Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Adaptive Context Modeling and Hyperparameter Selection in Neural-based Data Compression

Modelagem de Contextos Adaptativa e Seleção de Hiperparâmetros em Compressão de Dados Baseada em Redes Neurais

Lucas S. Lopes

Tese apresentada como requisito parcial para
conclusão do Doutorado em Informática

Orientador
Prof. Dr. Ricardo Lopes de Queiroz

Brasília
2024

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Adaptive Context Modeling and Hyperparameter Selection in Neural-based Data Compression

Modelagem de Contextos Adaptativa e Seleção de Hiperparâmetros em Compressão de Dados Baseada em Redes Neurais

Lucas S. Lopes

Tese apresentada como requisito parcial para
conclusão do Doutorado em Informática

Brasília, 19 de Julho de 2024

Acknowledgements

I would like to thank my advisor, Prof. Dr. Ricardo Lopes de Queiroz, for his constant and sincere feedback, which made me greatly improve as a researcher.

I would also like to thank Dr. Philip Andrew Chou, for his substantial contributions to this work, and Lucas Gribel dos Reis, for proofreading the final text.

Finally, I would like to thank my wife, Jessica, who supported me during the entire period of my Doctorate's Degree, and my parents, who first motivated me to study.

Abstract

Neural-based data compression has not yet reached its full potential. Context modeling for arithmetic coding is usually done through frequency counting and look-up tables (LUTs). These models are usually continuously updated as new samples are seen. All neural-based context models which have been proposed so far make use of previous training. We propose a neural-based method of context modeling for arithmetic coding in which the neural networks are trained on-the-fly. The model essentially begins as a uniform distribution, and gradually approaches the true probability distribution of the data, instead of the training data distribution. The method performs better than the simple frequency counting technique, and allows the increase of the context size to levels not possible with LUT-based methods. Black-box multi-objective hyperparameter optimization (MOHPO) methods exist which can be used in neural-based data compression. However, in data compression, the complexity of the compressor is generally as important, or more, than its compression performance. We propose a method of multi-objective hyperparameter optimization which naturally constructs the set of optimal solutions, or the lower-convex hull, in increasing order of complexity. This allows the algorithm to be stopped once the desireable value of compression performance, or the maximal value of acceptable complexity, is achieved. We compared this algorithm with state-of-the-art methods present in a popular MOHPO platform, with the proposed method showing competitive results.

Keywords: Arithmetic coding, context modeling, neural networks, hyperparameter optimization

Resumo

A compressão de dados baseada em redes neurais ainda não atingiu todo o seu potencial. A modelagem de contexto para codificação aritmética geralmente é feita por meio de contagem de frequência e tabelas de consulta (LUTs, do inglês, "look-up tables"). Esses modelos geralmente são atualizados continuamente à medida que novas amostras são vistas. Todos os modelos de contexto baseados em redes neurais que foram propostos até o momento fazem uso de pré-treinamento. Nós propomos um método de modelagem de contexto baseado em redes neurais para codificação aritmética em que as redes neurais são treinadas dinamicamente. O modelo começa essencialmente como uma distribuição uniforme, e gradualmente se aproxima da verdadeira distribuição de probabilidade dos dados, em vez da distribuição dos dados de treinamento. O método tem melhor desempenho do que a simples contagem de frequências, e permite o aumento do tamanho do contexto para níveis não possíveis com métodos baseados em LUT. Existem métodos caixa-preta de otimização de hiperparâmetros multiobjetivo (MOHPO, do inglês, "multi-objective hyperparameter optimization") que podem ser usados na compressão de dados baseada em redes neurais. Porém, em compressão de dados, a complexidade do compressor é geralmente tão importante, ou mais, do que seu desempenho de compressão. Propomos um método de otimização de hiperparâmetros multiobjetivo que constrói naturalmente o conjunto de soluções ótimas, ou o casco convexo inferior, em ordem crescente de complexidade. Isso permite que o algoritmo seja interrompido ao se atingir o desempenho de compressão desejado, ou o valor de complexidade máximo aceitável. Comparamos este algoritmo com métodos do estado-da-arte presentes em uma popular plataforma de

MOHPO, com o método proposto apresentando resultados competitivos.

Palavras-chave: Codificação aritmética, modelagem de contextos, redes neurais, otimização de hiperparâmetros

Contents

List of Figures

List of Tables

List of Acronyms

AC	Arithmetic Coding.
ALUT	Adaptive Look-Up Table.
APC	Adaptive Perceptron Coding.
ARNN	Adaptive Recurrent Neural Network.
AVC	Advanced Video Coding.
BO	Bayesian Optimization.
BPTT	Backpropagation Through Time.
CABAC	Context-Adaptive Binary Arithmetic Coding.
CNN	Convolutional Neural Network.
EHVI	Expected Hypervolume Improvement.
EI	Expected Improvement.
FLOP	Floating-Point Operations.
GD	Gradient Descent.
GLCH	Greedy Lower Convex Hull.
GPU	Graphics Processing Unit.

GRU Gated Recurrent Unit.

H.264 Video compression standard also known as MPEG-4 Part 10 or AVC.

H.265 Video compression standard also known as MPEG-H Part 2 or HEVC.

H.266 Video compression standard also known as MPEG-I Part 3 or VVC.

HEVC High Efficiency Video Coding.

HV Dominated Hypervolume.

IEEE Institute of Electrical and Electronics Engineers.

JPEG Joint Photographic Experts Group.

KLD Kullback Leibler Divergence.

LCH Lower Convex Hull.

LLM Large-Language Model.

LPS Least Probable Symbol.

LSTM Long Short-Term Memory Network.

LUT Look-Up Table.

MAC Multiply-Accumulate Operations.

MLE Maximum Likelihood Estimation.

MLP Multi-Layer Perceptron.

MOHPO Multi-Objective Hyperparameter Optimization.

MPS Most Probable Symbol.

MSE	Mean Squared Error.
ParEGO	Efficient global optimization for Pareto optimization.
PC	Perceptron Coding.
PDF	Portable Document Format.
PNG	Portable Network Graphics.
qNEHVI	Parallel noisy EHVI.
qNParEGO	Parallel noisy ParEGO.
ReLU	Rectified Linear Unit.
RNN	Recurrent Neural Network.
SGD	Stochastic Gradient Descent.
SOHPO	Single-Objective Hyperparameter Optimization.
SPL	Signal Processing Letters.
TBPTT	Truncated Backpropagation Through Time.
TIP	Transactions on Image Processing.
US	United States.
VAE	Variational Autoencoder.
VVC	Versatile Video Coding.

List of Symbols

a A neuron's output with activation function.

a_j A layer's j-th neuron output with activation function.

$\mathbf{a}$ A layer's output with activation function.

$\mathbf{a}^{(d)}$ MLP's d-th layer output with activation function.

$\mathbf{a}_n^{(d)}$ MLP's d-th layer output with activation function for the input at instant n.

$\bar{\mathbf{a}}_n^{(d)}$ Elman RNN's d-th layer output with activation function for the input at instant n.

b A neuron's bias.

b_j A layer's j-th neuron bias.

$b_j^{(d)}$ MLP's d-th layer j-th neuron bias.

$\mathbf{b}$ A layer's bias vector.

$\mathbf{b}^{(d)}$ MLP's d-th layer bias vector.

$\bar{\mathbf{b}}^{(d)}$ Elman RNN's d-th layer bias vector.

$\mathcal{B}$ Batch. Subset of data containing B samples.

$B, |\mathcal{B}|$ Batch size.

$\{\mathcal{B}_i\}$ Partition of the set $\{1, 2, ..., N\}$.

c One cost-function.

$c^{\min}$ Current estimate of the minimum value of the cost function c.

c_i One cost-function from the vector $\mathbf{c}$ of cost-functions.

$\mathbf{c}$ Vector of cost-functions.

C Complexity.

C_o Complexity operating point.

$C_{\mathbf{h}}$ Complexity associated with the hyperparameter vector $\mathbf{h}$.

$\Delta C_{\mathbf{h}'}$ difference between $C_{\mathbf{h}'}$ and $C_{\mathbf{h}}$.

$\mathcal{C}$ Closed set.

$d_{\max}$ Number of MLP layers.

$\mathscr{d}$ Some distance measure.

D Distortion.

$D_{\mathbf{h}}$ Distortion associated with the hyperparameter vector $\mathbf{h}$.

$D_{KL}(p||q)$ Kullback-Leibler divergence from q to p.

$\mathcal{D}$ Dataset.

$|\mathcal{D}|$ Dataset size.

E_0, E_1 Markov chain states.

E_t Current set of non-dominated solutions.

$\mathrm{EI}(\mathbf{h})$ Expected improvement for domain point $\mathbf{h}$.

$\mathrm{EHVI}(\mathbf{h})$ Expected hypervolume improvement for domain point $\mathbf{h}$.

$\mathcal{E}$ Edge set of graph $\mathcal{G}$.

f Output with activation function of a neural network of a single output.

$\hat{f}$ Output without activation function of a neural network of a single output.

$\boldsymbol{f}$ Outputs with activation function of a neural network of multiple outputs.

$\hat{\boldsymbol{f}}$ Outputs without activation function of a neural network of multiple outputs.

$F(x)$ Cumulative distribution function of X.

$\bar{F}(x)$ Cumulative distribution function of discrete random variable X which gives the midpoint of the step corresponding to x.

$\lfloor \bar{F}(x) \rfloor_{\ell(x)}$ Truncation of $\bar{F}(x)$ to $\ell(x)$ bits.

g Activation function.

$\boldsymbol{g}$ Function of multiple inputs and multiple outputs whose components are activation functions.

$\boldsymbol{g}^{(d)}$ MLP's d-th layer vector-valued activation function.

$\bar{\boldsymbol{g}}^{(d)}$ Elman RNN's d-th layer vector-valued activation function.

G_n Average self-information of sequences of length n.

$\mathcal{G}$ Graph.

h_k One hyperparameter from the vector $\mathbf{h}$ of hyperparameters.

$\mathbf{h}$ An hyperparameter vector.

$\mathbf{h}_{\text{min}}$ Minimal hyperparameter vector.

$\mathbf{h}_{\text{max}}$ Maximal hyperparameter vector.

$\mathbf{h}^*$ Selected hyperparameter vector during the execution of a GLCH select function.

H_n Average self-information per sample of sequences of length n.

$H(\mathcal{S})$ Source entropy.

$H(X)$ Entropy of random variable X.

$H(X|\boldsymbol{\xi})$ Entropy of X given $\boldsymbol{\Xi} = \boldsymbol{\xi}$.

$H(X|\boldsymbol{\Xi})$ Entropy of X conditioned on $\boldsymbol{\Xi}$.

$H(p)$ Entropy for probability distribution p.

$H(p, q)$ Cross-entropy of the distribution q relative to the distribution p.

$\mathrm{HV}(E_t)$ Hypervolume of set E_t.

$\mathcal{H}$ Set of possible hyperparameter vectors.

$I(c)$ Improvement for cost-function value c.

$I(\mathbf{c}(\mathbf{h}))$ Improvement for cost-function vector $\mathbf{c}(\mathbf{h})$.

$\mathcal{I}$ Subset of training data containing one sample.

J A layer's number of neurons.

$J^{(d)}$ MLP's d-th layer number of neurons.

$K^{(d)}$ MLP's d-th layer number of inputs.

K Number of hyperparameters being considered during hyperparameter optimization.

$\ell_1, \ell_2, ..., \ell_S$ Codeword lengths.

ℓ_i^* Optimal codeword lengths.

$\ell(x)$ Shannon-Fano-Elias coding codeword length for symbol x.

L Loss-function.

$L_{\mathcal{I}}$ Loss for one training sample.

$L_{\mathcal{B}}$ Loss for batch $\mathcal{B}$.

$L_{\mathcal{B}_i}$ Loss for batch $\mathcal{B}_i$.

L_{seq} Loss-function for a sequence.

L_n Loss for the n-th sample.

$L_{\mathbf{h}}$ Loss associated with the hyperparameter vector $\mathbf{h}$.

$\Delta L_{\mathbf{h}'}$ difference between $L_{\mathbf{h}'}$ and $L_{\mathbf{h}}$.

$\frac{\partial L}{\partial \theta}\Big|_{\theta_{\text{old}}}$ Partial derivative of L with respect to θ at θ_{old}.

$(\nabla_{\boldsymbol{\theta}} L)_{\boldsymbol{\theta}_{\text{old}}}$ Gradient of L with respect to $\boldsymbol{\theta}$ at $\boldsymbol{\theta}_{\text{old}}$.

$\mathcal{L}(\boldsymbol{\theta}|\mathbf{x}^{(N)})$ Likelihood of $\boldsymbol{\theta}$ given $\mathbf{x}^{(N)}$.

$m = S - 1$ Last symbol, assuming the alphabet are the first S nonnegative integers.

M Context size.

N Length of observed data.

N' Number of samples after which one RNN update is performed.

$N(x')$ Length of the data with x' as the context.

$\mathcal{O}$ Open set.

$\mathcal{O}'$ Set of deepest open nodes.

p Arbitrary probability distribution.

$p_{X|\boldsymbol{\Theta}}$ Distribution of X conditioned on $\boldsymbol{\Theta}$.

$p_{X|X',\boldsymbol{\Theta}}$ Distribution of X conditioned on X' and $\boldsymbol{\Theta}$.

$p_{\mathbf{X}^{(n)}}$ Model for the probability distribution of $\mathbf{X}^{(n)}$.

$p_{\mathbf{X}^{(N)}|\boldsymbol{\Theta}}$ Distribution of $\mathbf{X}^{(N)}$ conditioned on $\boldsymbol{\Theta}$.

$p_{\boldsymbol{\Psi}}$ Distribution of $\boldsymbol{\Psi}$.

$p_{\tilde{\boldsymbol{\Omega}}}$ Distribution of $\tilde{\boldsymbol{\Omega}}$.

$p_{\hat{\boldsymbol{\Omega}}}$ Distribution of $\hat{\boldsymbol{\Omega}}$.

$p_{\tilde{\boldsymbol{\Omega}}|\boldsymbol{\Psi}}$ Distribution of $\tilde{\boldsymbol{\Omega}}$ conditioned on $\boldsymbol{\Psi}$.

$p_{\boldsymbol{\Psi}|\tilde{\boldsymbol{\Omega}}}$ Distribution of $\boldsymbol{\Psi}$ conditioned on $\tilde{\boldsymbol{\Omega}}$.

$p_{\tilde{\Omega}_i}$ Probability distribution of $\tilde{\Omega}_i$.

p_i Probability of the i-th possible symbol, with $i = 1, ..., S$.

$\text{PDF}_{\mathbf{h}}(c)$ Estimated probability density function of c for domain point $\mathbf{h}$.

$\text{PDF}_{\mathbf{h}}(\mathbf{c})$ Estimated probability density function of $\mathbf{c}$ for domain point $\mathbf{h}$.

$\mathcal{P}$ Pareto set.

q Arbitrary probability distribution.

$q_{\mathbf{X}^{(n)}}$ Model for the probability distribution of $\mathbf{X}^{(n)}$.

$q_{\tilde{\boldsymbol{\Omega}}|\boldsymbol{\Psi}}$ Parametric approximation of the distribution of $\tilde{\boldsymbol{\Omega}}$ conditioned on $\boldsymbol{\Psi}$.

$\mathbf{r}$ Hypervolume reference point.

R Rate.

R_o Rate operating point.

$R_{\mathbf{h}}$ Rate associated with the hyperparameter vector $\mathbf{h}$.

S Alphabet size.

Softmax Softmax activation function.

$\mathcal{S}$ Random experiment representing the source.

$\mathbf{t}_1, \mathbf{t}_2, ..., \mathbf{t}_N$ The sequence $x_1, x_2, ..., x_N$ after one-hot-encoding each element.

T_k Number of possible values for hyperparameter h_k.

$v_t^{(k)}$ The t-th value that the hyperparameter h_k can take.

V_{nd} Set of objective vectors non-dominated by E_t.

$\mathcal{V}$ Vertex set of graph $\mathcal{G}$.

$w_{jk}^{(d)}$ MLP's d-th layer weight from the k-th input to the j-th output.

$\mathbf{w}$ A neuron's weight vector.

$\mathbf{w}_j$ A layer's j-th neuron weight vector.

$\mathbf{w}_j^{(d)}$ MLP's d-th layer j-th neuron weight vector.

$\mathbf{W}$ A layer's weight matrix.

$\mathbf{W}^{(d)}$ MLP's d-th layer weight matrix.

$\bar{\mathbf{W}}^{(d)}$ Elman RNN's d-th layer weight matrix.

x Realization from X.

x' Realization from X'.

$\mathbf{x}^{(n)} = \{x_1, x_2, ..., x_n\}$ Particular realization of $\mathbf{X}^{(n)}$.

X Random variable representing a single observation from the source.

X' Random variable representing the sample immediately before X.

$\{X_n\}_{n \in \mathbb{N}^+}$ Discrete-time random process.

$\mathbf{X}^{(n)} = \{X_1, X_2, ..., X_n\}$ First n random variables from the random process $\{X_n\}_{n \in \mathbb{N}^+}$.

$\mathcal{X}$ Alphabet.

$\mathcal{X}^{(n)}$ Set of all possible realizations of $\mathbf{X}^{(n)}$.

$\mathbf{y}$ Expected output associated with the input $\boldsymbol{\psi}$.

z A neuron's output without activation function.

z_j A layer's j-th neuron output without activation function.

$\mathbf{z}$ A layer's output without activation function.

$\mathbf{z}^{(d)}$ MLP's d-th layer output without activation function.

α Constant of exponentially weighted moving average.

γ Weight of complexity.

δ Diameter of graph $\mathcal{G}$.

Δ Quantization step.

ϵ Learning rate.

$\boldsymbol{\zeta}$ Function approximated by a neural network.

θ One generic model parameter.

θ_0 Particular value of θ.

θ_{old} Old value of θ.

θ_{new} New value of θ.

θ_{min} Minimum network parameter.

θ_{max} Maximum network parameter.

θ_q Quantized parameter.

$\boldsymbol{\theta}$ Model parameters. Particular realization from $\boldsymbol{\Theta}$.

$\boldsymbol{\theta}_{\text{old}}$ Old value of $\boldsymbol{\theta}$.

$\boldsymbol{\theta}_{\text{new}}$ New value of $\boldsymbol{\theta}$.

$\boldsymbol{\theta}_n$ Model parameters when processing the n-th sample.

$\boldsymbol{\Theta}$ Random vector representing the parameters of a model.

$\kappa(x)$ Absolute frequency of the symbol x.

$\kappa(x, x')$ Absolute frequency of the pair of symbols x, x'.

λ Weight of distortion.

λ' Constant related to the weight of distortion λ.

μ Controls the importance given to a distance in L compared to a distance in C when measuring the distance to the origin in one variant of the GLCH algorithm.

μ' Constant associated with the constant μ.

ν Number of cost-functions in $\mathbf{c}$.

$\boldsymbol{\xi}$ Context. One realization from $\boldsymbol{\Xi}$.

$\boldsymbol{\xi}_n$ Context for the n-th symbol in the sequence.

$\boldsymbol{\Xi}$ Random vector representing the context.

ρ_η Estimated probability that the η-th symbol is 1, in a sequence of binary symbols associated with a given context.

