measure the difference between the predicted values and the actual values, and by selecting a loss function that is well-suited to the task at hand, the model can be trained more efficiently and effectively. Essentially, we could see the Loss function as a way to measure how wrong the model when compared to the correct answer. The loss functions were provided by the Keras API, All of the loss functions will be executed for 100 epochs using the ADAM optimizer. The general architecture will consist of 128-64-32-1 layers, with the first three layers using a ReLU activation function and the last layer utilizing a linear activation function, resulting in the output being the raw value (i.e., the CDF itself), the reasoning behind the number of epochs and batch_size were the same as they were in the previous seciton. Figure 39 depicts this architecture.

After training the models, we were able to gather data and generate several graphs to further analyze the results. Figure 43 illustrates the efficiency of each model in terms of the Precision metric. Unlike some of the optimizers discussed in the previous section,

Figure 43 – Loss Functions - Precision



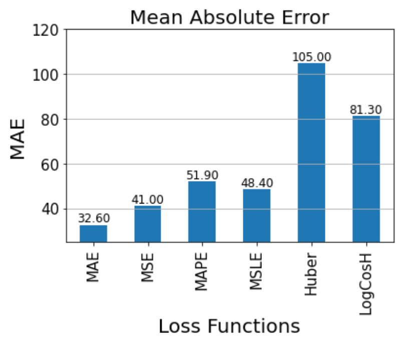
Source: Author (2023).

most of the loss functions performed quite well. The worst performing loss function was the Huber Loss Function, which could be attributed to its lack of sensitivity to outliers as well as its lack of robustness when compared to other loss functions.

Additionally, as seem at the theoretical reference chapter, the Huber loss function can also be less sensitive to small errors, meaning that it would not be properly helping the training step. However, most of the loss functions achieved an average of 94%, which still an excellent result.

The next graph, shown in Figure 44, illustrates the performance of various loss functions when compared to the Mean Absolute Error (MAE) metric. The results indicate that the Huber loss function and the LogCosh function performed poorly in comparison to the other options. It is worth noting that, although the LogCosh function had a poor performance, it still performed well in terms of the precision metric, this is because its MAE was below 100, and since the margin of acceptable error of the precision metric is 100, this resulted in the LogCosh performing pretty well at the precision metric, despite its high MAE when compared to the other loss functions. Unsurprisingly, the MAE loss function was the best-performing option for optimizing the MAE metric, however, it was

Figure 44 – Loss Functions - MAE

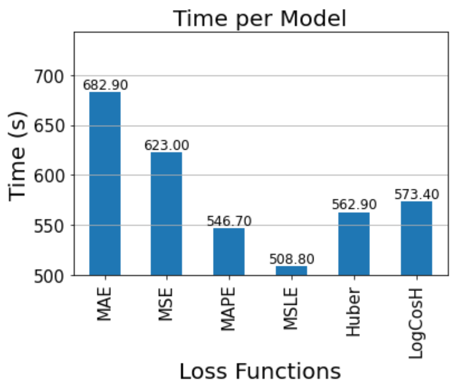


Source: Author (2023).

not significantly ahead of the other metrics.

The last relevant graph to demonstrate was the time required to train each of the models, as shown in Figure 45. As noted with the optimizers, the loss functions did not

Figure 45 – Loss Functions - Training times



Source: Author (2023).

have a significant impact on the training time of the models, which is expected, as the loss functions only iterate on the output of the model. Although the MAE loss function had the best performance in terms of the MAE metric, it also had the greatest impact on the training time of its respective model. However, as previously mentioned, the effect was not substantial enough to justify not choosing it over other metrics.

Finally, Table 7 summarizes the information from the graphs in a more condensed manner. The first column lists the Loss Functions used, the second column (Precision) is the metric discussed in the Metrics section, the Test MAE column shows the MAE obtained after evaluating the model using the Keras API's model.evaluate() function, and the Time column indicates the amount of time required to train the model. By comparing this table to the previous one, it can be observed that most of the values are less volatile, indicating that most of the Loss Functions are suitable for our task, with the exception of the Huber Loss Function. When examining the Precision metric, most of them average at 95%, which is an excellent result. Additionally, analyzing the Test MAE column, it can be noted that most of the results are similar, with the exception of the LogCosH loss function and Huber. The best performing Loss Functions are MAE and MSE, which is a plausible result, as they are designed to approximate the output values of linear models to the closest possible to the expected values.

Table 7 – Loss Functions - Summary

Model_ID	*Loss Function*	*Precision(%)*	*Test MAE*	*Time(s)*
8	*MAE*	95.67%	32.6	682.9 (11m22s)
9	*MSE*	95.90%	41.0	623 (10m23s)
10	*MAPE*	94.48%	51.9	546.7 (9m06s)
11	*MSLE*	95.62%	48.4	508.8 (8m28s)
12	*Huber*	78.80%	105	562.9 (9m22s)
13	*LogCosH*	95.37%	81.3	573.4 (9m23s)

Source: Author (2023).

In summary, the MAE Loss Function performed best when considering the MAE metric, and the MSE Loss Function performed best when considering the Precision metric. In terms of training time, all models averaged at 550 seconds, with LogCosH being the fastest and MAE the slowest, with a difference of only 180 seconds (3 minutes) between the two

As in the last section, the Computational Cost (CC) metric was not included because the architectures of the models had to stay the same so that we could properly evaluate the impact of each loss function.

5.7 Choosing a proper architecture for the Neural Network Model

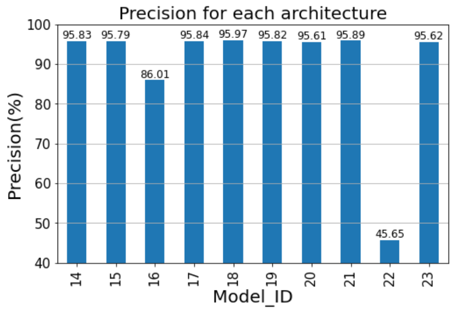
Finally, when building a model, it is essential to test different architectures to find the most efficient one in terms of Precision, MAE, and computational cost. Finding an appropriate architecture is crucial to achieve a model that is as small and efficient as possible. With the results of this section, we will be able to assess the impact of the model's architecture on its performance, and devise an architecture that would be considered optimal for our task. As in the previous section, all models were trained for 100 epochs using the ADAM optimizer and the MSE Loss Function, with all hidden layers using ReLU activation and the output layer using a linear activation to provide the raw value of the model. the reasoning behind the number of epochs and batch_size were the same as they were in the previous sections. Ten additional architectures were tested under these configurations, as follows:

- Model_ID 14 (16-1): 16 neurons in the first layer, 1 at the output layer
- Model_ID 15 (8-8-1): 8 neurons in the first layer, 8 in the second layer, 1 at the output layer
- Model_ID 16 (16-8-1): 16 neurons at the first layer, 8 in the second layer, 1 at the output layer
- Model_ID 17 (32-16-1): 32 neurons at the first layer, 16 in the second layer, 1 at the output layer
- Model_ID 18 (64-32-16-1): 64 neurons at the first layer, 32 at the second layer, 16 at the third layer, 1 at the output layer
- Model_ID 19 (128-64-32-16-1): 128 neurons at the first layer, 64 at the second layer, 32 at the third layer, 16 at the fourth layer, 1 at the output layer
- Model_ID 20 (256-128-64-32-16-1): 256 neurons at the first layer, 128 at the second layer, 64 at the third layer, 32 at the fourth layer, 16 at the fifth layer, 1 at the output layer
- Model_ID 21 (512-256-128-64-32-16-1): 512 neurons at the first layer, 256 at the second layer, 128 at the third layer, 64 at the fourth layer, 32 at the fifth layer, 16 at the sixth layer, 1 at the output layer
- Model_ID 22 (1024-512-256-128-64-32-16-1): 1024 neurons at the first layer, 512 at the second layer, 256 at the third layer, 128 at the fourth layer, 64 at the fifth layer, 32 at the sixth layer, 16 at the seventh layer, 1 at the output layer
- Model_ID 23 (512-1024-1024-1024-512-256-128-64-1): 512 neurons at the first layer, 1024 at the second layer, 1024 at the third layer, 1024 at the fourth layer, 512 at the fifth layer, 256 at the sixth layer, 128 at the seventh layer, 64 at the eighth layer, 1 at the output layer

Each one of the models will be called by its number, for example, model (16-1) will just be referred to as “Model 14” for the sake of simplicity. Since these models have a diverse range of sizes, it will be possible to evaluate how this characteristic impact on each one of the metrics, it is important to have a wide-range of architectures to properly select a good architecture. While these models can’t represent every possible architecture within their range, they give a good estimate to a wide variety of models.

As did on previous sections, the first graph that will be analyzed is the precision for each one of the architectures, this graph can be seemed at Figure 46, from this figure it is possible to see that most of the models averaged a good precision, with the exception of Model 16 and 22, that ended up below the average, possibly caused by a bad generalization on the training step, which is unusual but can happen sometimes.

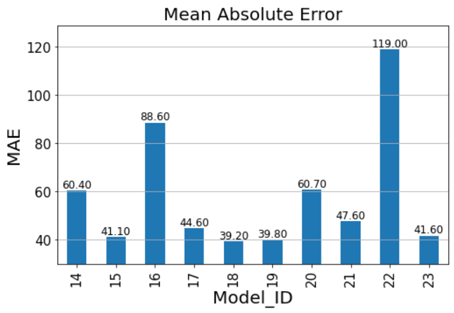
Figure 46 – Architectures - Precision



Source: Author (2023).

The next metric that is going to be analyzed is the MAE metric, the generated graph for this task can be seemed at Figure 47, by looking at the graph it is noticeable that most of the models averaged on a good metric, with the exception of 16 and 22, possibly because of the same previously mentioned reason.