σ Sigmoid activation function.

$\boldsymbol{\Sigma}$ Covariance matrix.

Υ Number of possible architectures being considered during hyperparameter optimization.

$\boldsymbol{\phi}^{(i)}$ The parameter vector for model $p_{\tilde{\Omega}_i}$.

$\{\chi_\eta\}_{\eta\in\mathbb{N}^+}$ Sequence of binary symbols associated with a given context.

$\boldsymbol{\psi}$ Neural network input. One realization from $\boldsymbol{\Psi}$.

$\tilde{\boldsymbol{\psi}}$ Reconstructed neural network input from $\tilde{\boldsymbol{\omega}}$.

$\hat{\boldsymbol{\psi}}$ Reconstructed neural network input from $\hat{\boldsymbol{\omega}}$.

$\boldsymbol{\Psi}$ Random vector representing the inputs of a neural network.

$\tilde{\omega}_i$ One component of $\tilde{\boldsymbol{\omega}}$.

$\boldsymbol{\omega}$ Point in transformed domain.

$\tilde{\boldsymbol{\omega}}$ Point in transformed domain after the addition of noise. One realization from $\tilde{\boldsymbol{\Omega}}$.

$\hat{\boldsymbol{\omega}}$ Point in transformed domain after quantization. One realization from $\hat{\boldsymbol{\Omega}}$.

$\tilde{\Omega}_i$ One component of $\tilde{\boldsymbol{\Omega}}$.

$\tilde{\boldsymbol{\Omega}}$ Random vector representing a point in transformed domain after the addition of noise.

$\hat{\boldsymbol{\Omega}}$ Random vector representing a point in transformed domain after quantization.

Chapter 1

Introduction

1.1 Contextualization

Although not evident to everyone, data compression is one of the enablers of the modern world. Without data compression, communication would be much slower and much more expensive, web pages containing images or videos would take much longer to load with a much lower quality and the average computer would only be able to store a tiny portion of the data it stores today. We may also assume there would have been critical technological development delay due to the absence of data compression.

To put things into perspective, an uncompressed image file is, in general, three to five times larger than a JPEG file [1]. The gains of video compression, however, are much more striking: in general, an uncompressed video file may be 20 to 200 times larger than an H.264 compressed file [2], but these numbers could be much larger depending on the quality level. Figure 1.1 shows a comparison between an image in PNG format, which preserves the original quality, and the same image converted to JPEG with a quality factor of 75. The former is 558kB while the latter is only 41kB. The difference can only be noticed with a closer look.

Data compression simply provides much more efficiency in a variety of ways. It is able to reduce the costs associated with storage, transmission, streaming, data access,

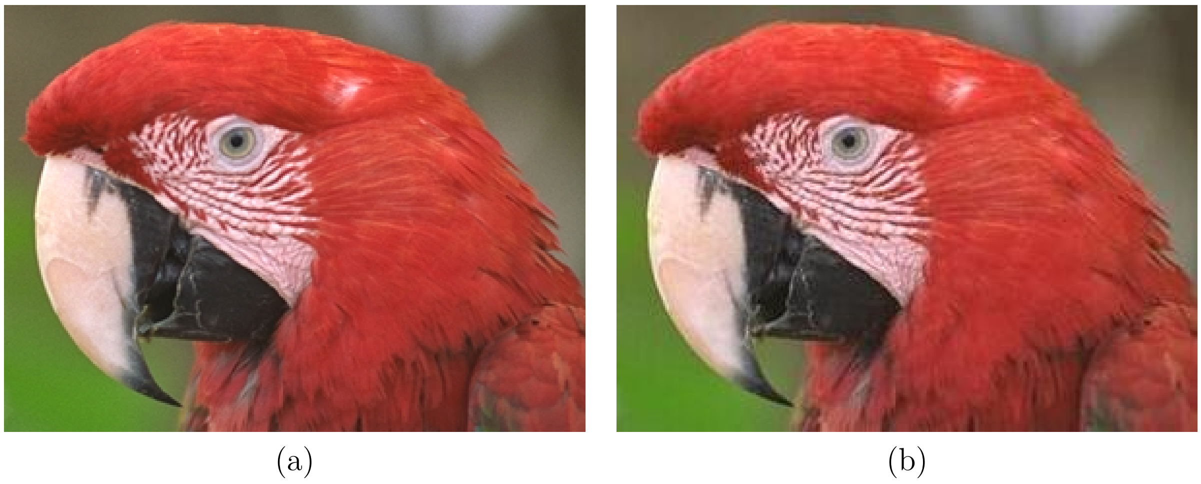


Figure 1.1: To the left, a portion of an image in PNG format and, to the right, the same portion with the image converted to JPEG with a quality factor of 75. The PNG file was downloaded from [3] and has 558kB. The image converted to JPEG has 41kB.

and many others, bringing considerable savings to everyone involved with these activities, from the large companies which develop new technologies, to the end users which consume them.

Therefore, it comes to no surprise that, in the recent years, with the newfound success of artificial neural networks in many areas of application, the data compression community has started making use of them. It is undeniable how neural networks increased the power of newly developed experimental data compressors. For example, in the Challenge on Learned Image Compression [4], for many years now, neural networks have surpassed the traditional coders in image compression. Also, in the Large Text Compression Benchmark [5], the main reference method uses neural networks for text prediction. And this is just to mention a few relevant data compression conferences.

However, the use of neural networks in data compression poses some additional challenges compared to other areas of application. Data compression, in general, is subject to much more stringent conditions than other areas. Neural networks owe its increasing success in several areas of application mainly to two reasons: the increase in availability of data which can be used for training; and the increase in model size and computational resources which can support these large sized models. Take the large-language models (LLMs) for example. Their success in text generation is largely due to a huge

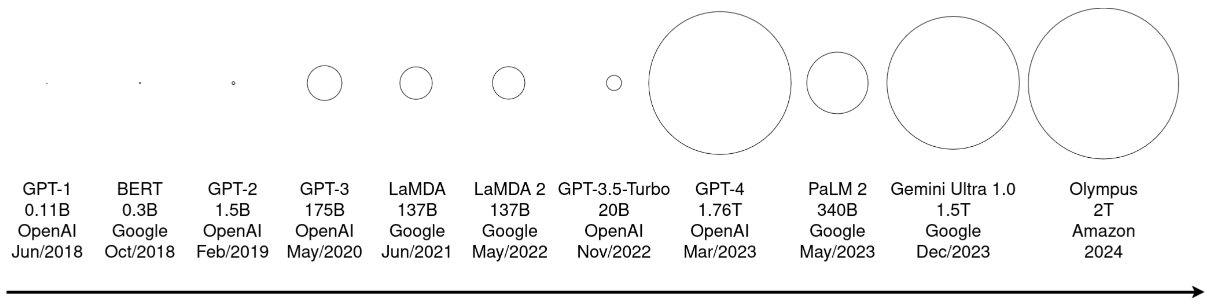


Figure 1.2: Evolution of the LLM model sizes over the years, with the model size measured in number of parameters. Based on data from [8].

increase in the model size, as the name implies, but it is also due to training models on a huge amount of content. That is why LLMs have primarily been developed by billionaire companies with outstanding computational resources. Figure 1.2 shows an evolution of the LLM model sizes over the years. It is clear that the model size has sharply increased. Nevertheless, one cannot put a huge model into a cellphone. Recent studies on learned video compression [6] have shown that most learned video codecs currently have a kilo-multiply-accumulate-operations per pixel (kMAC/pixel) between 1000 and 2000, as depicted in Figure 1.3 (a). A few codecs can reach the order of magnitude of hundreds of kMAC/pixel. However, the capabilities of a modern cellphone are closer to the order of 1 kMAC/pixel [7].

Other factor is peak memory usage. These learned codecs need to hold the equivalent of several frames in memory while running. This has implications for memory bandwidth requirements when they are stored in an external memory. Figure 1.3 (b) shows the peak memory for the same codecs of Figure 1.3 (a).

Matters such as these have made the consolidation of neural networks in data compression much harder than it has been in other areas. To this day, neural networks in data compression have mostly been present in the realm of academic research, and left out of commercial products and international standards, with only a few exceptions, for example [9],[10],[11]. Although they have shown great potential, there are still many aspects that require further development. Beyond the already mentioned reasons why they are not yet consolidated, other mentionable reasons are: the long time needed to develop

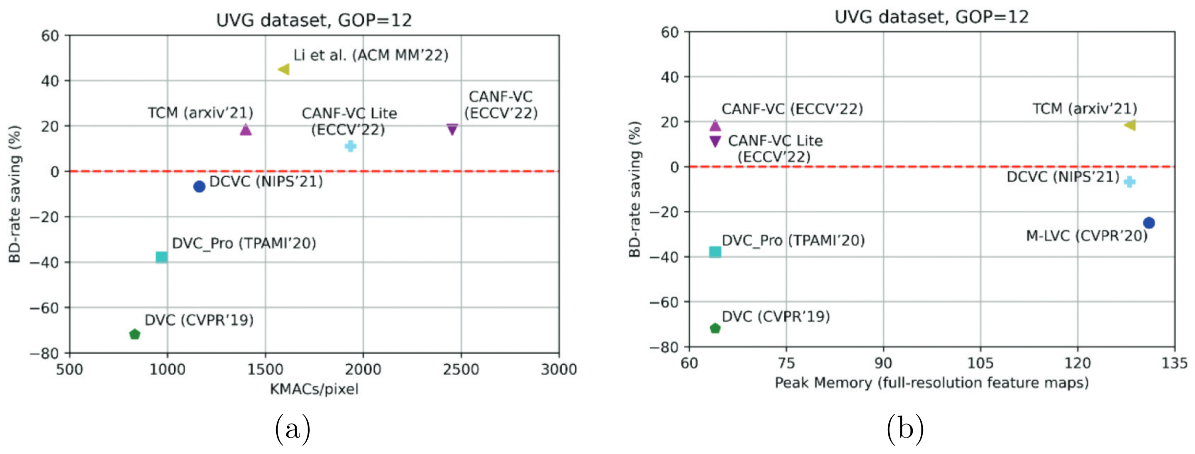


Figure 1.3: BD-rate savings versus kMAC/pixel and versus peak memory for several learned video codecs. Better compression is generally achieved with more arithmetic operations. Source: [6]. See the source for more details on the measures and labels.

new standards, and the time required for disseminating a new technology [12].

1.2 The problems we address

In this work, we address two main problems in neural-based data compression, namely online coding and architecture optimization. Both of these topics are currently active areas of research.

In order to be used in practical systems, in smartphones, in embedded devices, or in data streaming, there must be efficient and effective ways of achieving architecture designs that can fit the system time, power and computational resource restrictions, without having to resort too much on trial and error.

The traditional way of performing neural network design is by trial and error. This has traditionally been made through individual publications competing to improve the state-of-the-art. However, both for scientific and practical reasons, it is desireable to automate the design process. This can accelerate both the development of new architectures and the understanding of the problem.

Much more than simplifying the selection of an architecture that can fit into the constraints of a system, which would probably be done only once for that given system, what is intended is to find the best selection of architectures. However, there is an infinite number of possible architectures, and we cannot test all of them.

Figure 1.4 (a) shows the training costs of several learned codecs. To obtain these training costs, we combine kMAC/pixel data from [6], with estimation methods from [13]. We obtain the total training cost of a codec in number of floating-point operations (FLOPs) by assuming that: one MAC amounts to two FLOPs; the training set consists of 91701 videos, having 7 frames, each (Vimeo-90k septuplet dataset [14]); each frame is cropped to 256×256 pixels; one must train four networks, to operate in four quality settings; each network is trained during 50 epochs; the cost for training is twice the cost for inference (because of the forward and backward passes). By multiplying all those numbers ($2 \times 91701 \times 7 \times 256 \times 256 \times 4 \times 50 \times 2$) with the codec's kMAC/pixel metric, we obtain our estimate of the codec's total training cost in FLOPs. Next, we estimate the codec's total training cost in US dollars by first estimating the value of $\text{FLOPs} \cdot \text{second}^{-1} \cdot \text{dollar}^{-1}$ in the year that the associated paper was published. This is done assuming that: the value of GPU $\text{FLOPs} \cdot \text{second}^{-1} \cdot \text{dollar}^{-1}$ is multiplied by 10 every 8.17 years, and that its value in the year 2000 was 66 $\text{MFLOPs} \cdot \text{second}^{-1} \cdot \text{dollar}^{-1}$. This is based on analyzes from [15], using data from [16] and [17]. The data used to infer this trend went up to 2020 and was adjusted for inflation, therefore the units are in 2020 US dollars. As in [13], we multiply the initially estimated value in $\text{FLOPs} \cdot \text{second}^{-1} \cdot \text{dollar}^{-1}$ by 0.35, because we assume that the actual value of $\text{FLOPs} \cdot \text{second}^{-1}$ achieved during model training is 35% of the theoretical peak for the hardware. We multiply the value of $\text{FLOPs} \cdot \text{second}^{-1} \cdot \text{dollar}^{-1}$ by the estimated hardware replacement time in seconds, which we assume is the equivalent of two years. This gives an estimate of the amount of FLOPs per dollar. Finally, we divide the total training cost in FLOPs by the estimated value of FLOPs per dollar, to obtain the total training cost in dollars.

Figure 1.4 (b) also shows the time taken to train each codec, according to the authors of the codecs themselves. As can be seen from Figure 1.4, there is not only financial costs associated with the training of a neural network but also time investment. The

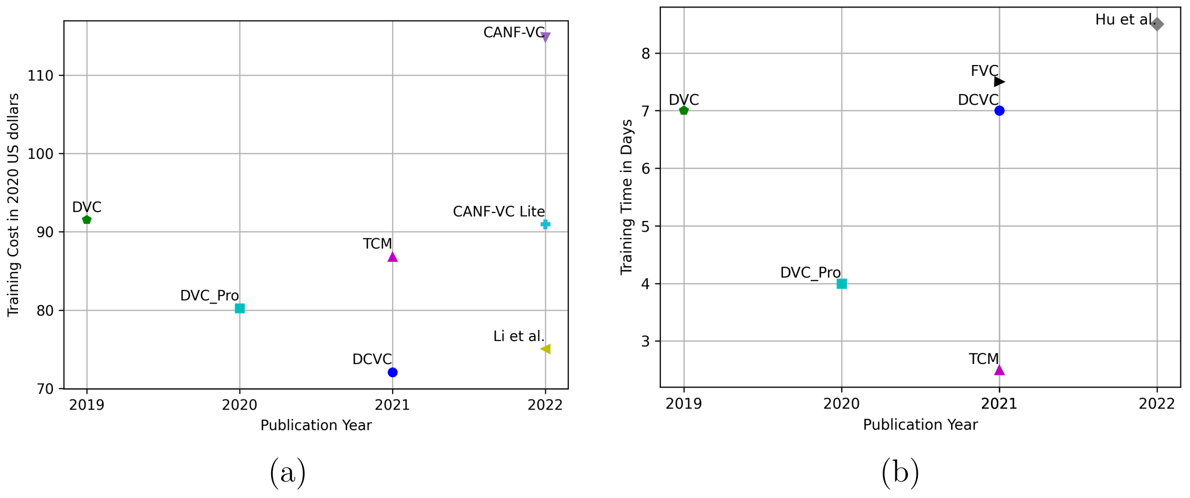


Figure 1.4: Training costs of several learned codecs in 2020 US dollars and in days of training. The training costs in dollars were obtained based on kMAC/pixel data from [6] and estimation methods from [13]. The training times were provided by the authors themselves on their papers. We use the same labels for the codecs as [6].

costs associated with a single network may not be too large, but they accumulate when training many networks.

To further demonstrate the importance of reducing the complexity of the neural networks used for data compression, Figure 1.5 shows the energy consumption of several learned codecs. Figure 1.5 (a) was obtained assuming that the energy cost of training is about 20% of the hardware cost [13]. That is, the energy cost is 20% of the cost shown in Figure 1.4 (a). We divide this number by the amount of pixels processed during training, which is $91701 \times 7 \times 256 \times 256$, and the number of network passes during training, which is $4 \times 50 \times 2$ (the number of different networks, times the number of epochs, times the number of forward and backward passes for one network update). Dividing the result by the average US energy price in 2020, which was approximately 13 cents per kWh [13], gives the value of kWh/pixel during execution of the network. Finally, considering that 1 kWh corresponds to $3.6 \cdot 10^6$J, this can be converted to the μJ/pixel value shown in Figure 1.5 (a). Figure 1.5 (b) was obtained by further assuming that videos of 256×256 pixels are displayed during 1 minute at 60 frames per second, and converting the units from μJ to Wh. The average energy consumption of these codecs is about 2 Wh per minute of video. Considering that a top Samsung cellphone has a battery capacity of

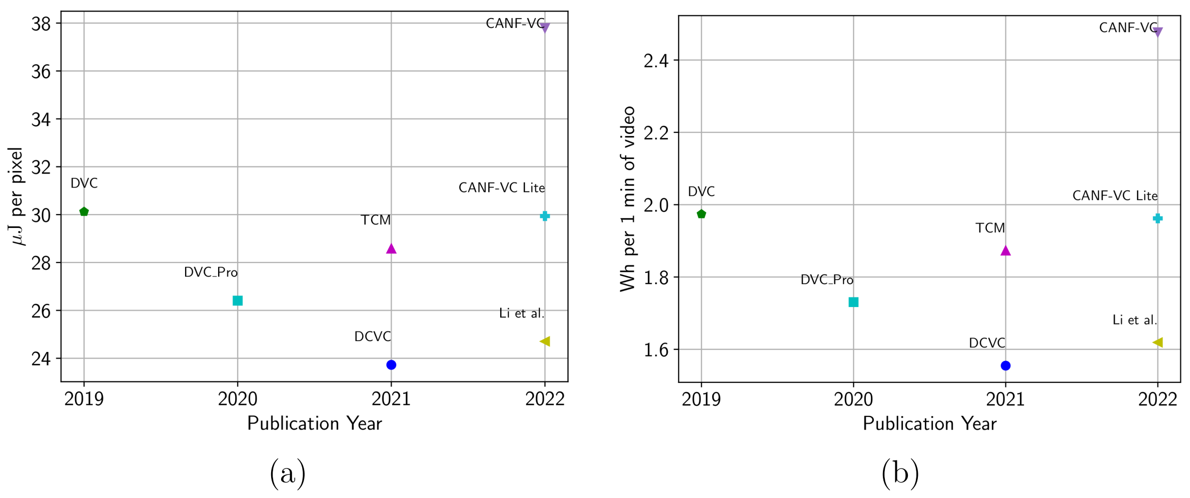


Figure 1.5: Energy consumption of several learned codecs in μJ/pixel and in Wh per 1 minute of video. We use the same labels for the codecs as [6].

about 5000 mAh, and that a lithium battery has a nominal voltage of 3.7V, the amount of energy this cellphone's battery is able to produce before recharging is 18.5 Wh. Such a cellphone running a modern learned codec would last only about 9 minutes without recharging. This shows how important it is to reduce as much as possible the complexity of the learned codecs.