Figure 47 – Architectures - MAE

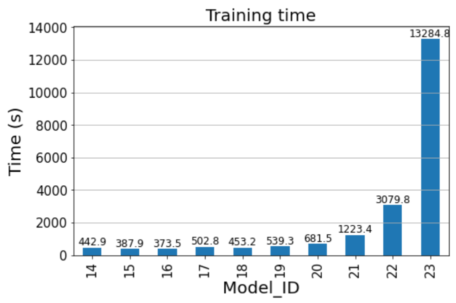


Source: Author (2023).

The next graph that is going to be discussed is seemed at Figure 48, this graph shows the different time to train each one of the models. Differently from previous sections, the training time of each architecture changed a lot, this behavior was expected since the size of each architecture was increasing for each model, increasing the model we have more weights, more biases, more activation functions, and it usually takes a longer time to compute even after trained

The final metric that needs to be considered in this section is the computational cost. To properly calculate the computational cost of each model, we first needed

Figure 48 – Architectures - Training times



Source: Author (2023).

to define the weight (or time required) for sums and multiplications. For this work, the computational cost of each model was calculated assuming a weight of three for multiplications and a weight of one for sums/subtractions. This can be thought of as each multiplication taking an average of three nanoseconds to complete, while a sum or subtraction would take one nanosecond. The number of multiplications and sums for each one of the models are displayed in Table 8.

Table 8 – Architectures - Multiplications & Sums

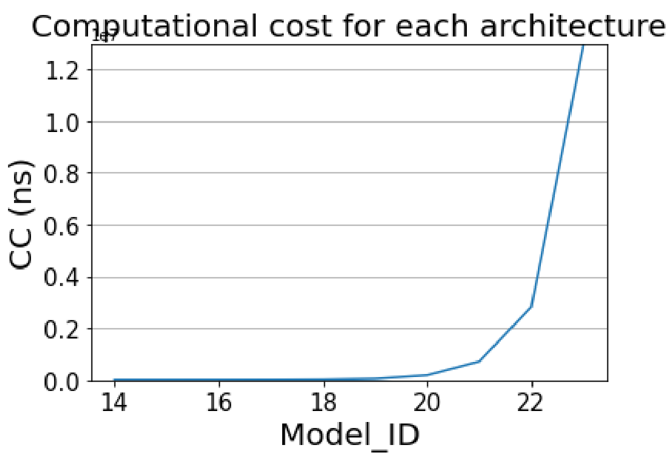
Model_ID	*Number of multiplications*	*Number of sums+subtractions*	*Number of Neurons*
14	128	144	17
15	128	144	17
16	248	272	25
17	752	832	49
18	3024	3136	113
19	11664	11904	241
20	45328	45824	497
21	178192	179200	1009
22	706064	708096	2033
23	3321408	2977792	4545

Source: Author (2023).

With the number of multiplications and sums for each model at hand, the next step is to multiply each one of them by their respective weights. With the results, it is possible to construct a graph to demonstrate the computational cost of each model and evaluate how much cost each architecture incurs. After performing each one of the multiplications, we obtain the graph in Figure 49. By examining the scale of the graph on the Y-axis (10^7), it is possible to see the magnitude of the difference between one model and another in terms of computational cost.

By looking at Table 8 again and comparing Model 19 to Model 18, which have a difference of 128 neurons each, it is striking the difference we will be able to see in the

Figure 49 – Architectures - Computational cost

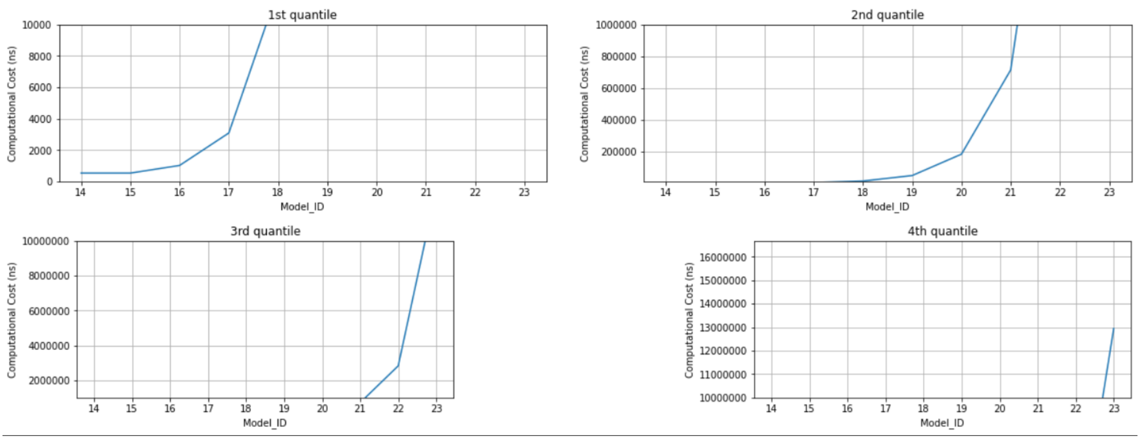


Source: Author (2023).

number of sums and multiplications. We could do the same exercise for Models 20 and 19, with a difference of 256 neurons, but with the amount of sums and subtractions being almost three times that of its predecessor. This shows that the computational cost tends to get worse as more neurons are added to the architecture.

To properly see all of the characteristics of the graph at Figure 49 it was necessary to divide him in 4 quantiles, that are shown in Figure 50.

Figure 50 – Architectures - Computational Cost - Quantiles



Source: Author (2023).

By looking at the first quantile we can see that the difference between the initial models are actually quite small, which makes sense if we considered that the early models have small differences in the number of neurons, but as the number of neurons increase we start to see the curve assume a more vertical shape, possibly because of the increase of the number of neurons between each model. By looking at the second quantile we see the same result of the previous one but for the Models 18,19,20 and 21, at the third quantile we only see Models 21 and 22, and finally at the fourth quantile we are finally able to see the Model 23. Although not proved, it is possible that the number of neurons have some kind of relation with the number of multiplications and sums, Table 9, shows the proportion between the number of multiplications and sums of each models, as well as the difference in the number of neurons from one model to another. By analyzing the proportion of multiplications and sums it is possible to notice, that the proportion is maintained as long as the difference between each model are constantly doubled. However, we still have to consider that the architecture of the models may be the cause of this constant proportion.

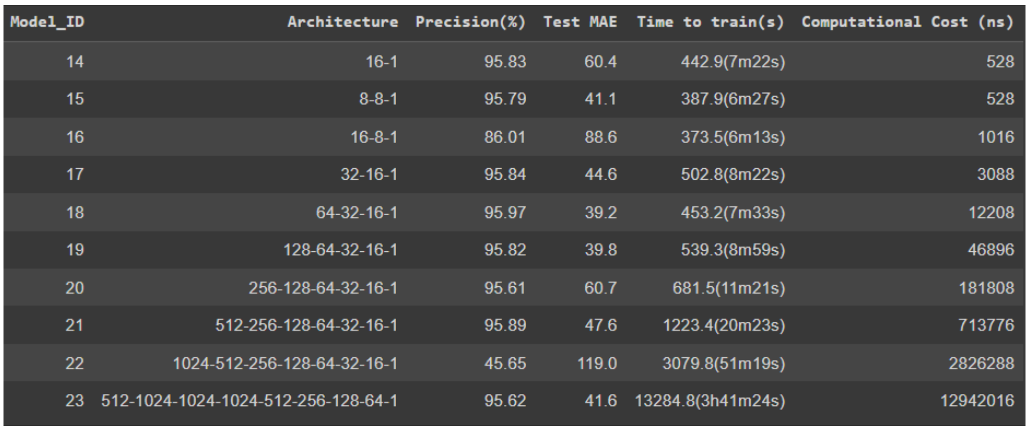
Table 9 – Architectures - Proportions

Model_ID	*Proportion of multiplications*	*Proportion of sums*	*Difference*
14	-	-	-
15	1	1	0
16	1.93	1.88	8
17	3.03	3.05	24
18	4.02	3.76	64
19	3.85	3.88	128
20	3.88	3.84	256
21	3.93	3.91	512
22	3.96	3.95	1024
23	4.70	4.20	2512

Source: Author (2023).

Finally, Figure 51 presents a summary of all of the data that was used to develop the previously shown graphs.

Figure 51 – Architectures - Summary



Model_ID	Architecture	Precision(%)	Test MAE	Time to train(s)	Computational Cost (ns)
14	16-1	95.83	60.4	442.9(7m22s)	528
15	8-8-1	95.79	41.1	387.9(6m27s)	528
16	16-8-1	86.01	88.6	373.5(6m13s)	1016
17	32-16-1	95.84	44.6	502.8(8m22s)	3088
18	64-32-16-1	95.97	39.2	453.2(7m33s)	12208
19	128-64-32-16-1	95.82	39.8	539.3(8m59s)	46896
20	256-128-64-32-16-1	95.61	60.7	681.5(11m21s)	181808
21	512-256-128-64-32-16-1	95.89	47.6	1223.4(20m23s)	713776
22	1024-512-256-128-64-32-16-1	45.65	119.0	3079.8(51m19s)	2826288
23	512-1024-1024-1024-512-256-128-64-1	95.62	41.6	13284.8(3h41m24s)	12942016

Source: Author (2023).

The first column describes the Model_ID, which, as the name indicates, is the ID of the model, basically a name that is given to him to differentiate him over the others. The Architecture column, describes the architecture of each one of the models, by separating each layer with a semi-column. The precision column describes the precision that was achieved by each one of the trained models. The "Test MAE" column shows the results that were obtained after using the model.evaluate() function from the Keras API. The "time to train" column, as suggested by the name, is the required time to train each one of the models and finally the "Computational Cost" column is the CC obtained to each one of the models by using the previously defined equations at the metrics section.