Meanwhile, an often overlooked issue with modern neural-based data compression systems is that the models are mostly trained offline. Online models have been used in data compression for years for a variety of reasons. Among them is practicality. In general, training a model beforehand requires the gathering of a huge amount of data which is representative of the type of data being compressed. In some situations, however, this is not very practical or desirable. Take, for example, the case when a new type of data is just emerging. It may seem a little difficult to think of new types of data emerging nowadays, but it just happened recently. Point clouds is a type of data which has been popularized and only gained more attention in the recent years, prompting the creation of new compression standards specific to this type of data. You can predict that, gathering a huge amount of examples of this type of data initially was not very easy. Online models, on the other hand, can learn the statistics of the data while coding, and can adapt themselves to local changes in the signal statistics.

Other factor is that coding and decoding must be reproducible. Compression algorithms must be described in international norms and standards. In this way, coding and decoding can be run on different platforms, created by different companies. If one is going to create a standard based on pre-trained neural networks, how the neural network is to be described in the standard? Are the network weights to be included in the standard? This is not to say it can't be done, and there are standards being developed right now which will have to deal with these issues [10],[11]. If the model is trained online, it is not necessary to describe the weights in the standard, since they are learned from the data being encoded. Only a pseudo random number generator algorithm and a random seed used to initialize the network weights have to be described.

Of course, other issues may emerge from the use of online training, for example, the decoding time may considerably increase. With further technological development, or dedicated hardware, it may become feasible to use online-trained neural networks in data compression. In the long run, it may be possible to have fully automated and universal compression systems, which could learn to encode any type of data on the fly.

1.3 Overview of relevant literature

Neural networks have been investigated for both lossless and lossy data compression [18]–[30]. Most of these networks have been pre-trained, that is, trained offline on representative datasets. In this section we will analyze some of the previous literature on neural based data compression.

In lossless neural-based compression, a neural network is typically used as a probability model, often called a context model or entropy model, to drive an arithmetic coding (AC) system. Pre-trained neural networks have been used for context modeling to drive AC in various settings (e.g., [18]–[25]). In both [20] and [21], the authors replaced the original context models in H.265/HEVC, which were adaptive, with pre-trained neural networks. In [22] and [23], the authors proposed methods to progressively transmit voxelized point cloud geometries using AC, neural networks and octree. The idea is, given the occupied voxels in one level of the octree, transmit the occupancy of the children of those voxels in the next level. The probabilities of occupancy of the voxels were estimated by pre-trained neural networks and used to drive AC. In [18] and [19] the authors built language models using recurrent neural networks (RNNs) and used those language models to feed the AC for text compression.

In lossy neural-based compression, many works have followed the seminal work of [26], for example [27]–[30]. In this method, two neural networks jointly learn how to encode and decode a latent space representation which is quantized and transmitted using context modeling and AC, in a very similar way to how it is done in lossless neural-based compression. The method is an adaptation of the variational autoencoder [31], proposed in the field of variational inference, to the task of lossy data compression. It is a form of transform coding using neural networks. The subsequent works have mainly focused on improving the entropy model, for example, by: learning and transmitting priors on the parameters of the entropy model [27]; generalizing the entropy model to a Gaussian mixture model and including an autoregressive component [28]; and leveraging discretized Gaussian mixture likelihoods [29]. Other works have focused on extending the methods for video-compression [30].

Adaptive context modeling has previously been covered in part in [24] and [25], mostly for text compression. In [24], it was hinted that the off-line methods they proposed could be adapted by blocking the data and continuously retraining the model. In [25], a hybrid on-off-line method, DZip, was specifically proposed for sequential data such as text. Outside of the data compression literature, in the context of language modeling, [32] and [33] proposed methods to continuously adapt RNN pre-trained weights during evaluation.

There is a couple of recent works that tackle the problem of jointly optimizing rate, complexity, and distortion in neural compression [34], [35]. Both aim at controlling complexity through specific network hyperparameters. Works on neural network compression are also closely related, for example [36]–[39], because they intend to reduce the model size in bits, while keeping as much as possible the original performance of the network. Energy-constrained data compression also intend to reduce the complexity in Joules of the coder, be it neural or not, while keeping its original performance and has previously been studied in [40]–[44]. There is also a vast literature on rate-distortion optimization [45]–[49]. In a more recent work [12], instead of specifying fixed weights for rate and distortion in the loss function during the training of the neural network, a fixed rate is specified, allowing the more effective tracing of the lower convex hull of the rate-distortion points. In [50]–[52], the authors optimize rate and distortion in tree structured domains with pruning algorithms, but they are applied to non-neural coders.

There are also general methods for single-objective hyperparameter optimization (SOHPO) and multi-objective hyperparameter optimization (MOHPO) that can be used to search for optimum architectures in neural-based data compression [53]–[55]. SOHPO methods optimize a single objective, or a combination of multiple objectives reduced to a single objective, for example by a weighted sum. MOHPO methods look for the set of optimal solutions for all tradeoffs between objectives. MOHPO is in general a much harder problem and has been investigated to a much lesser extent than SOHPO. MOHPO has recently been reviewed in [53]. SOHPO has been reviewed for example in [54], [55]. Popular SOHPO methods include grid-search [56], random-search [56] and Bayesian Optimization [55], all of which have adaptations for MOHPO [53].

1.4 Objectives and dissertation layout

The main objective of this work is to investigate neural-based adaptive context modeling, while also tackling the problem of neural architecture search. Our main contribution is the proposal of a lossless binary coding scheme, coupled with the proposal of a MOHPO method which can be used in the neural network design.

Traditionally, adaptive context modeling uses a frequency counting method based on look-up tables (LUTs). In part, this work investigates the replacement of the LUTs by neural networks in adaptive context modeling. LUTs are used to count the frequencies of occurrence of all symbols given all contexts. The counts are usually updated after every new observed sample. By developing a neural-based drop-in replacement for LUTs, we provide a method of compression that, at the same time, combines the advantages of neural networks and adaptive context modeling. In order to achieve this goal, we first develop the method that can replace the LUT and then evaluate it on representative data. We measure the method's performance and compare it with other alternatives to properly assess its effectiveness.

This research also investigates how to design neural networks for data compression. There exists general methods for multi-objective optimization that can be used in data compression as well. However, there are particularities specific to data compression that could be explored to develop methods more effective for this area of application. In order to achieve this, we need to develop such methods, evaluate them objectively, and compare them with other available generic methods.

This work is organized as follows. In Chapter 2, we review the fundamentals of adaptive context modeling, which is necessary for the discussions on the next chapters. In Chapter 3, we review the basics of neural networks and hyperparameter selection. We formally introduce MOHPO, and present some state-of-the-art methods which are used to tackle this problem. Chapters 4 and 5 are devoted to our research problem. We divide our discussion into two parts. First, in Chapter 4, we present our neural-based method, which is able to replace the LUT in traditional adaptive context modeling, without delving into the neural network design process. We call our proposed method adaptive perceptron coding (APC). Then, in Chapter 5, we describe our algorithm for MOHPO in data compression, which we call greedy lower convex hull (GLCH). This work is concluded in Chapter 6, with a review of our contributions, and possible topics for future research.

Chapter 2

Data Compression Fundamentals

2.1 Preliminaries

Data compression is the science of shrinking data to a more compact form [57]. It falls into the area of information theory which is "the mathematical field dealing with the transfer of information from one location to another and with the storage of information for later retrieval and use" [58]. Data compression can be classified into two broad categories: lossless compression, in which the original data can be perfectly reconstructed , and lossy compression, in which the data goes through higher compression but is also distorted.

In information theory, a general communication system is subdivided into five parts: a source, which generates a message; a transmitter, which turns the message into a signal for transmission; a channel, which conveys the signal; a receiver, which reconstructs the message; and a destination, which acquires the message [58]. The source outputs symbols, or letters. The conjunction of several letters forms messages. The set of possible letters of the source is called its alphabet. Letters are also often called samples.

Information theory answers two major problems in communication: how to efficiently represent messages, and how to reliably transmit messages over a noisy channel. Although the second problem is also very important in its own right, we will not be covering it in this work. In the first front, information theory provides a lower limit for the expected number of bits per symbol when transmitting messages without loss of information, which is the entropy.

Because of the way digital systems work, information is usually measured in bits. The process of assigning binary sequences, or codewords, to elements of an alphabet is called coding, and a code is a set of codewords associated with an alphabet [57]. Encoding is the conversion of symbols to their codewords, and decoding is the inverse of encoding. We want the original sequence to be recovered with certainty, therefore we are only interested in uniquely decodable codes, that is, each sequence of codewords can only be decoded in one way. Source coding is the type of coding done at the source level with the purpose of removing redundancy and achieving compression [58].

Entropy is a measure of uncertainty. For example, the entropy of the toss of a fair coin is larger than the entropy of the toss of an unfair coin, because the uncertainty about the outcome of the fair coin is larger. Information, in its turn, is a reduction in uncertainty. In a sequence of 10 coin tosses, knowing the outcomes of the first 3 tosses reduces the uncertainty about the final outcome of the 10 tosses, unless one of the sides of the coin was never possible in the first place. One of the important aspects about information in information theory is that it has nothing to do with semantics.

The amount of information gained with the occurrence of an event A is called its self-information. If its probability is $P(A)$, then its self-information can be measured in bits by

$$\log_2(1/P(A)) = -\log_2(P(A)). \tag{2.1}$$

The choice of this expression is not arbitrary, and it can be derived from a set of properties expected from a measure of information [57]. This definition is intuitive as well. The information gained by the occurrence of an event with probability $P(A) = 1$, or in other words $\log_2(1/1) = 0$, is lower than the information gained by the occurrence of an event with probability $P(A) = 0.5$, or in other words $\log_2(1/0.5) = 1$.

The definition of entropy relies on the concepts of self-information and random process. A source in a communication system can be seen as a random experiment $\mathcal{S}$ with sample space $\mathcal{X}$. The sample space is the set of all possible individual outcomes from the random experiment. Therefore, the sample space is also the alphabet. The random experiment observed over time forms a random process $\{X_n\}_{n\in\mathbb{N}^+}$, which is a sequence of indexed random variables. In this case we assume, without loss of generality, that the index n is a positive, integer-valued, time instant. A stochastic process is stationary when the joint distribution of any subset of the random variables is invariant with respect to shifts in the time index [59] and it is independent and identically distributed (iid) when any subset of the random variables are independent and the marginal distributions are the same [60]. If the process is iid, the entropy of the source can be characterized by a single random variable from the random process $\{X_n\}_{n\in\mathbb{N}^+}$, for example, X_1. More specifically, the entropy of the iid source $\mathcal{S}$ can be obtained by the expected self-information of X_1:

$$H(X_1) = -\sum_{x_1\in\mathcal{X}} P(x_1)\log_2(P(x_1)). \tag{2.2}$$

However, this is only if the samples are iid. More generally, the entropy of a stationary source can be defined as the average self-information per sample of longer and longer sequences generated by the source [57]. We represent the sequence of the first n random variables of the random process as $\mathbf{X}^{(n)} = \{X_1, X_2, ..., X_n\}$, one possible realization of such sequence as $\mathbf{x}^{(n)} = \{x_1, x_2, ..., x_n\}$, and all possible realizations as $\mathcal{X}^{(n)}$. The average self-information of the sequences of length n is

$$G_n = -\sum_{\mathbf{x}^{(n)}\in\mathcal{X}^{(n)}} P(\mathbf{x}^{(n)})\log_2(P(\mathbf{x}^{(n)})) \tag{2.3}$$

and the average self-information per sample is

$$H_n = \frac{G_n}{n}. \tag{2.4}$$

The entropy of the stationary source is

$$H(\mathcal{S}) = \lim_{n \to \infty} H_n, \tag{2.5}$$

where stationarity is a sufficient condition for the limit to exist [59]. The entropy of the source represents a lower bound for the number of bits per sample, also called the average codeword length, of lossless compression schemes. Coding algorithms that aim at this lower bound are referred to as entropy coding algorithms.

2.2 Arithmetic Coding

We now turn our focus to source coding and how we can achieve a coding such that there is no loss of information and the average codeword length is close to the entropy of the source.

A code is said to be nonsingular if every symbol is mapped into a different codeword. It is called uniquely decodable if all sequences of symbols are mapped to different code strings, and it is called a prefix code if no codeword is a prefix of any other codeword. A prefix code is also called an instantaneous code because the end of a codeword is immediately recognizable and therefore a codeword can be decoded without reference to future codewords. Data compression can be achieved by assigning short codewords to the most frequent symbols, and longer codewords to the less frequent symbols. It is possible to show that the broader class of uniquely decodable codes does not offer better choices of codeword lengths than the narrower class of prefix codes [59].

It is intuitive that we cannot assign short codewords to all source symbols. Consider a source with set of possible outcomes composed of S elements. The codeword lengths $\ell_1, \ell_2, ..., \ell_S$ of a prefix code must satisfy the Kraft inequality [59]:

$$\sum_i 2^{-\ell_i} \leq 1. \tag{2.6}$$

Also, given a set of codeword lengths that satisfy this inequality, there exists a prefix code with these word lengths.

Let p_i be the probability associated with the i-th possible symbol. We want to find the prefix code with the minimum expected length that satisfies the Kraft inequality. In other words, we want to minimize:

$$\begin{cases} \min_{\ell_1,\ell_2,\ldots,\ell_S} \sum_i p_i \ell_i \\ s.t. \sum_i 2^{-\ell_i} \leq 1 \end{cases} \tag{2.7}$$

whose solution according to calculus is given by $\ell_i^* = -\log_2 p_i$. But since the ℓ_i must be integers, we will not always be able to set these codeword lengths. We can do this only when $p_i = 2^{-\ell_i}$ for integer ℓ_i, that is, when the distribution is dyadic. The optimal code under these restrictions can be obtained by finding the dyadic distribution that is closest to the distribution of the symbols [59]. However, a good approximation can be easily obtained by rounding up the fractional optimal lengths:

$$\ell_i = \left\lceil \log_2 \frac{1}{p_i} \right\rceil . \tag{2.8}$$

This approximation satisfies the Kraft inequality because

$$\sum 2^{-\lceil \log_2 \frac{1}{p_i} \rceil} \leq \sum 2^{-\log_2 \frac{1}{p_i}} = \sum p_i = 1 \tag{2.9}$$

and is within 1 bit from the expected self-information because

$$\log_2 \frac{1}{p_i} \leq \ell_i < \log_2 \frac{1}{p_i} + 1, \tag{2.10}$$

therefore

$$-\sum p_i \log_2 p_i \leq \sum p_i \ell_i < -\sum p_i \log_2 p_i + 1. \tag{2.11}$$

A practical algorithm which has similar codeword lengths is the Shannon-Fano-Elias

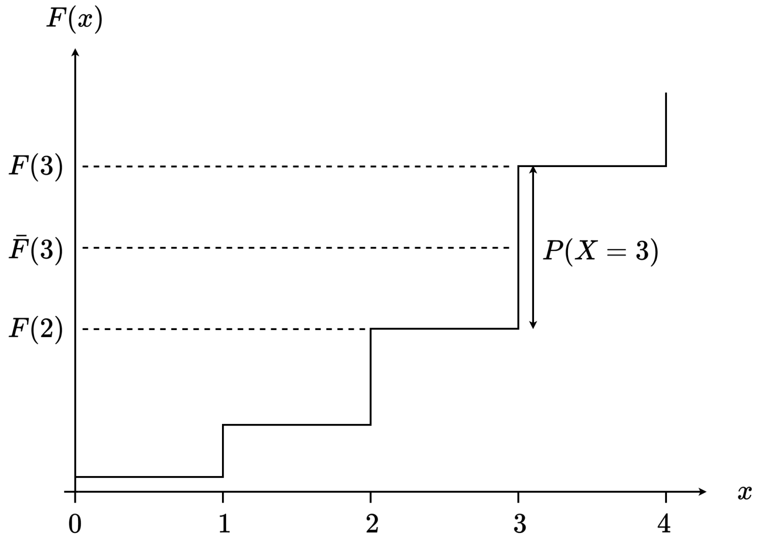


Figure 2.1: Cumulative distribution function of a discrete random variable X with sample space $\{0, 1, 2, 3, 4\}$. $\bar{F}(x)$ is the midpoint of the step corresponding to x. Based on figure 5.5 of [59].

coding algorithm. Assume that the alphabet is $\{0, 1, ..., m\}$, where $m = S - 1$. The cumulative distribution function of a discrete random variable X representing one sample is

$$F(x) = \sum_{a \leq x} P(a). \tag{2.12}$$

The cumulative distribution function of X looks like a staircase whose sizes of the steps are $P(x)$, as shown in Figure 2.1. The following modification of the cumulative distribution function

$$\bar{F}(x) = \sum_{a < x} P(a) + \frac{1}{2}P(x) \tag{2.13}$$

gives the midpoint of the step corresponding to x. If all $P(x)$ are greater than zero, then $\bar{F}(x)$ can be used as a code for x, since $\bar{F}(a) \neq \bar{F}(b)$ if $a \neq b$. More interestingly, it can be shown that $\bar{F}(x)$ truncated to $\ell(x) = \left\lceil \log_2 \frac{1}{P(x)} \right\rceil + 1$ bits, represented by $\lfloor \bar{F}(x) \rfloor_{\ell(x)}$, is also guaranteed to be within the step corresponding to x [59]. Therefore $\lfloor \bar{F}(x) \rfloor_{\ell(x)}$ can also be used as a code for x. However, because of the additional 1 bit per symbol, the Shannon-Fano-Elias coding expected codeword length is within 2 bits from the entropy, instead of only 1.

So far we have seen that, when encoding a symbol, the codeword lengths $\ell_i = \left\lceil \log_2 \frac{1}{p_i} \right\rceil$ give an expected codeword length at most 1 bit from the expected self-information. The overhead per symbol can be reduced by encoding sequences of symbols instead of individual symbols. This way the extra bit is spread out over many symbols. Then, assuming the set of possible symbols is $\mathcal{X}^{(n)}$, taking the inequality in (2.11), replacing p_i and ℓ_i with $P(\mathbf{x}^{(n)})$ and $\lceil \log_2(1/P(\mathbf{x}^{(n)})) \rceil$, and dividing by n we get:

$$-\frac{1}{n} \sum_{\mathbf{x}^{(n)} \in \mathcal{X}^{(n)}} P(\mathbf{x}^{(n)}) \log_2 P(\mathbf{x}^{(n)}) \leq \frac{1}{n} \sum_{\mathbf{x}^{(n)} \in \mathcal{X}^{(n)}} P(\mathbf{x}^{(n)}) \left\lceil \log_2 \left(\frac{1}{P(\mathbf{x}^{(n)})} \right) \right\rceil$$
$$< -\frac{1}{n} \sum_{\mathbf{x}^{(n)} \in \mathcal{X}^{(n)}} P(\mathbf{x}^{(n)}) \log_2 P(\mathbf{x}^{(n)}) + \frac{1}{n} \quad (2.14)$$

Therefore by using large block lengths n we can achieve an expected codeword length per symbol arbitrarily close to the entropy.