Upon examination of Table 8, it is clear that there is a significant difference in the number of sums and multiplications between Model 19 and Model 18, which differ by 128 neurons. Similarly, when comparing Model 20 and Model 19, which differ by 256 neurons, it is evident that the number of sums and multiplications increases nearly threefold. This demonstrates that the computational cost escalates as the number of neurons in the architecture increases.

By analyzing the results at Figure 51 it is possible to notice that the best architecture in terms of Precision and MAE was the Model 18, with 64 neurons at the first layer, 32 at the second, 16 at the third and one neuron at the output layer.

5.8 Final Results

To conclude the development of this work, considering the previous sections' results, a total of 23 models were trained until now. By reutilizing the results of these 23 models we have built three new models that have the potential to perform better in their respective metric. These models will be compared in order to evaluate their tradeoffs. It is important to note that while we are selecting the best architecture, optimizer, and loss function for a specific metric, it does not guarantee that this model will be the best possible model for this metric. It only assures that we are choosing metrics and optimizers that are more suited to our target metric than when comparing to standard choices. To build these models, we will first need to revisit the results from previous sections.

Table 10 presents the best-performing architecture, optimizer, and loss function for the precision metric that were obtained throughout this chapter. The model 64-32-16-1 was the best architecture, Adam was the best optimizer and the MSE loss function was the most efficient in terms of precision. Further on, Table 10 also demonstrates the best performing metrics in terms of the MAE metric. As it was for precision, the best

Table 10 – Optimal Choices - Precision & MAE

Model_ID	*Characteristic*	*Best choice*	*Precision(%)*	*MAE*	*Optimized for*
18	*Architecture*	64-32-16-1	95.97	-	Precision
1	*Optimizer*	Adam	95.90	-	Precision
1	*Loss Function*	MSE	95.90	-	Precision
18	*Architecture*	64-32-16-1	-	39.2	MAE
4	*Optimizer*	Nadam	-	33.2	MAE
8	*Loss Function*	MAE	-	32.6	MAE

Source: Author (2023).

architecture was 64-32-16-1, the best optimizer was Nadam and unsurprisingly the best performing loss function was the MAE. Considering the informations that were presented at the Table 10, it will be possible to deviate three models that will be optimized for each one of the Metrics, all of the models can be seemed at Table 11.

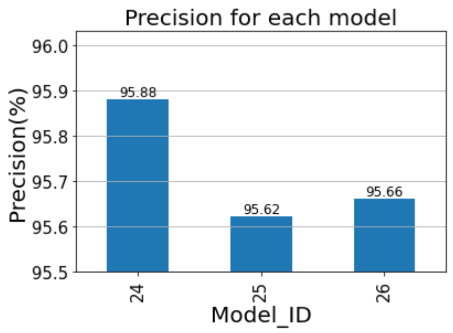
Table 11 – Final models

Model_ID	*Architecture*	*Optimizer*	*Loss Function*	*CC*	*Optimized for*
24	64-32-16-1	Adam	MSE	12208ns	Precision
25	64-32-16-1	Nadam	MAE	12208ns	MAE
26	16-1	Nadam	MAE	528ns	CC

Source: Author (2023).

All of these models were trained for 250 epochs, with a batch size of 16. After training each one of the models, the next step to take was to evaluate the generated models in terms of precision, MAE and CC. As did in the previous sections, The first graph that will be presented is a bar plot. Figure 52 shows the precision of each one of the models.

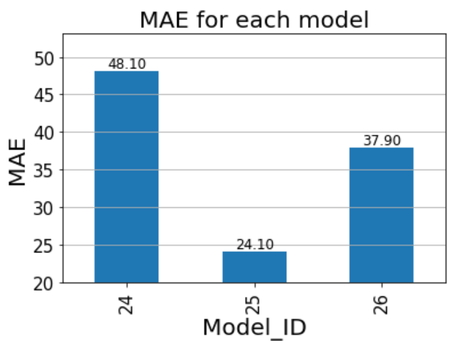
Figure 52 – Final models - Precision



Source: Author (2023).

From the graph, it is possible to notice that the models, overall, had a good performance, with most of them averaging at 95% of precision. It is also noticeable that the Model 24 is slightly better at this metric, which makes sense, since he was designed to outperform in this metric. Further on, the next graph that needs to be considered will be the graph at Figure 53. As for the previous graph, most of the models had good results,

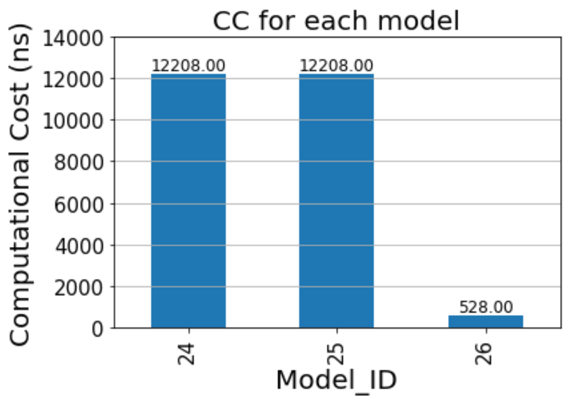
Figure 53 – Final models - MAE



Source: Author (2023).

but this time we can see a larger gap between Model 25 and the others, this result was not a surprise either, since Model 25 was built to perform on this metric. Finally, the last relevant metric to consider is the Computational Cost of each one of the models, this graph will be shown at Figure 54

Figure 54 – Final models - Computational Cost



Source: Author (2023).

Differently from what we have seen so far, the models at Figure 54 presented a huge gap of CC when comparing to Model 26. This result also makes sense when the architecture of each one of the models are considered, along with the correlation that the CC metric has to its architectures. To continue with the analysis, all of the results will be summarized at Table 12

The first column, "model_id," serves as the designated name for the model. The "Precision" column represents the metric defined during the metrics section. The "Test

Table 12 – Summary - Final models

Model_ID	*Precision(%)*	*Test MAE*	*Training time (s)*	*CC(ns)*
1	95.88%	48.1	2604.3 (43m24s)	12208
2	95.62%	24.1	2424.27 (40m24s)	12208
3	95.66%	37.9	2066.4 (34m26s)	528

Source: Author (2023).

MAE" column refers to the Mean Absolute Error (MAE) obtained while utilizing the "model.evaluate()" function from the Keras API. The "Training Time" column displays the duration it took to train each model. Lastly, the "CC" column represents the computational cost metric, also discussed during the metrics section.

In order to deviate a final model from this table, it is necessary to examine the metrics in more depth. Taking into account that the dataset used to validate the model consists of 17,335 entries, a precision of, for example, 95.88% would indicate that only 714 values out of the 17,335 were out of the proposed scale. For the MAE metric, a MAE of 24.1 would imply that the average error indicates a variation of 0.06% on the probability of each symbol. In regard to the CC metric, it must be remembered that it represents the time spent on a single entry in the Neural Network. To compare the CC metric, the CC of Model 24 and 25 can be divided by the CC of Model 26 to determine how much slower they are in comparison. With these considerations in mind, a table can be developed that includes all of these details, allowing for the deviation of the best model. This table is presented in Table 13.

Table 13 – Final Models - Tradeoffs

Model_ID	*Number of missed values*	*Change in probability (%)*	*Performance ratio*
24	714	0.15	-
25	763	0.06	-
26	760	0.11	23

Source: Author (2023).

As illustrated in Table 13, it is evident that Model 26 is the superior option. The use of a model that is 23 times slower than another one, solely due to a difference of 46 missed values, cannot be justified. The MAE metric also supports this claim, as it shows that Model 26, while also being 23 times faster, achieves a difference of only 0.05% in efficiency when comparing to Model 25. Furthermore, in the context of this model being applied in a video encoder, it is crucial for the model to perform as efficiently as possible, both in terms of CC and precision, as video encoders are used in many industries such as entertainment, security, and telecommunication, the efficiency of the model will affect the overall performance of the system.

To conclude, all of the models[1] and graphs [2] presented in this chapter can be found in the footnote of the page

[1]https://colab.research.google.com/drive/1Wg8Dw7mGnV5-zqj4iZxedu8K3gsQQJRE?usp=sharing
[2]https://colab.research.google.com/drive/1p5mgDIC-$_{j}UtvhnA4w-_{H}PKaRkV01FSz?usp=sharing$

6 FINAL CONSIDERATIONS

This chapter is going to cover the final considerations of this work, along with some future works.

6.1 Conclusion

Video coding is an area that is constantly in need of new tools to further optimize its results. The increasing demands of the 21st century, particularly the growing use of visual resources, have made this optimization increasingly necessary. New technologies, such as AI, have the potential to provide these new tools and therefore have become an important area of research in video coding.

The recent growth in AI has led to new alternatives in various fields, including video coding. However, the application of AI in video coding is still a relatively new area and requires further investigation. This research focused on one aspect of video coding, the AV1 codec, which is the most recent codec that has been growing in popularity. due to this growth in popularity and how new The AV1 codec is, it was selected to be the codec to be the object of study in this work.

The AV1 codec, as with every other codec that follows the Hybrid-workflow, has an entropy stage, which is the final stage in generating the final bitstream of the video sequence being encoded. At this stage, the probability of all the Syntax Elements (SEs) is considered. Properly estimating the probability of each SE is essential to achieve a high compression rate. Among the SEs, there is a subtype called RSEs, which only appear after the quantization step and, as stated by Ramos (2019), are responsible for most of the syntax elements in a video sequence.

To address this problem, this monograph proposed an ANN model to estimate the probability of residual Syntax Elements of the AV1 codec, specifically the DC_sign RSE. A total of 26 models were evaluated and trained using a dataset of 86,671 entries. The software libaom was used to gather the probabilities of the DC_sign RSE, and six video sequences were encoded to gather the probabilities. These models were trained to choose the best configurations for each task that was expected in a model that would replace the probability estimation of the DC_sign element for the AV1.