Consider an infinite sequence of random variables $X_1, X_2, ...$ with alphabet $\{0, ..., m\}$. For any outcome $x_1, x_2, ...$ we can place 0 and a dot in front of the sequence and consider it as a real number of base $m+1$ between 0 and 1. We can treat this sequence as representing an interval $[0.x_1x_2...x_n000..., 0.x_1x_2...x_nmmm...)$, or equivalently, $[0.x_1x_2...x_n, 0.x_1x_2...x_n + (\frac{1}{m+1})^n)$. This is the set of infinite sequences that start with $0.x_1x_2...x_n$. It is possible to show that the cumulative distribution function forms an invertible mapping from infinite source sequences to incompressible infinite binary sequences [59]. Under this transform, this interval gets mapped into another interval, $[F_X(0.x_1x_2...x_n), F_X(0.x_1x_2...x_n + (\frac{1}{m+1})^n))$, whose length is equal to $P(x_1, x_2, ..., x_n)$, the integral of the probability densities of all infinite sequences that start with $0.x_1x_2...x_n$. Similarly to Shannon-Fano-Elias coding, the binary representation of the midpoint of the interval can be truncated to $\left\lceil \log_2 \frac{1}{P(x_1, x_2, ..., x_n)} \right\rceil + 1$ bits, and the resulting number is still guaranteed to be within the limits of the interval. Therefore it can be used as a code for the sequence $x_1x_2...x_n$.

It is not necessary to transmit the whole sequence all at once. As more and more symbols are seen, the interval goes from $[0, 1)$, to $[F_X(0.x_1), F_X(0.x_1 + \frac{1}{m+1}))$, then to

$[F_X(0.x_1x_2), F_X(0.x_1x_2 + (\frac{1}{m+1})^2))$ and so on. At first sight, it may seem that this would require infinite precision arithmetic, since the number of digits in the top and bottom ends of the interval keep increasing. But arithmetic coding provides a way to do this with finite precision. As more symbols are seen, more leading digits from the top and bottom ends of the interval become equal. As soon as the two ends of the interval agree about some bits, we can output these bits and shift them out of the calculation. This way all calculations can be made with finite precision.

Figure 2.2 illustrates through an example how arithmetic coding may work in practice, assuming that the sequence of symbols is independent and identically distributed. From a practical point of view, the basic idea of arithmetic coding is to start with an interval from 0 to 1, allocate portions of the interval to the symbols according to their probabilities, restrict the interval based on the symbol seen and repeat the process with the restricted interval. The compressed representation of the sequence is a binary fractional number in the middle of the final interval, truncated to the minimum amount of bits such that the number is still guaranteed to be in the interval. This number is also called the tag for the sequence. It is not necessary to wait until the end of the process to start transmitting the tag. For example, in the example of Figure 2.2, after encoding AB, the bits 0.01 can already be transmitted, because the top and bottom ends of the interval already agree on those bits.

The decoding process basically mimics the encoding process. At the decoder, the symbols are reconstructed basically by retracing the steps done by the encoder with the help of the tag. Starting with the 0 to 1 interval, the decoder subdivides the interval between the symbols according to their probabilities the same way as the encoder, identifies the symbol based on which range the tag falls in, and repeats the process with the new interval until the last symbol has been reconstructed. This process is illustrated in Figure 2.3.

We have showed how arithmetic coding works when the samples are iid. However, when the samples are not iid, the process is basically the same. The only difference is

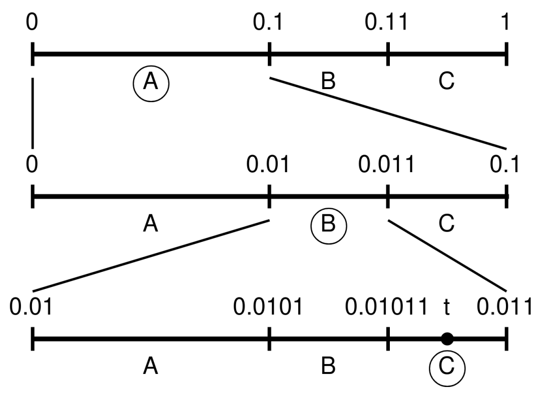


Figure 2.2: Illustration of the arithmetic coder encoding process. Numbers are represented in base 2. In this example, the symbols are A,B and C with probabilities $P(A) = 0.5 = (0.1)_2$ and $P(B) = P(C) = 0.25 = (0.01)_2$. The encoded sequence is ABC, and the tag is $(0.010111)_2$. The fractional part of the tag is already truncated to the correct amount of bits since the length of the final interval is $(0.00001)_2$ and $\lceil -\log_2((0.00001)_2) \rceil + 1 = 6$ bits. Based on Fig. 2 of [18].

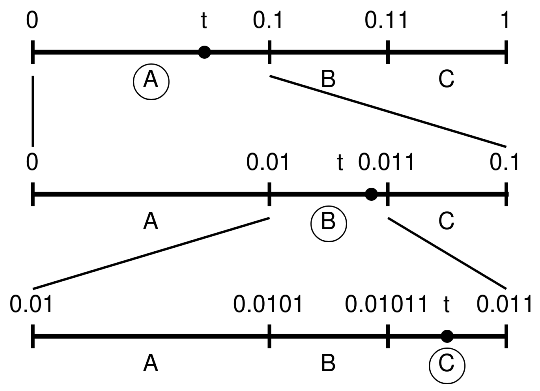


Figure 2.3: Illustration of the arithmetic coder decoding process. Numbers are represented in base 2. In this example, the symbols are A,B and C with probabilities $P(A) = 0.5 = (0.1)_2$ and $P(B) = P(C) = 0.25 = (0.01)_2$. The value of the tag is $(0.010111)_2 = 0.359375$ and the decoded sequence is ABC. Based on Fig. 2 of [18].

that the portions of the intervals assigned to the different symbols keep changing in size. Figure 2.4 illustrates the arithmetic encoding process when the samples are not iid.

The decoder must reproduce the same sequence of partitions produced by the encoder, reconstructing the symbols and restricting the interval based on the portion of the interval that the tag falls in [1]. Figure 2.5 illustrates the arithmetic decoder when the samples are not iid.

[1]Note that to correctly decode the sequence, it is essential that the decoder has access not only to the bitstream generated by the encoder, but also to the exact probabilities used during encoding, and in the same order.

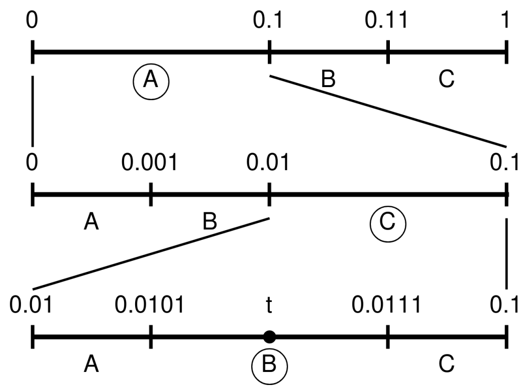


Figure 2.4: Illustration of the arithmetic coder encoding process when the samples are not iid. In this example, the probabilities of the symbols A,B,C change from 0.5,0.25,0.25 to 0.25,0.25,0.5 to 0.25,0.5,0.25, the encoded sequence is ACB and the tag is $(0.0110)_2$. The fractional part of the tag is already truncated to the correct amount of bits since the length of the final interval is $(0.001)_2$ and $\lceil -\log_2((0.001)_2)\rceil + 1 = 4$ bits. Based on Fig. 2 of [18].

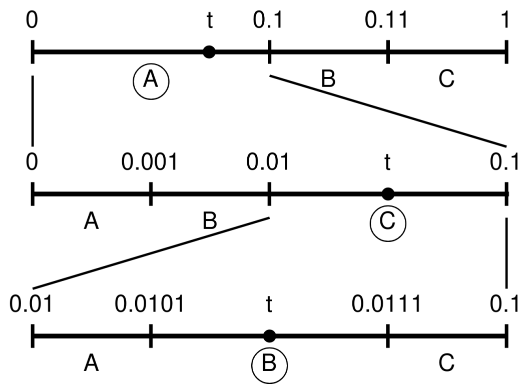


Figure 2.5: Illustration of the arithmetic coder decoding process when the samples are not iid. In this example the probabilities of the symbols A,B,C change from 0.5,0.25,0.25 to 0.25,0.25,0.5 to 0.25,0.5,0.25, the tag is $(0.0110)_2 = 0.375$ and the decoded sequence is ACB. Based on Fig. 2 of [18].

In summary, the arithmetic encoder can be seen as a black-box in which you enter data in raw format and the probabilities of the symbols at each instant, and receive in return the compressed sequence, as illustrated in Figure 2.6. Similarly, the arithmetic decoder can be seen as a black-box in which you enter data in compressed format and the probabilities of the symbols at each instant, and receive in return the uncompressed sequence.

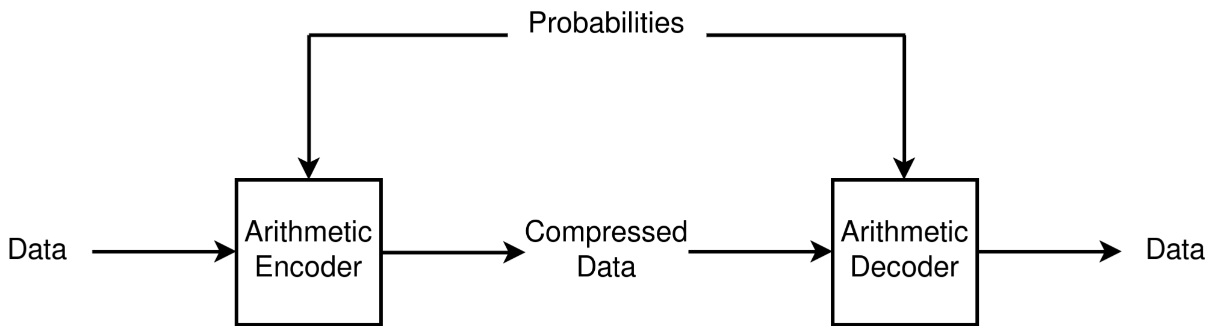


Figure 2.6: Schematic representation of the arithmetic encoder and the arithmetic decoder. Given a stream of symbols in uncompressed/compressed format and their probabilities, the arithmetic encoder/decoder returns the compressed/uncompressed sequence.

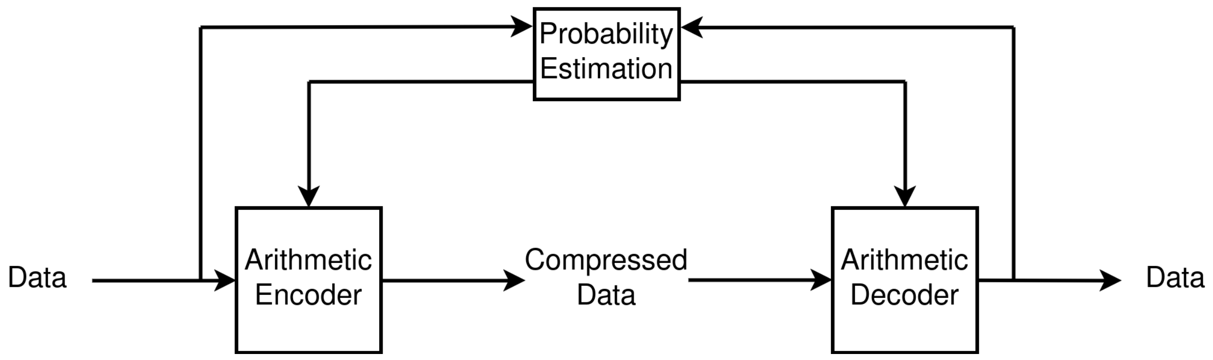


Figure 2.7: A more complete representation of a compressor and of a decompressor based on arithmetic coding. A probability estimation step is required to obtain the probabilities used with arithmetic coding.

2.3 Probability Estimation

In our previous discussion, we have intentionally left out a major component which is part of any data compressor based on arithmetic coding. The piece that is missing, as depicted in Figure 2.7, is a probability model of the source. A probability model is a mapping from events to probabilities. Models are mathematically represented as probability distributions. However, the distributions used in probability models have a distinguishing property. They are particularly made to approximate an unknown distribution and typically have a parametric form with parameters fitted to the data. A model whose probabilities do not change with the position in the sequence is called a static model, while a model whose probabilities do change is called an adaptive model [57].

The whole construction of arithmetic coding is based on the availability of the symbol probabilities. In general, it is not possible to precisely know them. Therefore, they must be estimated somehow. In theory, they could even be obtained by human intuition [61]. However, in practice probability models are built based on empirical observations. If the model is built prior to encoding, using the data that will be encoded, or any other data, this is called forward estimation or coding. If the model is built during encoding or decoding using the previously encoded or decoded samples, this is called backward estimation or coding. A model built with backward estimation is necessarily an adaptive model, while a model built with forward estimation may be static or adaptive, depending, for instance, if context-modeling is used (Section 2.4).

From Section 2.2 we know that arithmetic coding is capable of getting arbitrarily close to the entropy as the number of coded samples increases. Let $p_{\mathbf{X}^{(n)}}$ denote a model for the joint probability distribution of n samples from the source. Assume that this is the correct model. If this model is used with arithmetic coding, the average codeword length approaches

$$\lim_{n\to\infty} -\frac{1}{n} \sum_{\mathbf{x}^{(n)}\in\mathcal{X}^{(n)}} p_{\mathbf{X}^{(n)}}(\mathbf{x}^{(n)}) \log_2(p_{\mathbf{X}^{(n)}}(\mathbf{x}^{(n)})). \tag{2.15}$$

However, what happens in the most common case where the correct model is not known? If a wrong model $q_{\mathbf{X}^{(n)}}$ is used instead of $p_{\mathbf{X}^{(n)}}$, then the average codeword length actually gets closer to

$$\lim_{n\to\infty} -\frac{1}{n} \sum_{\mathbf{x}^{(n)}\in\mathcal{X}^{(n)}} p_{\mathbf{X}^{(n)}}(\mathbf{x}^{(n)}) \log_2(q_{\mathbf{X}^{(n)}}(\mathbf{x}^{(n)})). \tag{2.16}$$

The expression $-\sum p\log q$, where p and q are arbitrary probability distributions, is referred to as the cross-entropy of the distribution q relative to the distribution p, and is compactly represented as $H(p,q)$. With straightforward manipulation, the cross-entropy can be rewritten as

$$H(p,q) = -\sum p\log q = -\sum p\log p + \sum p\log\frac{p}{q}, \tag{2.17}$$

where $\sum p\log(p/q)$ is referred to as the Kullback-Leibler divergence (KLD) from q to p, and is also represented as $D_{KL}(p||q)$. The expression $-\sum p \log p$ is the entropy for the probability distribution p and is sometimes denoted as $H(p)$. With this in mind, the expression in (2.16) can be rewritten as

$$\lim_{n\to\infty} \frac{1}{n} \left(H(p_{\mathbf{X}^{(n)}}) + D_{KL}(p_{\mathbf{X}^{(n)}}||q_{\mathbf{X}^{(n)}})\right). \tag{2.18}$$

That is, the cost of compressing the data using a wrong model $q_{\mathbf{X}^{(n)}}$ instead of $p_{\mathbf{X}^{(n)}}$ is determined by the KLD from $q_{\mathbf{X}^{(n)}}$ to $p_{\mathbf{X}^{(n)}}$. The minimum of the expression in (2.18) is the entropy and it is achieved when $q_{\mathbf{X}^{(n)}}$ is equal to $p_{\mathbf{X}^{(n)}}$ or, equivalently, when the KLD is zero.

When talking of probability estimation, a very common method is maximum likelihood estimation (MLE). Let $\mathbf{x}^{(N)} = \{x_1, x_2, ..., x_N\}$ be observed data from the joint probability distribution of N random variables. Assume that we do not know the distribution, but we know that it is from a specific class of probability distributions, for example, that it is a multivariate normal distribution. Let the distribution parameters be represented by a random vector $\mathbf{\Theta}$, and a particular realization by $\boldsymbol{\theta}$. The likelihood function is the joint probability of the observed data given the distribution parameters $\boldsymbol{\theta}$, or $\mathcal{L}(\boldsymbol{\theta}|\mathbf{x}^{(N)}) = p_{\mathbf{X}^{(N)}|\mathbf{\Theta}}(\mathbf{x}^{(N)}|\boldsymbol{\theta})$. The maximum likelihood estimate is the argument $\boldsymbol{\theta}$ that maximize the likelihood function.

Assume a family of probability distributions in which the model parameters $\boldsymbol{\theta}$ are the probabilities of the outcomes themselves. This family of distributions is capable of representing any discrete probability distribution. In this case, the relative frequencies form the maximum likelihood estimate for the observed data. Without loss of generality, assume that the samples are iid. If they are not, and the probabilities depend on a few previous samples, or the context, as we will discuss more in Section 2.4, the same conclusions are valid for each context individually. Assuming iid samples, the joint probability of the data is $p_{\mathbf{X}^{(N)}|\mathbf{\Theta}}(\mathbf{x}^{(N)}|\boldsymbol{\theta}) = \prod_{n=1}^{N} p_{X|\mathbf{\Theta}}(x_n|\boldsymbol{\theta})$. This expression is maximized when $p_{X|\mathbf{\Theta}}(x_n|\boldsymbol{\theta}) = \kappa(x_n)/N$, where $\kappa(x_n) = |\{i : 1 \le i \le N \text{ and } x_i = x_n\}|$ is the absolute frequency of the symbol x_n. This is the case because any distribution with different probabilities than the relative frequencies would have lower chances of having generated that data.

Other way of verifying this is by realizing that there is an equivalence between maximizing the likelihood function and minimizing the average codeword length of an ideal arithmetic coder driven by a matching probability distribution [62]. Since the average codeword length would be $-\log_2(p_{\mathbf{X}^{(N)}|\mathbf{\Theta}}(\mathbf{x}^{(N)}|\boldsymbol{\theta}))/N$ and the likelihood is $p_{\mathbf{X}^{(N)}|\mathbf{\Theta}}(\mathbf{x}^{(N)}|\boldsymbol{\theta})$, the average codeword length would be minimized when the likelihood is maximized. The average codeword length is given by

$$-\frac{1}{N}\log_2 p_{\mathbf{X}^{(N)}|\mathbf{\Theta}}(\mathbf{x}^{(N)}|\boldsymbol{\theta}) = -\frac{1}{N}\log_2 \prod_{n=1}^{N} p_{X|\mathbf{\Theta}}(x_n|\boldsymbol{\theta}) = \\ -\sum_n \frac{1}{N}\log_2(p_{X|\mathbf{\Theta}}(x_n|\boldsymbol{\theta})) = -\sum_{x\in\mathcal{X}} \frac{\kappa(x)}{N}\log_2(p_{X|\mathbf{\Theta}}(x|\boldsymbol{\theta})) \quad (2.19)$$

where $\mathcal{X}$ is the set of all possible outcomes and x is one possible outcome from the random variable X. This expression is the cross-entropy between $\kappa(x)/N$ and $p_{X|\mathbf{\Theta}}(x|\boldsymbol{\theta})$, which we know is minimized when their KLD is zero, or in other words when $p_{X|\mathbf{\Theta}}(x|\boldsymbol{\theta}) = \kappa(x)/N$.