The final model obtained a precision of 95.66%, to estimate the values of each CDF, with an estimated runtime of 528ns per entry. This model has the potential to improve the AV1 codec's coding efficiency by more accurately estimating the probability of the DC_sign RSE. This serves as a demonstration of the feasibility of utilizing AI in video coding, particularly for determining the probabilities of SE.

In conclusion, this research has developed a neural network model that can estimate the probability of one of the RSEs of the AV1 codec. The model's performance was evaluated, and it was determined that the final model is the most appropriate choice for this work. This research also highlights the potential of AI in video coding, and the need for further investigation in this area. However, the proposed model is only applied to the estimation of one RSE, so it is necessary to further research to estimate the other RSEs and to test the model with the AV1 codec to validate its performance in a real-world scenario.

6.2 Future works

In this section, we will discuss several potential avenues for future research that build on the work presented in this monograph.

- The first suggestion is to test the final model in a practical scenario by coupling it with AV1 and comparing the results with the baseline version of AV1. This would be the most appropriate next step, as it would validate the proposed solution in a realistic scenario. The results of this work would be crucial in determining the effectiveness of the model.
- Another possible line of research would be to develop a Neural Network Model that considers all of the context-modeled RSEs, rather than just the DC_Sign RSE. This suggestion would also be a logical follow-up from the results of this work and is expected to bring more compression gains. However, it would require collecting relevant information for all the context-modeled RSEs, which can be a time-consuming task.
- Another interesting avenue of research would be to develop a Neural Network Model that substitutes the static look-up table of AV1 with a dynamic one. This would be highly beneficial to the AV1 codec, as it would adjust the coding parameters according to the specific characteristics of the video sequence. However, this suggestion would be one of the most challenging to implement, as it would require changes in many stages of the encoder.

- Another potential direction for future research is to develop a model that can estimate the probability of all SEs in the AV1 codec, not just the RSEs. This would be a time-consuming task, as all SEs would need to be considered, but it could potentially provide additional compression gains.
- Another approach that could be taken is to explore other machine learning techniques, such as decision trees or gradient boosting, to see if they can provide better results than the ANN-based model proposed in this work
- Another area of research that could be of interest is to explore hardware architectures that can optimize the execution of the ANN-based model, by reducing the number of multiplications and sums required. This could be particularly beneficial for real-time video coding applications
- Finally, another important avenue for future research is to investigate methods for "properly" estimating the probability of each RSE, instead of using the probabilities obtained from the reference software. This could involve re-encoding video sequences with different settings or using other techniques to estimate the probabilities. However, it is important to note that more research would be needed to determine if this is a viable approach and if it would provide additional compression gains.

Overall, there are many potential directions for future research that build on this work and could be beneficial to AV1s research.

REFERENCES

BARROS, M. **IDC prevê crescimento no mercado mundial de IA em 2021**. 2021. Disponível em: https://olhardigital.com.br/.

BORGES, A. M. **Soluções para redução de complexidade da transcodificação de vídeos HEVC para AV1**. Dissertação (Mestrado) — Universidade Federal de Pelotas, 2019.

BRASPENNING F. THUIJSMAN, A. J. M. M. W. e. P. J. **Artificial Neural Networks: An Introduction to ANN Theory and Practice**. 1. ed. [S.l.]: Springer, 1995. (Lecture Notes in Computer Science №931).

BROSS, B. et al. Versatile video coding (draft 10). **Joint Video Experts Team (JVET) of ITU-T SG**, v. 16, p. 3–12, 2020.

BURKOV, A. **The hundred-page machine learning book**. [S.l.]: Andriy Burkov Quebec City, QC, Canada, 2019. v. 1.

CHEN, M. et al. Machine learning based symbol probability distribution prediction for entropy coding in av1. In: IEEE. **2020 IEEE International Conference on Image Processing (ICIP)**. [S.l.], 2020. p. 3374–3378.

CHEN, Y. et al. An overview of core coding tools in the av1 video codec. In: **2018 Picture Coding Symposium (PCS)**. [S.l.: s.n.], 2018. p. 41–45.

DAEDE, T.; NORKIN, A.; BRAILOVSKIY, I. **Video Codec Testing and Quality Measurement**. [S.l.], 2019. Work in Progress. Disponível em: https://datatracker.ietf.org/doc/draft-ietf-netvc-testing/07/.

DOZAT, T. Incorporating Nesterov Momentum into Adam. In: **Proceedings of the 4th International Conference on Learning Representations**. [S.l.: s.n.], 2016. p. 1–4.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep learning**. [S.l.]: MIT press, 2016.

HAN, J. et al. A technical overview of av1. **Proceedings of the IEEE**, v. 109, n. 9, p. 1435–1462, 2021.