This is also true even when there is dependence among samples. Consider, for simplicity, that each sample depends only on the sample immediately before it. The joint probability of the data is $p_{\mathbf{X}^{(N)}|\mathbf{\Theta}}(\mathbf{x}^{(N)}|\boldsymbol{\theta}) = \prod_{n=1}^{N} p_{X|X',\mathbf{\Theta}}(x_n|x_{n-1},\boldsymbol{\theta})$ [2], which is maximized when $p_{X|X',\mathbf{\Theta}}(x_n|x_{n-1},\boldsymbol{\theta}) = \kappa(x_n,x_{n-1})/N(x_{n-1})$, where $\boldsymbol{\theta}$ represents the probability model parameters, $N(x_{n-1})$ is the number of occurrences of x_{n-1} or $|\{i : 0 \leq i < N$ and $x_i = x_{n-1}\}|$ and $\kappa(x_n,x_{n-1})$ is the number of occurrences of the pair x_n,x_{n-1}, or $|\{i : 0 < i \leq N$ and $x_i, x_{i-1} = x_n, x_{n-1}\}|$ or the absolute frequency of x_n given x_{n-1}. The

[2]It simplifies the notation and the calculations if we extrapolate the data and assume it equal to a specific symbol s_0 outside of the observed support, so for example $x_0 = s_0$. This may be a new symbol added to the set of possible symbols or an already existing symbol.

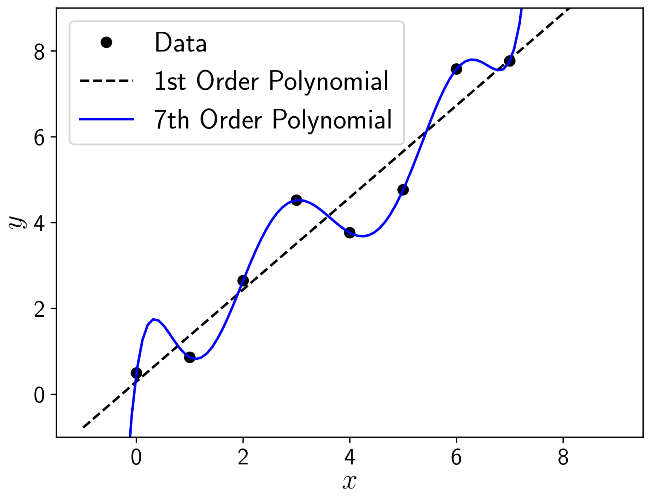


Figure 2.8: Example of overfitting. The data is linear with added noise. A polynomial of degree 7 perfectly fits the observed data, but performs poorly on unobserved data.

average codeword length of an ideal arithmetic coder would be given by

$$\begin{aligned} -\frac{1}{N}\log_2 p_{\mathbf{X}^{(N)}|\mathbf{\Theta}}(\mathbf{x}^{(N)}|\boldsymbol{\theta}) &= -\frac{1}{N}\sum_{n=1}^{N}\log_2 p_{X|X',\mathbf{\Theta}}(x_n|x_{n-1},\boldsymbol{\theta}) \\ &= -\sum_{x'\in\mathcal{X}}\frac{N(x')}{N}\left(\sum_{x\in\mathcal{X}}\frac{\kappa(x,x')}{N(x')}\log_2 p_{X|X',\mathbf{\Theta}}(x|x',\boldsymbol{\theta})\right) \end{aligned} \tag{2.20}$$

The expression in parenthesis is the cross-entropy of $\kappa(x,x')/N(x')$ and $p_{X|X',\mathbf{\Theta}}(x|x',\boldsymbol{\theta})$, which is minimized when $p_{X|X',\mathbf{\Theta}}(x|x',\boldsymbol{\theta}) = \kappa(x,x')/N(x')$.

When performing MLE, care must be taken to avoid overfitting [63]. Overfitting is a common problem encountered in mathematical models in general which is when the model performs well on observed data, but poorly on unobserved data. An example of overfitting is shown in Figure 2.8 in the setting of polynomial fitting. The larger the order of the polynomial, the lower is the error on the observed data, also called the bias or the approximation error, but the larger is the error on the unobserved data, also called the variance or the generalization error. This is known as the bias-variance tradeoff [64].

Back to the MLE problem, we would like to have a good estimate of the true probability distribution. Except in particular situations, we do not want a model which explains the data perfectly, we want a model with a good generalization performance.

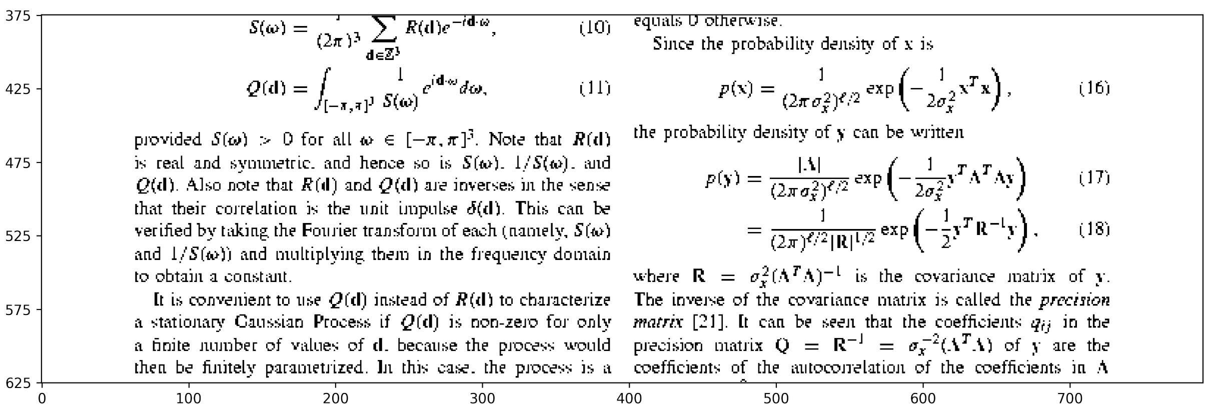
2.4 Context-based Probability Estimation

As we have seen in previous sections, entropy coding is based on assigning shorter codewords to the most frequent symbols, or the symbols with the highest probabilities, and longer codewords to the least frequent symbols, or the symbols with the lowest probabilities. This suggests that higher data compression can be achieved the more unbalanced is the set of probabilities.

In fact, the entropy is smaller when there are only a few symbols with high probabilities, and all others have low probabilities, compared to when all symbols have similar probabilities. Consider for example the tosses of a fair and an unfair coins. The outcomes have equal probabilities with the fair coin, while one has 75% probability and the other has 25% probability with the unfair coin. The entropy in the first case is $0.5\log_2(1/0.5) + 0.5\log_2(1/0.5) = 1$, and in the second case is $0.75\log_2(1/0.75) + 0.25\log_2(1/0.25) \approx 0.81$.

Therefore, if it were possible to make the probabilities of the symbols more unbalanced, we would be able to achieve better compression. The actual entropy of the source is a fixed and, in general, unknown quantity, and therefore cannot be changed. However, the minimum possible entropy assuming our model can be changed. For example, when we develop a static model, we are essentially assuming that the samples are iid, and that the entropy follows the Equation (2.2). The iid assumption limits how low the entropy of the source can be.

One way of making the probability distribution more unbalanced is to condition the probabilities on previous samples. For example, in english, the probability of a vowel is larger if we know that the previous letter was a consonant. Therefore, by conditioning the probability of a symbol on the value of the previous symbol, we are able to make the probability distribution more unbalanced. In general, the condition may be much more complex than only the previous sample. Such a condition is called the context. Let the context be represented by a random vector $\boldsymbol{\Xi}$ and let one possible context be represented by the vector $\boldsymbol{\xi}$. Our assumption is that $P(x|\boldsymbol{\xi})$ is more unbalanced than $P(x)$, and therefore $H(X|\boldsymbol{\xi})$ is lower than $H(X)$. The final entropy is the average value


375
425
475
525
575
625
0
100
200
300
400
500
600
700

$$S(\omega) = \frac{}{(2\pi)^3} \sum_{\mathbf{d}\in\mathbb{Z}^3} R(\mathbf{d})e^{-i\mathbf{d}\cdot\omega}, \qquad (10)$$

$$Q(\mathbf{d}) = \int_{[-\pi,\pi]^3} \frac{1}{S(\omega)} e^{i\mathbf{d}\cdot\omega} d\omega, \qquad (11)$$

provided $S(\omega) > 0$ for all $\omega \in [-\pi, \pi]^3$. Note that $R(\mathbf{d})$ is real and symmetric, and hence so is $S(\omega)$, $1/S(\omega)$, and $Q(\mathbf{d})$. Also note that $R(\mathbf{d})$ and $Q(\mathbf{d})$ are inverses in the sense that their correlation is the unit impulse $\delta(\mathbf{d})$. This can be verified by taking the Fourier transform of each (namely, $S(\omega)$ and $1/S(\omega)$) and multiplying them in the frequency domain to obtain a constant.

It is convenient to use $Q(\mathbf{d})$ instead of $R(\mathbf{d})$ to characterize a stationary Gaussian Process if $Q(\mathbf{d})$ is non-zero for only a finite number of values of $\mathbf{d}$, because the process would then be finitely parametrized. In this case, the process is a

equals 0 otherwise.

Since the probability density of $\mathbf{x}$ is

$$p(\mathbf{x}) = \frac{1}{(2\pi\sigma_x^2)^{\ell/2}} \exp\left(-\frac{1}{2\sigma_x^2}\mathbf{x}^T\mathbf{x}\right), \qquad (16)$$

the probability density of $\mathbf{y}$ can be written

$$p(\mathbf{y}) = \frac{|\mathbf{A}|}{(2\pi\sigma_x^2)^{\ell/2}} \exp\left(-\frac{1}{2\sigma_x^2}\mathbf{y}^T\mathbf{A}^T\mathbf{A}\mathbf{y}\right) \qquad (17)$$

$$= \frac{1}{(2\pi)^{\ell/2}|\mathbf{R}|^{1/2}} \exp\left(-\frac{1}{2}\mathbf{y}^T\mathbf{R}^{-1}\mathbf{y}\right), \qquad (18)$$

where $\mathbf{R} = \sigma_x^2(\mathbf{A}^T\mathbf{A})^{-1}$ is the covariance matrix of $\mathbf{y}$. The inverse of the covariance matrix is called the *precision matrix* [21]. It can be seen that the coefficients q_{ij} in the precision matrix $\mathbf{Q} = \mathbf{R}^{-1} = \sigma_x^{-2}(\mathbf{A}^T\mathbf{A})$ of $\mathbf{y}$ are the coefficients of the autocorrelation of the coefficients in $\mathbf{A}$

Figure 2.9: Portion of the binary image used to compute the entropies of Table 2.1. It corresponds to the third page of the article [65] converted to binary following the process described in Section 4.5.

of the entropy given each condition, or in other words [57]:

$$H(X|\Xi) = \sum_{\boldsymbol{\xi}} P(\boldsymbol{\xi})H(X|\boldsymbol{\xi}), \qquad (2.21)$$

but since every $H(X|\boldsymbol{\xi})$ is lower than $H(X)$, then this expression can only be lower than $H(X)$. Therefore, by using contexts, we are able to reduce the entropy of the source given our model.

Table 2.1 shows the values of entropies calculated for the binary image of Figure 2.9 assuming the different context definitions of Figure 2.10. It is assumed that the image pixels are scanned in a row by row basis, from the left to the right and from the top to the bottom. The first option (a) actually corresponds to no-context. In the second option (b), the context is made of the causal values of the left and the upper pixel. In the third option (c), the context is made of the values of the left pixel, the upper pixel, and the pixels to the left and to the right of the upper pixel. The other context definitions, (d) and (e), can be interpreted in a similar way. The calculated entropy continuously drops as we increase the number of previous samples that make up the context. It is reduced to almost one fifth when the number of previous samples increases from 0 to 26 pixels.

Questions may be raised about how to choose contexts, what are the best contexts

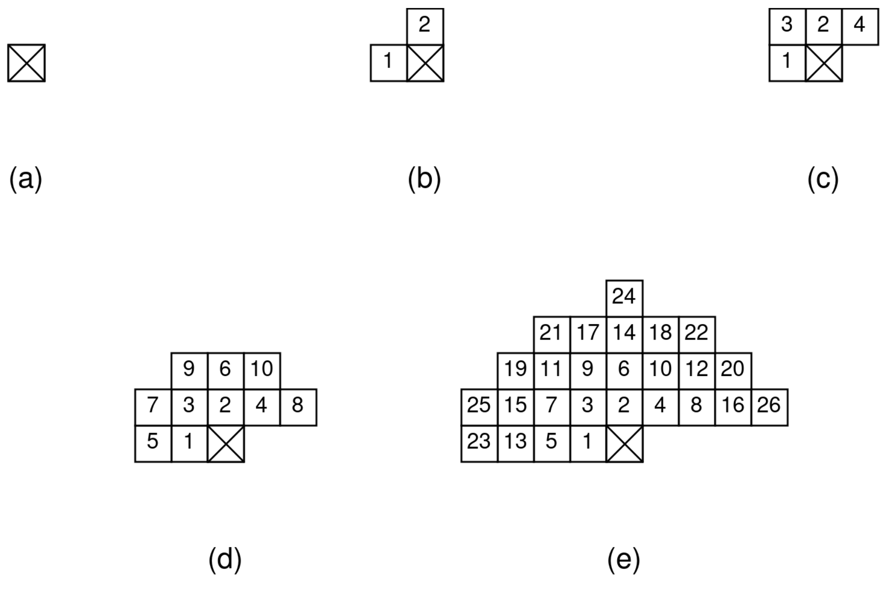


Figure 2.10: Five alternative context definitions to use with a binary image, corresponding to different amounts of pixels surrounding the current pixel. The current pixel is represented by a cross, while the pixels that make up the context are numbered from closest to farthest, respecting the rule that two pixels cannot have the same number.

Table 2.1: Values of entropies calculated for the binary image of Figure 2.9 assuming the different context definitions of Figure 2.10. The calculated entropy reduces to almost one fifth when the number of previous samples used increases from 0 to 26 pixels.

(a)	0.318
(b)	0.198
(c)	0.168
(d)	0.141
(e)	0.067

to use, what is the best context size, or the size of the vector $\boldsymbol{\xi}$. These are the types of questions that are addressed by context modeling. Context modeling is the action of defining contexts, and assigning context patterns to probability estimates. The context model is the rule that maps the patterns to the estimates.

There is a close relationship between context modeling and discrete-time Markov chains. A discrete-time Markov chain can be thought of as a finite state machine with probabilities attached to each arc [66]. They are very useful when representing dependencies among samples and when developing dynamic probability models. A common

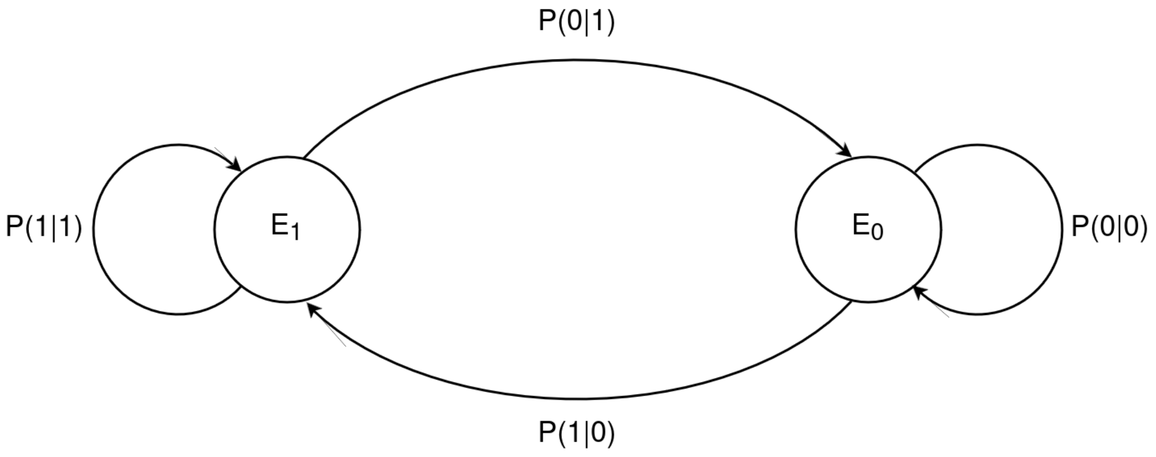


Figure 2.11: State machine corresponding to a first order discrete-time Markov chain which could be used to model the dependencies in a binary source. E_0 and E_1 are the states, which could mean, for example, that the current symbol is 0 and 1, respectively. $P(0|0)$, $P(0|1)$, $P(1|0)$, $P(1|1)$ are the transition probabilities. Based on Fig. 2.3 of [57].

example of a discrete-time Markov chain is a random process with the property that

$$P(x_n|x_{n-1}, ..., x_{n-M}, ...) = P(x_n|x_{n-1}, ..., x_{n-M}). \tag{2.22}$$

That is, the probability of the n-th sample only depends on the M samples immediately preceding it. The value M is said to be the memory of the random process. Many random processes are not strict Markov chains of this kind, but can be well approximated by one with sufficiently large M, since the influence of previous samples fades away with distance. In this particular example, each state is a combination of values taken by the set $\{x_{n-1}, ..., x_{n-M}\}$. Let S be the size of the sample space, or the number of possible symbols in the jargon of data compression, then the number of states is given by $(S)^M$. Figure 2.11 shows one example for $M = 1$ and $S = 2$. This Markov chain could be used to model a binary source. In this example, there are only two states, E_0 and E_1. The probabilities of transitioning from one state to the other, or staying in the same state, are represented by the conditional probabilities $P(0|0)$, $P(0|1)$, $P(1|0)$ and $P(1|1)$.

The state of the Markov chain corresponds to the context in context modeling. It is important to note that the state is not restricted to be the M samples immediately before the n-th sample. Any set of previous samples, or any information available at the time the n-th sample is processed, can be used to compose the context. Of importance is the fact that we can represent the process as a Markov process, in which the next state can be predicted solely based on the present state [66]. For example, in the case of a binary image, the context could be any of the contexts depicted in Figure 2.10. In any case, in this work, we will use M to denote the size of the context, regardless of the context being or not the M previous samples.

2.5 Context Modeling

The traditional way of performing context modeling is by counting symbol occurrences for each context and by using a look-up table (LUT) to store the relative or absolute frequencies. The relative frequencies can be easily obtained from the absolute frequencies for a given context. As discussed in Section 2.3, the relative frequencies are the maximum-likelihood estimates of the conditional probabilities.

LUTs can be used in both forward and backward coding. In forward coding, they are either built on the data being encoded or on some other data. If built on the data being encoded, the LUT must be transmitted to the decoder in the compressed file. In backward coding, the LUT is built on-the-fly by both the encoder and the decoder using the previously encoded or decoded samples, and it is continuously updated as new samples are seen.