ITU-T. **High Efficiency Video Coding**. Seattle, WA, 2013. Available at <https://www.itu.int/itu-t/recommendations/rec.aspx?rec=11885lang=en>. Visited in April, 2022.

KIM, J. et al. Fast inter-prediction based on decision trees for av1 encoding. In: IEEE. **ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)**. [S.l.], 2019. p. 1627–1631.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. **arXiv preprint arXiv:1412.6980**, 2014.

KINSLEY, D. K. H. **Neural Networks from Scratch in Python**. 1. ed. [s.n.], 2020. Disponível em: libgen.li/file.php?md5=885d87e9cbb51164026920c589da7cf5.

LIU, C. **AOM AV1 vs HEVC/H.265, the Future of 4K/8K UHD Codec**. 2021. <https://www.macxdvd.com/mac-dvd-video-converter-how-to/aom-av1-vs-hevc-h265.htm> last accessed 28 Jun. 2022.

MA, C. et al. Neural network-based arithmetic coding for inter prediction information in hevc. In: IEEE. **2019 IEEE International Symposium on Circuits and Systems (ISCAS)**. [S.l.], 2019. p. 1–5.

MARPE, D.; SCHWARZ, H.; WIEGAND, T. Context-based adaptive binary arithmetic coding in the h. 264/avc video compression standard. **IEEE Transactions on circuits and systems for video technology**, IEEE, v. 13, n. 7, p. 620–636, 2003.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. **The bulletin of mathematical biophysics**, Springer, v. 5, n. 4, p. 115–133, 1943.

MCMAHAN, H. B. et al. Ad click prediction: a view from the trenches. In: **Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)**. [S.l.: s.n.], 2013.

MEADE, N.; ISLAM, T. An evaluation of regression models for forecasting. **International Journal of Forecasting**, Elsevier, v. 21, n. 4, p. 635–650, 2005.

MEHLIG, B. **Machine Learning with Neural Networks: An Introduction for Scientists and Engineers**. New. [S.l.]: Cambridge University Press, 2021.

MONTEIRO, E. R. Caracterização energética da codificação de vídeo de alta eficiência (hevc) em processador de propósito geral. 2017.

OZER, J. **It's The Year of AV1. 2023, That Is**. 2021. <https://www.macxdvd.com/mac-dvd-video-converter-how-to/aom-av1-vs-hevc-h265.htm> last accessed 28 Jun. 2022.

RAMACHANDRAN, A. **Decode to encode**. First edition. [S.l.]: Avinash Ramachandran, 2018.

RAMOS, F. L. L. Efficient high-throughput and power-saving hardware architectural design for the hevc entropy encoder. 2019.

RAMOS, F. L. L. et al. Residual syntax elements analysis and design targeting high-throughput hevc cabac. **IEEE Transactions on Circuits and Systems I: Regular Papers**, v. 67, n. 2, p. 475–488, 2020.

RICHARDSON, I. E. **The H.264 Advanced Video Compression Standard, Second Edition**. 2. ed. [S.l.]: Wiley, 2010. ISBN 0470516925,9780470516928.

RIVAZ, P. D.; HAUGHTON, J. Av1 bitstream & decoding process specification. **The Alliance for Open Media**, v. 681, 2018.

ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. **Psychological review**, American Psychological Association, v. 65, n. 6, p. 386, 1958.

RUTZ, G. **The Cost of Codecs: Royalty-Bearing Video Compression Standards and the Road that Lies Ahead**. 2016. <https://www.cablelabs.com/blog/the-cost-of-codecs-royalty-bearing-video-compression-standards-and-the-road-that-lies-ahead> last accessed 28 Jun. 2022.

SALOMON, D. **Data Compression: The Complete Reference**. Second. [s.n.], 2004. xx+899 p. ISBN 0-387-40697-2. Disponível em: http://www.ecs.csun.edu/ dxs/DC3advertis/Dcomp3Ad.html.

SHANNON, C. E. A mathematical theory of communication. **The Bell system technical journal**, Nokia Bell Labs, v. 27, n. 3, p. 379–423, 1948.

SONG, R. et al. Neural network-based arithmetic coding of intra prediction modes in hevc. In: IEEE. **2017 IEEE Visual Communications and Image Processing (VCIP)**. [S.l.], 2017. p. 1–4.

STATISTA. **Average YouTube video length as of December 2018, by category**. 2018. <https://www.statista.com/statistics/1026923/youtube-video-category-average-length/> last accessed 28 Jun. 2022.

STATISTA. **YouTube - Statistics Facts**. 2022. <https://www.statista.com/topics/2019/youtube/> last accessed 28 June. 2022.

SU, H. et al. Machine learning accelerated transform search for av1. In: IEEE. **2019 Picture Coding Symposium (PCS)**. [S.l.], 2019. p. 1–5.

ZATT, B. Energy-efficient algorithms and architectures for multiview video coding. 2012.

ZEILER, M. D. ADADELTA: an adaptive learning rate method. **CoRR**, abs/1212.5701, 2012. Disponível em: http://arxiv.org/abs/1212.5701.