Figure 2.12 shows one example of a LUT created from a binary image. Note that we adopt the common convention that black corresponds to 0 and white corresponds to 1. For simplicity, we have assumed that the values of the pixels outside of the image are 1. The context definition is the one shown in Figure 2.10 (b). The LUT stores how many times the bits 0 and 1 are seen immediately after each context, or the absolute frequencies of the bits 0 and 1. The relative frequencies for a context can be easily obtained from the absolute frequencies for that context. For example, the relative frequency of the bit

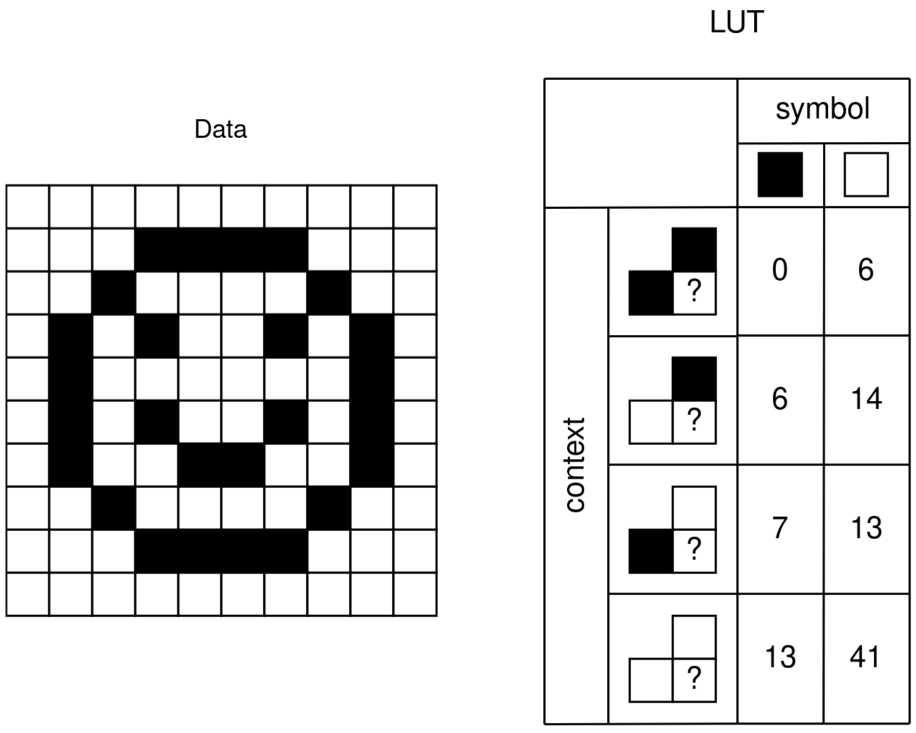


Figure 2.12: Example of LUT created from a binary image. The context definition is the one shown in Figure 2.10 (b). This LUT stores how many times each symbol is seen immediately after each context.

1 for the first context is 6 out of 6, or 100%, and the relative frequency of the bit 0 for the second context is 6 out of 20, or 30%.

Despite its simplicity and often effectiveness, there are some caveats to be aware of when using LUTs. Firstly, the number of contexts grows exponentially with the context size. For example, in the case of a binary source, the number of contexts is given by 2^M, where M is the context size. For $M = 20$, the number of contexts has already reached 1 million. This limits the maximum value of context size M that can be used with LUTs, because of the increasing memory requirements. Secondly, except when the LUT is built prior to encoding on the data that will be encoded, or on other sufficiently large amount of data, the larger the context size, the more often we will encounter unexpected situations. For example: contexts that never occurred in the original data; symbols that never occurred after a particular context. This is known as the zero frequency problem [57] or context dilution [67]. If no countermeasure is taken, the associated symbols are

Table 2.2: Number of unseen contexts and total number of contexts for the context definitions of Figure 2.10 on the binary image of Figure 2.9. This illustrates that the number of unseen contexts grows quickly with the context size.

	Unseen	Total
(a)	0	1
(b)	0	4
(c)	0	16
(d)	188	1024
(e)	67080469	67108864

encoded using many bits, which considerably increases the final bitrate in bits per sample. Context dilution also happens when the context occurrences are so infrequent that the probability estimates are inaccurate.

Table 2.2 shows the number of unseen contexts, for the context definitions in Figure 2.10, on the binary image of Figure 2.9. When $M = 10$ (context (d)) the number of unseen contexts amounts for 18.36% of the total number of contexts, while when $M = 26$ (context (e)) the number of unseen contexts amounts for 99.96% of the total number of contexts. Because the LUT cannot extrapolate the conditional probabilities for unseen contexts, even when it has seen very similar ones, this can be a big problem. Exception is made when the LUT is used to encode the same data that was used to build it.

In backward adaptive coding, the corresponding entry of the LUT is updated every time a symbol is seen after a context. The probabilities of the symbols are estimated from their absolute frequencies. Initially, the symbols can be assumed to be equiprobable, or the initial counts of all symbols can be assumed to be equal to 1, for example. This would solve the problem of not having previous samples to estimate the probabilities and would avoid the zero-frequency problem right at the beginning of coding. In Section 2.6, we discuss in more detail adaptive relative-frequency-based probability estimation in the particular case of binary sources.

2.6 Context-Adaptive Binary Arithmetic Coding

Context-adaptive binary arithmetic coding (CABAC) is the entropy coder behind the video coding standards H.265, H.266 and one of the entropy coders available in H.264. The basic H.26X video coder consists of: a prediction step, including intra-frame and inter-frame prediction, a transformation step, a quantization step, and an entropy coding step. Figure 2.13 illustrates the inner workings of the H.26X standards. In summary, the video frames are divided into blocks, each block is subtracted by an intra or inter-frame prediction, transformed using the Discrete Cosine Transform, quantized, and entropy coded along with other relevant information, such as the prediction method [68]. Upon receiving the bitstream, the decoder reconstructs the frames by undoing the steps performed by the encoder. Because of the quantization step, the reconstruction process is lossy.

Figure 2.14 illustrates in more detail what happens inside the CABAC module. In order to simplify context modeling, all values that are to be entropy coded are converted to binary strings first. In this way, context modeling can be performed on a subsymbol level, instead of the original domain, which permits the use of higher order conditional probabilities without suffering from context dilution [67]. The way binarization is performed depends on the syntax element [57]. Syntax elements describe how the video signal can be reproduced at the decoder. It is important to note that the H.26X coding standards are very complex and contain many different syntax elements. For example: quantizer labels, position of the last nonzero label, the prediction mode, and many other indicators and flags [68]. CABAC relies on hundreds of context models. The context modeling performed in H.26X, that is, the decision of which probability model to use for a given symbol, is not done based solely on the values of a few previous symbols. It is done based on several factors, such as the quantizer state, if it is chroma or luma, the position of the coefficient inside the transform block, and also on spatially neighboring quantization labels. Certain syntax elements with a more random nature are not coded using arithmetic coding and bypass the arithmetic coder.

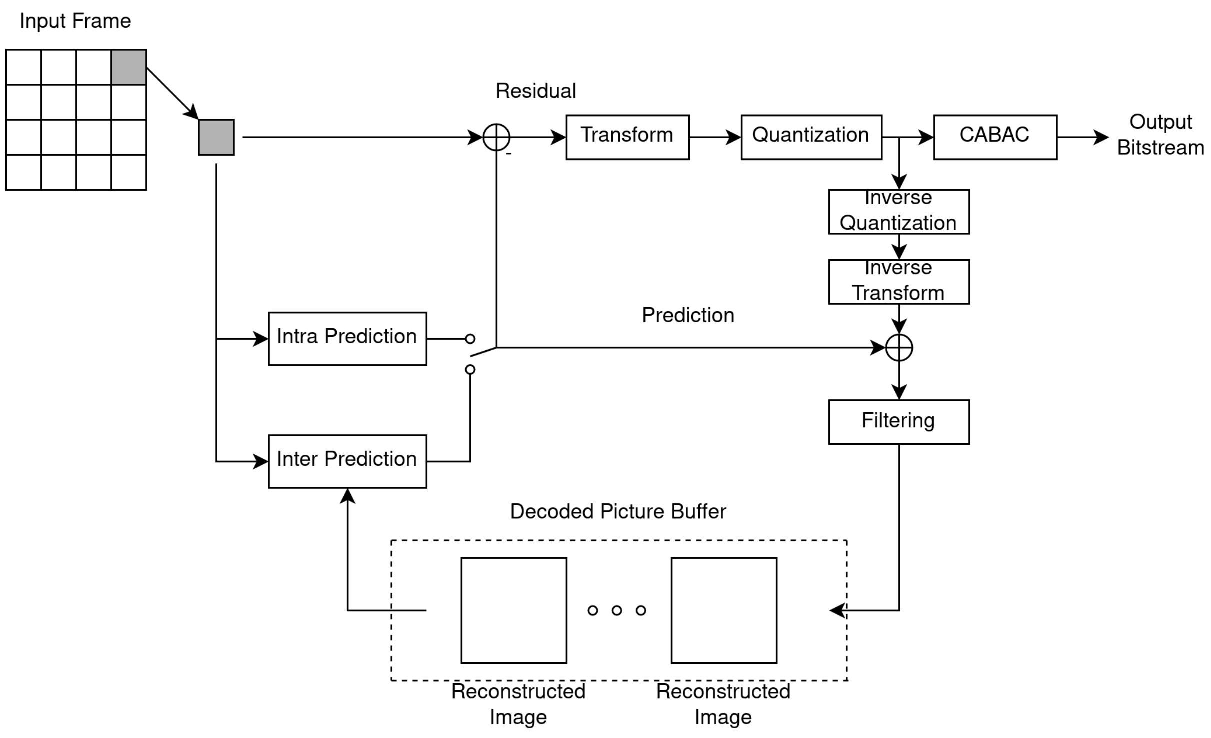


Figure 2.13: Block diagram of the H.26X video coders. The frames are divided into blocks. Each block is processed first by subtracting a prediction, which can be intra or inter-frame, then by transforming the residual, in general using the Discrete Cosine Transform. The transform coefficients are quantized and, together with other relevant information, entropy coded using CABAC. After decoding the quantization labels, the decoder recovers the residual transform coefficients, converts them to pixel values, and adds them to the prediction. Finally, the recovered blocks are grouped to form the frames and filtered to remove blocking artifacts, for example. This figure was adapted from Figure 1 of [69].

Figure 2.15 (a) shows the context template used in H.266/VVC. The quantizer labels of the elements in the shaded area are used to determine the context model, together with other information such as the quantizer state and the position inside the transformation block [68]. The context is made of elements in the lower right because the coefficients are scanned in the reverse diagonal order of the block in Figure 2.15 (b). This template is also present in H.265/HEVC [70]. As another example, when the H.264/AVC standard uses spatially neighboring syntax elements to determine the context, it generally uses the context template illustrated in Figure 2.15 (c), made of the elements to the left and above the current element in the scanning order [67]. This template is also used in a particular mode of H.266/VVC [68].

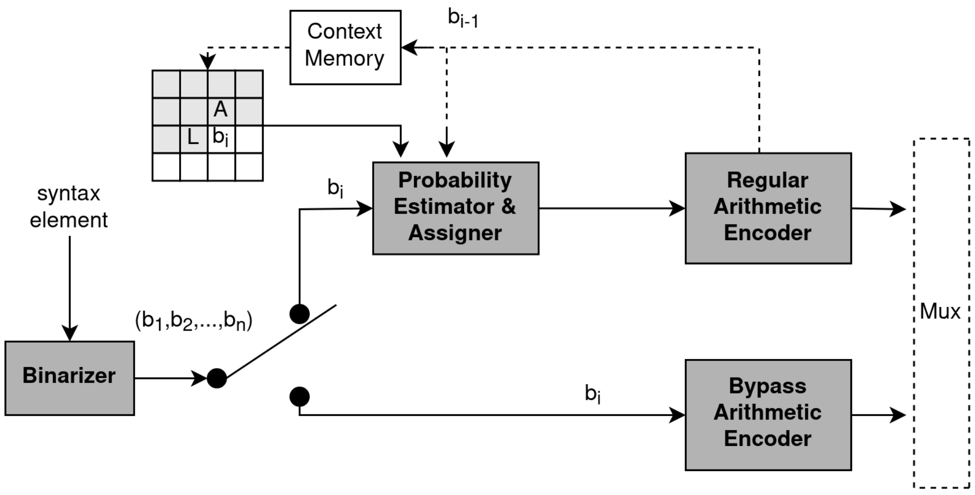


Figure 2.14: Block diagram of CABAC. First, the syntax elements are converted to binary strings. Some syntax elements, with a more random nature, bypass the arithmetic coder. The others are coded using arithmetic coding with one of several context models. The previously coded sample is used to update the context models and is stored to form the context for the next samples. This figure was adapted from Fig. 1 of [70].

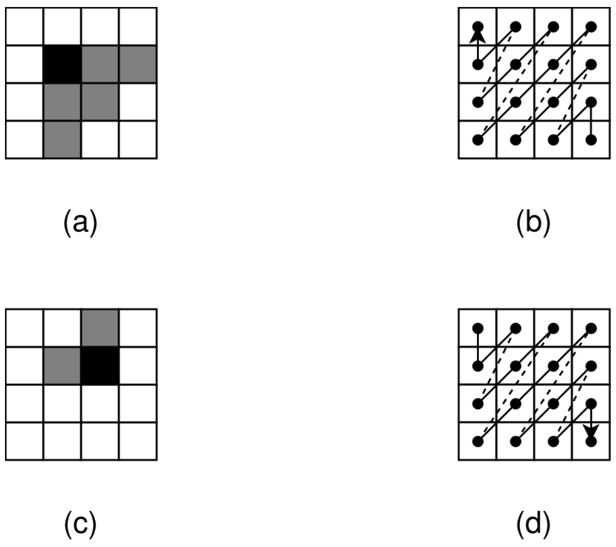


Figure 2.15: (a) Context template and (b) reverse diagonal scan coding order used in the regular residual coding mode of H.266/VVC. (c) Context template and (d) forward scan order used in the transform skip residual coding mode of H.266/VVC. This figure was adapted from Figures 3 and 5 of [68], and Figure 1 of [71].

In CABAC, the probability of the next symbol being 1 is estimated, as expected, from the values of the previous symbols coded using the same context. However, instead of doing this by simply computing the cumulative average, which would correspond to the relative frequency, the H.26X standards use an exponential decay window [68]. Consider a sequence of symbols, $\{\chi_\eta\}_{\eta \in \mathbb{N}^+}$, associated with a given context. Instead of estimating the probability of the next symbol being 1 from the cumulative average

$$\rho_{\eta+1} = \frac{\chi_1 + ... + \chi_\eta}{\eta}, \tag{2.23}$$

or, in recursive form, from

$$\rho_{\eta+1} = \frac{\chi_\eta + (\eta - 1)\rho_\eta}{\eta}, \tag{2.24}$$

CABAC estimates the probability from [68] [72]

$$\rho_{\eta+1} = \alpha[\chi_\eta + (1-\alpha)\chi_{\eta-1} + (1-\alpha)^2\chi_{\eta-2} + ... + (1-\alpha)^{\eta-1}\chi_1], \tag{2.25}$$

where α is a constant between 0 and 1. Note that $1 + (1-\alpha) + (1-\alpha)^2 + ... + (1-\alpha)^{\eta-1}$ is equal to $(1-(1-\alpha)^\eta)/\alpha$, which, for large η, tends to $1/\alpha$. That is, Equation (2.25) is essentially a weighted average with exponentially decaying weights, such that closer values have larger weights. Equation (2.25) can also be written in recursive form as [68] [72]

$$\rho_{\eta+1} = \alpha\chi_\eta + (1-\alpha)\rho_\eta. \tag{2.26}$$

The constant α controls the rate of adaptation. A lower α results in slower adaptation, while a larger α results in faster adaptation. In H.264 and H.265, the value of α is [67] [70]

$$\alpha = 1 - \left(\frac{0.01875}{0.5}\right)^{1/63}. \tag{2.27}$$

In other words, $\alpha \approx 0.05$. In H.266, two estimates for two rates of adaptation, α_0 and α_1, are maintained for each context, with the final probability estimate being the average of the two estimates, and the values of α_0 and α_1 were optimized for each context, together with the initial probabilities, using a training algorithm [68].

In H.264 and H.265 this exponential smoothing estimator is implemented using a finite state machine with 128 states. What is tracked is a value between 0 and 0.5 which represents the probability of the least probable symbol (LPS), and the value of the most probable symbol (MPS). The probability of the LPS changes from one state to the other, being increased if the LPS occurs and decreased if the MPS occurs. If the probability reaches 0.5 and the LPS occurs, the probability is kept the same but the value of the MPS is toggled [67] [70]. The H.266 standard does not use this state machine, and derives the probability estimates using directly the recursive function in Equation (2.26) [68].

Chapter 3

Neural Networks and Hyperparameter Optimization

3.1 Feedforward Neural Networks

An artificial neural network can be defined as "a massively parallel distributed processor made up of simple processing units that has a natural propensity for storing experiential knowledge and making it available for use." [73]. The artificial neurons are the basic processing units that make up the artificial neural network. Each of them is a combination of an affine function of the form $z(\boldsymbol{\psi}) = \mathbf{w}^T\boldsymbol{\psi} + b$, followed by a nonlinear function g, also called an activation function. The output of the neuron, also called activation, then is $a = g(z)$. [1] The values in $\mathbf{w}$ are called its weights, and the value b is called its bias. Figure 3.1 (a) shows a schematic representation of a neuron.

Many neurons can be stacked together in parallel, forming a layer. Let J be the number of neurons being stacked in parallel, let $\mathbf{w}_j$ and b_j be the weight vector and the bias of the j-th neuron, then the activation of the j-th neuron is $a_j = g(z_j)$, where $z_j = \mathbf{w}_j^T\boldsymbol{\psi} + b_j$. The outputs of the J neurons can be represented in a single equation by $\mathbf{a} = \boldsymbol{g}(\mathbf{z}) = \boldsymbol{g}(\mathbf{W}\boldsymbol{\psi} + \mathbf{b})$, where $\mathbf{a} = [a_1, ..., a_J]^T$, $\mathbf{z} = [z_1, ..., z_J]^T$, $\mathbf{W} = [\mathbf{w}_1 ... \mathbf{w}_J]^T$,

[1]We have purposely left out the dependence of a and z on $\boldsymbol{\psi}$, and will continue to do so, in order to simplify the notation.

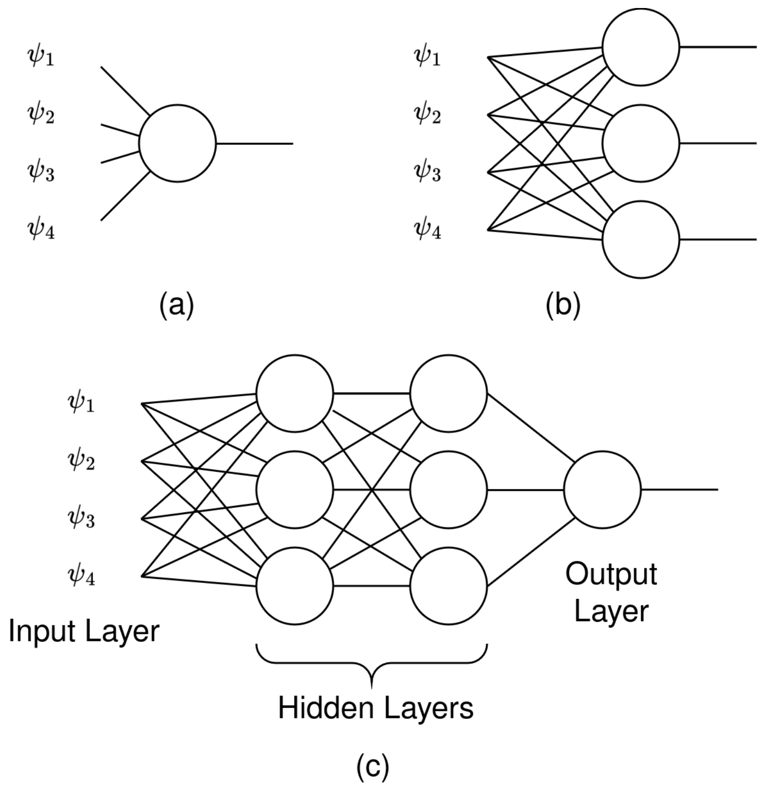


Figure 3.1: Illustrations of: (a) a neuron; (b) a layer of neurons; (c) a neural network.

$\mathbf{b} = [b_1, ..., b_J]^T$ and $\boldsymbol{g}(\mathbf{z}) = [g(z_1), ..., g(z_J)]^T$. Figure 3.1 (b) shows an illustration of a layer.

A neural network can be obtained by sequentially stacking many layers. Let a series of layers in a neural network be indexed by $d \in \{1, ..., d_{\max}\}$, such that the lower the index, the closer the layer is to the input. Then, the output of an arbitrary layer in the network can be represented by $\mathbf{a}^{(d)} = \boldsymbol{g}^{(d)}(\mathbf{z}^{(d)}) = \boldsymbol{g}^{(d)}(\mathbf{W}^{(d)}\mathbf{a}^{(d-1)} + \mathbf{b}^{(d)})$, with the input of the network corresponding to $\mathbf{a}^{(0)} = \boldsymbol{\psi}$. The output of the network may be $\mathbf{a}^{(d_{\max})}$ or $\mathbf{z}^{(d_{\max})}$ depending on the application, that is, it may contain or not the activation function. The j-th row in $\mathbf{W}^{(d)}$ is the transposed weight vector $(\mathbf{w}_j^{(d)})^T$ of the j-th neuron in the d-th layer, while the j-th element in $\mathbf{b}^{(d)}$, or $b_j^{(d)}$, is the bias. We will use $J^{(d)}$ to indicate the number of outputs of the d-th layer, which is equal to the number of neurons in the layer, and $K^{(d)}$ to indicate the number of inputs of the d-th layer. For convenience, we define $J^{(0)}$ as the number of elements in $\boldsymbol{\psi}$. Then, $K^{(d)} = J^{(d-1)}$ for all d. Furthermore, each vector $\mathbf{w}_j^{(d)}$ has $K^{(d)}$ elements, represented by $w_{jk}^{(d)}$.

Figure 3.1 (c) illustrates a neural network constructed this way. By convention, the set of inputs is called the input layer. It is different from the other layers because it does not contain any neurons. The last layer is called the output layer. The layers in between the input and output layers are called the hidden layers. The networks of this type are called feedforward neural networks, because they do not have any loops, and can be represented by directed acyclic graphs, in contrast with other broad class of neural networks called recurrent neural networks, which is characterized by the presence of loops, feedbacks or cycles [73]. A network like the one in Figure 3.1 (c) is also called a multi-layer perceptron (MLP) for historical reasons. A perceptron is a particular type of artificial neuron, which preceded the general definition we gave in the beginning of this section. In a perceptron, the inputs and outputs are either 0 or 1, and the activation function is the heaviside step function [74]. Initially, a MLP was a network formed by combining perceptrons, but the community continued using the term to describe more general networks, with inputs other than 0 and 1, and activation functions other than the heaviside step function.

It can be shown that MLPs are universal approximators [74] [75]. That is, they can approximate any continuous function $\boldsymbol{\zeta} : \mathbb{R}^{J^{(0)}} \rightarrow \mathbb{R}^{J^{(d_{\max})}}$ of $J^{(0)}$ inputs and $J^{(d_{\max})}$ outputs with arbitrary precision, even if the neural network has a single hidden layer, as long as there are enough neurons in that layer, or even if the neural network has a limited number of neurons in each layer, as long as there are enough layers. These results are known as universal approximation theorems, particularly the arbitrary width and the arbitrary depth cases. The nonlinear functions in the hidden layers are essential. Without them, the neural network could only approximate linear functions. The approximation can be proved for different nonlinear functions. The most common nonlinear functions are the sigmoid function and the ReLU function (Figure 3.2). With respect to the neurons in the last layer, the theorems usually assume that there is no activation function. In this way, the range of values in an output is unlimited. However, if the outputs are supposed to be probabilities, then the neurons in the output layer may contain a sigmoid activation function, if the probabilities are supposed to be independent. Alternatively,

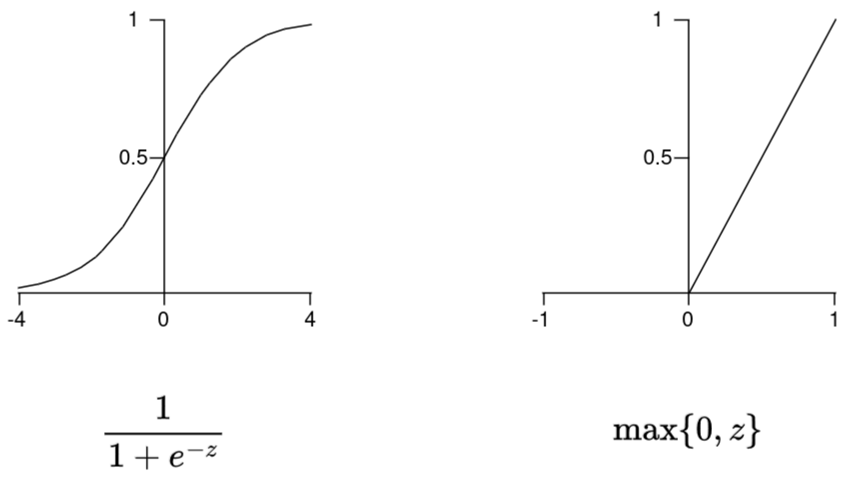


Figure 3.2: Sigmoid (left) and ReLU (right) activation functions.

they may contain a softmax activation function, if the probabilities are supposed to be from the same probability distribution. For example, consider a single layer of neurons. The softmax activation function receives the vector $\mathbf{z}$ as input, and outputs a vector with the same size as the input, Softmax($\mathbf{z}$). The j-th element of the output of the softmax has the form $e^{z_j} / \sum_{j'=1}^{J} e^{z_{j'}}$. Not only the softmax transforms the inputs into probabilities from the same distribution (fits the numbers into the range $(0, 1)$ and makes them add up to 1), but it also emphasizes the largest number (except when all z_j are small) [75].

3.2 Neural Network Training

Neural network training is usually based on two algorithms: mini-batch stochastic gradient descent and backpropagation. The first one provides a way to minimize a particular loss function, which is based on the gradients of the loss function, and the second one provides an efficient way to obtain the gradients of the loss function with respect to the parameters of the network.

A loss function, or cost function, is a function that we intend to minimize. For example, we may wish to minimize the mean squared error between the outputs of the network and the expected outputs for particular inputs. The set of inputs and their corresponding expected outputs form the training dataset. This loss function can be represented as:

$$L = \sum_{(\boldsymbol{\psi},\mathbf{y})\in\mathcal{D}} \frac{\|\mathbf{y} - \boldsymbol{f}(\boldsymbol{\psi},\boldsymbol{\theta})\|^2}{|\mathcal{D}|}, \tag{3.1}$$

where $(\boldsymbol{\psi}, \mathbf{y})$ is one pair of input and expected output from the dataset $\mathcal{D}$, the vector $\boldsymbol{\theta}$ represents the network parameters, and $\boldsymbol{f}(\boldsymbol{\psi}, \boldsymbol{\theta})$ is the output of the network for the input $\boldsymbol{\psi}$.

In order to minimize such a loss function, we assume the unlikely case that the loss function in (3.1) is convex with respect to the network parameters. The partial derivative of the loss function with respect to a generic network parameter θ, when this is at a particular value θ_0, or $\frac{\partial L}{\partial \theta}\Big|_{\theta_0}$, gives the direction of change in L for a positive change in θ. If the partial derivative is positive, then an increase in θ causes an increase in L. If the partial derivative is negative, then an increase in θ causes a decrease in L. If we want to minimize L, then it makes sense to change θ in the opposite direction of the partial derivative. If a positive change in θ causes a positive change in L, then we should reduce θ. If a positive change in θ causes a negative change in L, then we should increase θ. This can be achieved by the following equation:

$$\theta_{\text{new}} = \theta_{\text{old}} - \epsilon \left.\frac{\partial L}{\partial \theta}\right|_{\theta_{\text{old}}}, \tag{3.2}$$

where the constant ϵ is called the learning rate. Let $\boldsymbol{\theta}$ denote all network parameters. Instead of updating only θ, we could approach the minimum much quicker if we did similar updates to all parameters in $\boldsymbol{\theta}$. This could be compactly represented by:

$$\boldsymbol{\theta}_{\text{new}} = \boldsymbol{\theta}_{\text{old}} - \epsilon(\nabla_{\boldsymbol{\theta}} L)_{\boldsymbol{\theta}_{\text{old}}}, \tag{3.3}$$

where $\nabla_{\boldsymbol{\theta}} L$ is the gradient of L with respect to $\boldsymbol{\theta}$. Naturally, a single update of all network parameters using this update rule would most likely not minimize L with respect to the network parameters. However, if, on the other hand, this update rule is sequentially applied, then the function L can in fact be minimized, provided that the value of ϵ is low enough [76]. This is the gradient descent (GD) algorithm.

We have assumed that L is convex with respect to the network parameters. This ensures that the loss function has a single minimum which is also a global minimum. Unfortunately, this does not match the common case. In practice, it is often the case that L is a function with a very complex shape in a very high-dimensional space, with many local minima. However, this function can still be considered locally convex in many places. Therefore, GD can still find a local minimum, and we want this minimum to be not too far from the global minimum. In order to find a good local minimum, the network parameters' initialization values are very important. As a consequence, there exist many initialization methods.

One update using the loss function in (3.1) considers all training samples at once. Because of this, the algorithm we just described is also called batch gradient descent. However, this algorithm is not very practical when the training samples are very high-dimensional. This is the case, for example, with images. It would require a computer with a very large volatile memory to hold all such data at once. One alternative is to replace the single update for the whole dataset with several updates, each considering a portion of the data at a time. Consider, for example, that one update is made for every sample. Let $\mathcal{I}$ indicate a set composed of a single element from $\mathcal{D}$. Then, the loss function for one sample would be

$$L_{\mathcal{I}} = \|\mathbf{y} - \boldsymbol{f}(\boldsymbol{\psi}, \boldsymbol{\theta})\|^2. \tag{3.4}$$

The update for one training sample would be

$$\theta_{\text{new}} = \theta_{\text{old}} - \epsilon \left. \frac{\partial L_{\mathcal{I}}}{\partial \theta} \right|_{\theta_{\text{old}}}. \tag{3.5}$$

The update for all network parameters at once would be

$$\boldsymbol{\theta}_{\text{new}} = \boldsymbol{\theta}_{\text{old}} - \epsilon (\nabla_{\boldsymbol{\theta}} L_{\mathcal{I}})_{\boldsymbol{\theta}_{\text{old}}}. \tag{3.6}$$

Surprisingly, the algorithm with one update per sample would still approach the minimum of (3.1), assuming there is only one, with respect to the network parameters [64],[77],[76]. This is so because $\partial L_{\mathcal{I}}/\partial\theta|_{\theta_{\text{old}}}$ is an unbiased estimator to $\partial L/\partial\theta|_{\theta_{\text{old}}}$. However, for a constant ϵ, there is a bias term separating the minimum found by the algorithm and the true minimum. This bias term can be made to disappear if the learning rate is reduced as the number of iterations increases [64],[77],[76]. This is the stochastic gradient descent algorithm.

Similar statements can be made if more samples are taken at a time. For example, for loss function $L_{\mathcal{B}} = \sum_{(\boldsymbol{\psi},\mathbf{y})\in\mathcal{B}} \|\mathbf{y} - \boldsymbol{f}(\boldsymbol{\psi},\boldsymbol{\theta})\|^2/|\mathcal{B}|$ and partial derivatives $\partial L_{\mathcal{B}}/\partial\theta$, where $\mathcal{B}$ is a subset of the data containing $|\mathcal{B}| < |\mathcal{D}|$ samples. This is called mini-batch stochastic gradient descent. It is normally used over the other alternatives when training neural networks because it is more memory efficient than GD and more time efficient than SGD. The parameter $B = |\mathcal{B}|$ is the "batch size". One pass over all training samples, be it on GD, SGD or Mini-batch SGD, is called one "epoch". One GD update corresponds to one epoch, while several SGD or Mini-batch SGD updates are necessary to conclude one epoch.

The difference between machine learning and normal optimization is that the function one minimizes is not exactly the function one wants to minimize. You want to minimize the loss function for the test data, or more specifically, for the true distribution of the data, but instead the loss function for the training data is minimized [64].

As mentioned at the beginning of this section, the backpropagation algorithm can be used to find the partial derivatives of the loss function with respect to the network parameters in an efficient way. We will not be covering it in detail, as this is out of the scope of this work. However, the interested reader is referred to the excellent description of the backpropagation algorithm present in [74].

3.3 Recurrent Neural Networks

Consider again a sequence of data elements $\mathbf{x}^{(N)} = \{x_1, x_2, ..., x_N\}$. When discussing recurrent neural networks (RNNs), the elements of the sequence are often words in a language, for example English, or letters from an alphabet, for example the English alphabet, and are generally called tokens. In order to treat the tokens numerically, they are converted to a suitable numerical representation. Associating each token to an integer is known as ordinal encoding. Associating each token to a binary vector with a single nonzero entry, with the index of the nonzero entry encoding the token, is known as one-hot-encoding. In the following, assume that each token is represented by its one-hot-encoding. Let us compare a couple of network architectures when processing this input sequence one input at a time.

Let the sequence of tokens in one-hot-encoding representation be denoted by $\mathbf{t}_1, \mathbf{t}_2, ..., \mathbf{t}_N$. The following equations describe a MLP with one hidden layer, acting on one input at a time:

$$\mathbf{a}_n^{(1)} = \boldsymbol{g}^{(1)}(\mathbf{W}^{(1)}\mathbf{t}_n + \mathbf{b}^{(1)}) \tag{3.7}$$

$$\mathbf{a}_n^{(2)} = \boldsymbol{g}^{(2)}(\mathbf{W}^{(2)}\mathbf{a}_n^{(1)} + \mathbf{b}^{(2)}) \tag{3.8}$$

where $\mathbf{W}^{(d)}$ is the weight matrix, $\mathbf{b}^{(d)}$ is the bias vector, $\boldsymbol{g}^{(d)}$ are the activation functions and $\mathbf{a}_n^{(d)}$ is the output of the d-th layer at the n-th time instant.

Since the network has no knowledge of the previous input when computing the output for the current input, the most this network can do is compute a mapping from the one-hot-encoding representation of the inputs to another representation, possibly with less numbers. In other words, it can, at most, be used as an embedding layer [64]. Note that this is different from coding for data compression. The output vectors are continuous-valued, with a fixed size, and there is no guarantee that different inputs would not be mapped to equal or similar outputs. In fact, the purpose of an embedding layer is usually to approximate the representations of similar inputs, for example, words that have similar meanings. These representations are often called word embeddings. In practice an embedding layer often corresponds to a single linear layer.

A recurrent neural network, on the other hand, can take previous inputs into account when computing the output for the current input. The following equations describe a basic (Elman) RNN [73], [78]:

$$\bar{\mathbf{a}}^{(1)}_{n} = \bar{\boldsymbol{g}}^{(1)}\left(\bar{\mathbf{W}}^{(1)}\begin{bmatrix}\bar{\mathbf{a}}^{(1)}_{n-1}\\ \mathbf{t}_n\end{bmatrix} + \bar{\mathbf{b}}^{(1)}\right), \tag{3.9}$$

$$\bar{\mathbf{a}}^{(2)}_{n} = \bar{\boldsymbol{g}}^{(2)}(\bar{\mathbf{W}}^{(2)}\bar{\mathbf{a}}^{(1)}_{n} + \bar{\mathbf{b}}^{(2)}), \tag{3.10}$$

where the symbols have similar meanings to what they had in the MLP equations. The only difference here is that the input of the network is the concatenation of $\mathbf{t}_n$ with $\bar{\mathbf{a}}^{(1)}_{n-1}$, the output of the hidden layer for the previous input, and therefore $\bar{\mathbf{W}}^{(1)}$ has more columns. In this way, information about the previous network state can be passed to the calculation of the current network state. For initialization purposes, $\bar{\mathbf{a}}^{(1)}_{0}$ may be all zeros, for example.

For the sake of illustration, consider the common task of predicting the next token in a sequence using a RNN. Assume that the output of the network approximates the one-hot-encoding of the next token in the sequence as a real-valued vector. This could be achieved with a Softmax activation function in the output layer and appropriate training. After training, during utilization of the model, the actual prediction of the network for the next token could be obtained, in ordinal encoding, by taking the argmax of the output.

The RNN can be trained for this task as follows. Let $\boldsymbol{f}(\mathbf{x}^{(n-1)}, \boldsymbol{\theta})$ be the output of the RNN. Let the loss function for one sequence position n be $\mathscr{d}(\mathbf{t}_n, \boldsymbol{f}(\mathbf{x}^{(n-1)}, \boldsymbol{\theta}))$, where $\mathscr{d}$ is some distance measure. Then the loss function for the entire sequence would be:

$$L_{\text{seq}} = \sum_{n=1}^{N} \mathscr{d}(\mathbf{t}_n, \boldsymbol{f}(\mathbf{x}^{(n-1)}, \boldsymbol{\theta})), \tag{3.11}$$

where $\mathbf{x}^{(0)} = \{\}$ is the empty set and $\boldsymbol{f}(\{\}, \boldsymbol{\theta})$ represent the output of the RNN for the first element of the sequence. During training with stochastic gradient descent, the batch is composed of several sequences of a given length. The network goes through each sequence computing outputs. The loss for the outputs are computed, first for each token, then for one entire sequence, then for all sequences. The total loss is used to calculate the partial derivatives of the loss function with respect to the network parameters. Then, the network parameters are updated using the partial derivatives. Since there is parameter-sharing across time steps, the partial derivatives of the loss function with respect to the shared network parameters contain terms for different time steps. For this reason, the algorithm used to compute the partial derivatives of the loss function with respect to the parameters of a RNN is called backpropagation through time (BPTT) [79]. Sometimes the number of past samples considered is truncated and the resulting algorithm is called truncated backpropagation through time (TBPTT) [79].

In order to use a MLP for the same task of predicting the next token from the previous tokens, the input to the network needs to include $M - 1$ more previous samples besides only $\mathbf{t}_{n-1}$. Hence, the network would have M one-hot-encoded vectors as input $\mathbf{t}_{n-1}, \mathbf{t}_{n-2}, \mathbf{t}_{n-3}, ..., \mathbf{t}_{n-M}$ and the output of the network would be the network prediction for $\mathbf{t}_n$. The number of input nodes would be $M \cdot S$. This could be a problem depending on the vocabulary size S, because the number of parameters could quickly become very large. One could try using ordinal encoding for the inputs instead of one-hot-encoding, in which case the number of input nodes would simply be M. However, since the neural network inputs are usually normalized to the $[0, 1]$ range for numerical reasons, this would mean fitting many numbers into a small range, except, for example, in the special case of binary data. In any case, the MLP would only be capable of taking a limited amount of previous samples (M) into account, while, in theory, a RNN is capable of considering an unlimited amount of previous samples.

Note that naive RNNs as described here are known to have short memory, that is, the influence of previous samples on the current output tend to vanish quickly, due to the vanishing gradient problem [64]. This has motivated the proposal of more sophisticated RNNs, mainly long-short-term-memory (LSTM) networks and gated recurrent units

(GRU) [64]. Although their inner workings may considerably change, as black boxes they are actually quite similar to the naive RNNs. They are all based on the same idea of having a hidden state being passed from the previous time step to the next.

3.4 Cross-Validation

As mentioned in Section 3.2, one of the main differences between machine learning and regular optimization is that, in machine learning, the models are optimized on a different setting from the one in which we are primarily interested. The phase in which the model is optimized is known as the training phase, while the phase in which the model is effectively utilized is known as the inference phase, or the prediction phase.

However, since the model is intended to be utilized on data different from the one it was trained on, it is essential to estimate the generalization performance of the model before utilizing it. Furthermore, since there are so many different machine learning algorithms, each of which may have multiple variations, it is also very common to compare multiple algorithms, and algorithm configurations, before deciding on a specific one. These two steps are part of a third phase of machine learning which is known as the cross-validation phase [80]. The specific part in which different models are compared is also known as the model selection phase, and the part in which the generalization performance is estimated is known as the test phase.

The different machine learning algorithms often have parameters which are not learned during training, and have to be decided on by other means. Think of the number of layers in a multi-layer perceptron, the number of neurons in each layer, or the type of activation function used in the hidden neurons, for example. These parameters are referred to, in the machine learning literature, as the model hyperparameters. When the model selection phase is focused on selecting hyperparameters for a specific algorithm, this phase can also be called the hyperparameter selection phase, or hyperparameter optimization.

There are two main types of cross-validation: holdout cross-validation and k-fold cross-validation [80]. In proper holdout cross-validation with model selection, the available data is split into three datasets: a training dataset, a validation dataset, and a test dataset. As mentioned in Section 3.2, the training dataset is used to optimize the model. The validation dataset, in its turn, is used to estimate the generalization performance of the different models during model selection. However, since we use the results on the validation set to select a specific model, it may now have been overfitted to the validation set. In order to properly estimate its generalization performance, we must evaluate it again on a third dataset. This is the purpose of the test set.

In k-fold cross-validation, the training and validation datasets are normally combined together and subdivided into k folds. During model selection, each of the different models is trained and evaluated k times. In each time, $k-1$ folds are used for training, and the other 1 fold is used for evaluation. After evaluating each model on all folds, their final generalization performances are estimated by their average performances on all folds. Then, the selected model is retrained on all folds and tested on the third, and independent, test set [80]. k-fold cross-validation is therefore more reliable than holdout cross-validation, but it is also more costly.

3.5 Hyperparameter Optimization

The problem of selecting the hyperparameters of a model is also an optimization problem, as is the model training. However, the main difference is that we do not know the derivatives of the cost function with respect to the hyperparameters. Therefore we cannot use backpropagation and stochastic gradient descent to learn those parameters. Another difference is that the models are evaluated on the validation set, instead of the training set.

There are two main types of hyperparameter optimization: single-objective hyperparameter optimization (SOHPO) and multiple-objective hyperparameter optimization

(MOHPO). Consider first SOHPO. Let the hyperparameters of a model be represented by a vector $\mathbf{h} \in \mathcal{H}$, where $\mathcal{H}$ is the set of possible hyperparameter vectors, and the single-objective function by $c : \mathcal{H} \to \mathbb{R}$. SOHPO methods seek to solve [2]

$$\mathbf{h}^* = \arg\min_{\mathbf{h} \in \mathcal{H}} c(\mathbf{h}), \tag{3.12}$$

where the value of the objective function is taken on the validation set. Note that, in general, the space of hyperparameter vectors, $\mathcal{H}$, is too large. Therefore it is not possible to train and evaluate the models for all possible $\mathbf{h}$. Then, the optimal hyperparameter vector must be estimated based on a subset of $\mathcal{H}$. SOHPO methods differ on how this subset is obtained. The most basic SOHPO methods are grid-search and random-search. In grid-search, it is previously decided on a set of values to be tested for each hyperparameter. Then all possible combinations of those hyperparameter values are tested. In random-search, on the other hand, instead of testing all possible combinations, only a handful of randomly selected combinations are tested. This allows for considering a larger set of possible values for each hyperparameter.

In contrast to SOHPO, when optimizing multiple objectives there is not one single best solution, but rather a set of incomparable non-dominated solutions, or Pareto-optimal solutions. Let the multiple objectives be represented by a vector-valued function $\mathbf{c} : \mathcal{H} \to \mathbb{R}^\nu$. A hyperparameter configuration $\mathbf{h}$ is said to Pareto-dominate another $\mathbf{h}'$, writen as $\mathbf{h} \prec \mathbf{h}'$, if and only if [53]:

$$\begin{gathered} \forall\ i \in \{1, ..., \nu\} : c_i(\mathbf{h}) \leq c_i(\mathbf{h}') \ \wedge \\ \exists\ j \in \{1, ..., \nu\} : c_j(\mathbf{h}) < c_j(\mathbf{h}'). \end{gathered} \tag{3.13}$$

A configuration is said to be non-dominated, or Pareto-optimal, if and only if there is no other configuration that dominates it. Different from SOHPO, two configurations can be incomparable, when there exist $i, j \in \{1, ..., \nu\}$ such that $c_i(\mathbf{h}) < c_i(\mathbf{h}')$ and $c_j(\mathbf{h}') < c_j(\mathbf{h})$.

[2]In this work, we assume, without loss of generality, that all objectives that should be maximized have been converted to ones that should be minimized, for example, by multiplying them by -1.

The set of incomparable non-dominated solutions is called the Pareto set, and is defined as

$$\mathcal{P} := \{\mathbf{h} \in \mathcal{H} \mid \nexists\ \mathbf{h}' \in \mathcal{H} \text{ s.t. } \mathbf{h}' \prec \mathbf{h}\}. \tag{3.14}$$

These solutions have different trade-offs. It is not possible to improve one objective without degrading another objective. The image of $\mathcal{P}$ under $\mathbf{c}$, $\mathbf{c}(\mathcal{P})$, is called the Pareto front.

MOHPO algorithms seek to approximate the set of Pareto-optimal solutions. This can be expressed as [53]:

$$\min_{\mathbf{h}\in\mathcal{H}} \mathbf{c}(\mathbf{h}) = \min_{\mathbf{h}\in\mathcal{H}}(c_1(\mathbf{h}), ..., c_\nu(\mathbf{h})). \tag{3.15}$$

Or [81]:

$$\begin{gathered} \min c_1(\mathbf{h}), ..., \min c_\nu(\mathbf{h}), \\ \mathbf{h} \in \mathcal{H}. \end{gathered} \tag{3.16}$$

Again, it is not possible to test all hyperparameter configurations in $\mathcal{H}$. Therefore, a few hyperparameter vectors must be selected somehow. Grid-search and random-search can also be used in MOHPO. They have to be combined with some algorithm to find the Pareto set of a known set of points. One such algorithm is the `Best` algorithm described in [82]. This algorithm starts with any point, and iterates through the list of hyperparameter vectors, removing the ones dominated by the given point, until it finds a point which dominates it. It then removes any dominated points that may have remained in the list. It repeats this process until the list is empty.

3.6 Lower Convex Hull

A related concept to the Pareto front is the concept of lower convex hull (LCH). Being on the LCH is actually a stronger requirement than being on the Pareto frontier. The points on the LCH are a subset of the points that are on the Pareto frontier. Points on the

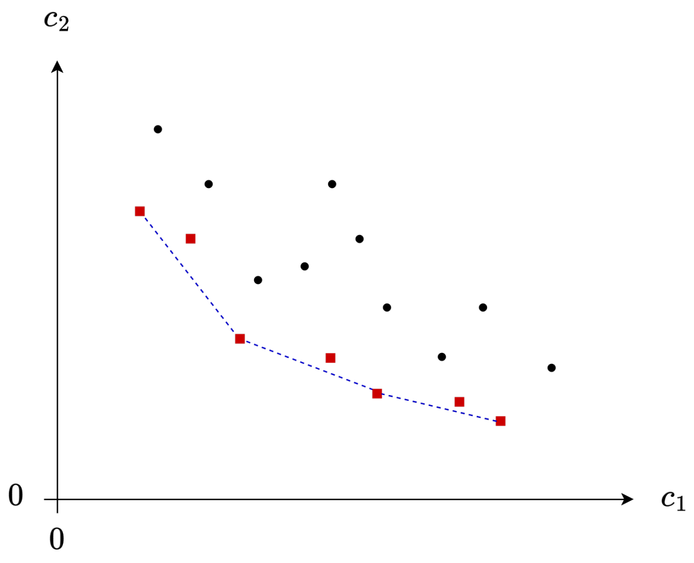


Figure 3.3: The Pareto frontier versus the LCH of a set of points. The points on the Pareto frontier are those represented by the red squares, while the points on the LCH are those touching the blue dashed lines.

Pareto frontier are those with no points on the lower left quadrant. The Pareto frontier includes some "interior" points, which the LCH does not.

Assume, without loss of generality, that there are only two objectives, c_1 and c_2. Consider the minimization problem:

$$\min_{\mathbf{h}} c_1(\mathbf{h}) + \lambda c_2(\mathbf{h}). \tag{3.17}$$

Minimizing it for a specific λ corresponds to sliding a line with a specific inclination, starting from the origin, until it hits a point $(c_1(\mathbf{h}), c_2(\mathbf{h}))$. The first point that is touched by the line is the minimum. When $\lambda = 0$, the line is vertical. When $\lambda \to \infty$, the line is nearly horizontal. For $0 \leq \lambda < \infty$, the orientation of the line is something in between. Doing so with various choices of $\lambda \in [0, \infty)$ allow us to sweep out the lower left convex hull of the points. This process excludes some of the points on the pareto frontier. The difference between the LCH and the Pareto frontier is exemplified in Figure 3.3.

Instead of always saying "the lower left convex hull" of the cloud of points, it is common to refer to it simply as "the lower convex hull". Some might even simply say "the convex

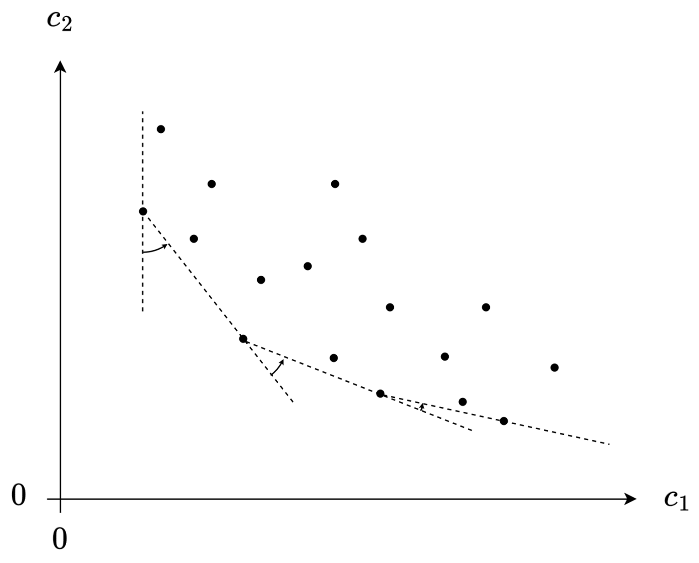


Figure 3.4: Illustration of an algorithm which can be used to find the LCH based on the Gift Wrapping algorithm. The algorithm starts with the leftmost point, and subsequently selects the point with lowest polar angle, with respect to the last found convex-hull side.

hull" when referring to it.

The LCH can be defined in higher dimensions in a similar way. For example, for objectives c_1, c_2 and c_3, in which case the minimization problems would have the form $\min_{\mathbf{h}} c_1(\mathbf{h}) + \lambda c_2(\mathbf{h}) + \gamma c_3(\mathbf{h})$, with $\lambda \in [0, \infty)$ and $\gamma \in [0, \infty)$.

The LCH in 2D can be found in various ways. One of them is by a modified version of the gift wrapping algorithm [83]. We begin with the leftmost point. This is the point found by minimizing the expression in (3.17) for $\lambda = 0$, or, in other words, the first point touched when sliding a vertical line along the horizontal axis. Consider the line with λ going through the most recently selected point. The point which gives the smallest anticlockwise angle with respect to this line, with the most recently selected point as vertex, is the next point of the LCH. The algorithm then repeats this procedure, keeping track of the inclination of the last convex-hull side chosen, and selecting the next LCH point which gives the smallest polar angle with respect to this line-segment. The algorithm stops when the line becomes horizontal or increasing. This procedure is illustrated in Figure 3.4.

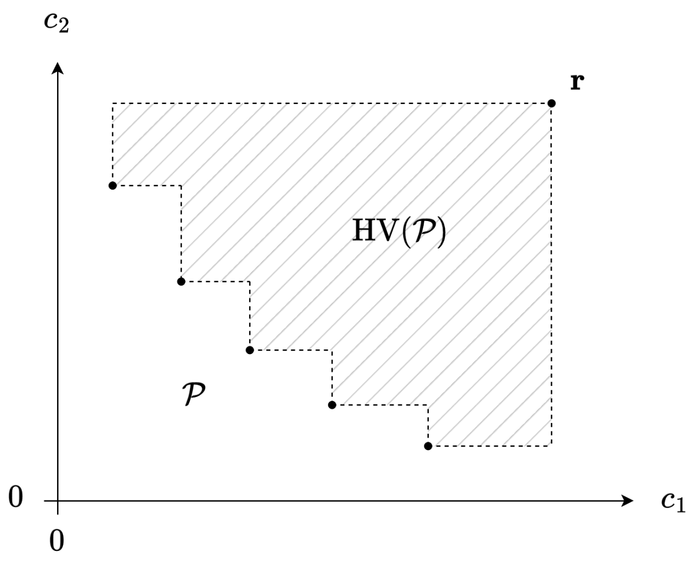


Figure 3.5: Illustration of the Hypervolume of Pareto set $\mathcal{P}$ with reference point $\mathbf{r}$. The hypervolume is the volume of the space dominated by $\mathcal{P}$ and bounded from above by r.

3.7 Dominated Hypervolume

Given a reference point $\mathbf{r} \in \mathbb{R}^{\nu}$, the dominated hypervolume (HV) of a Pareto set $\mathcal{P}$ is the volume of the space dominated by $\mathcal{P}$ and bounded from above by $\mathbf{r}$ [53]. The coordinate values of the reference point should be set to slightly worse than the values a decision maker would tolerate. Figure 3.5 illustrates the hypervolume in a two-objective problem. The dominated hypervolume is also known as the hypervolume indicator.

3.8 Bayesian Optimization

Bayesian optimization [84] [85] is an optimization procedure which is basically composed of three main elements: an objective function; a surrogate function; and an acquisition function. The objective function is the unknown function we wish to optimize. Since it is unknown, that is, it does not have a closed form, we cannot estimate its derivatives, in order to use methods such as gradient descent. To further complicate things, it is usually a noisy function, and, in general, very expensive to evaluate. The surrogate function is a probabilistic model of the objective function that we build based on previously observed samples. It is very common to use a Gaussian Process model as surrogate function. The Gaussian Process model returns not only the estimated value of the function at a given domain point (the mean) but also a level of uncertainty (the variance). Finally, the acquisition function is a function of the mean and variance values returned by the Gaussian Process Model which is used to determine which point of the domain should be evaluated next using the costly objective function. Once a new point has been evaluated using the expensive-to-compute function, the Gaussian Process Model is updated, and a new point from the domain is selected using the acquisition function. This process is continued as long as the user desires.

There are many acquisition functions that can be used with Bayesian optimization. One of the most common is the expected improvement (EI) [81]. Let improvement be defined as

$$I(c) = \begin{cases} 0, \text{ if } c > c^{\min} \\ \\ c^{\min} - c, \text{ otherwise} \end{cases}, \tag{3.18}$$

where c is an arbitrary objective value, and $c^{\min}$ is the best estimate of the objective minimum so far. Then the EI can be written as:

$$\mathrm{EI}(\mathbf{h}) = \int_{-\infty}^{c^{\min}} I(c)\mathrm{PDF}_{\mathbf{h}}(c)dc, \tag{3.19}$$

where $\mathrm{PDF}_{\mathbf{h}}$ denotes the estimated probability density function of the objective for domain point $\mathbf{h}$. The EI can be extended to noisy settings by treating the current best objective value as a random variable as well, and obtaining the corresponding expected value of expression (3.19) [86]. The resulting integral does not have a closed form, but can be handled with Monte-Carlo integration [86] [87].

The simplest, and also the most limited, approach to tackle multiple objectives is Scalarization. Scalarization turns a multi-objective optimization problem into a single-objective one, which can be handled with single-objective Bayesian optimization, for example. One of the most common scalarization methods is the weighted sum approach [53].

The main drawback of scalarization is that it is not truly multi-objective optimization. It does not approximate the Pareto frontier. It creates a new objective, which is a function of the multiple-objectives, and optimizes it instead.

The EI acquisition function can be extended to multi-objective optimization by defining the improvement in terms of the hypervolumes of two estimates of the Pareto front. Let E_t be the current set of non-dominated solutions. Then we can define the multi-objective improvement as [81]

$$I(\mathbf{c}(\mathbf{h})) = \begin{cases} \mathrm{HV}(E_t \cup \{\mathbf{c}(\mathbf{h})\}) - \mathrm{HV}(E_t), \text{ if } E_t \text{ non-dominates } \mathbf{c}(\mathbf{h}) \\ 0, \text{ otherwise} \end{cases}. \tag{3.20}$$

The expected hypervolume improvement (EHVI) can be defined as:

$$\mathrm{EHVI}(\mathbf{h}) = \int_{\mathbf{c}\in V_{nd}} I(\mathbf{c})\mathrm{PDF}_{\mathbf{h}}(\mathbf{c})d\mathbf{c}, \tag{3.21}$$

where $\mathbf{c}$ is the vector of multiple objectives, and $V_{nd} := \{\mathbf{c} : \mathbf{c} \text{ is non-dominated by } E_t\}$. The EHVI can also be extended to noisy settings in a similar way to EI, by integrating over the uncertainty in the function values at the observed points [88].

ParEGO is a multi-objective Bayesian Optimization method which relies on scalarization to approximate the whole Pareto front [89]. It is based on the assumption that the Pareto front can be approximated by using a different scalarization with the acquisition function at each iteration. Each time, with a different weight vector. With this approach, an approximation to the whole Pareto front can be gradually built up [89]. ParEGO can be easily extended to the noisy setting by using the noisy version of the EI [90].

3.9 Variational Image Compression

Variational image compression [26] [27] is a form of transform coding using neural networks. In traditional transform coding of images, the input image $\boldsymbol{\psi}$ is transformed from pixel domain to another more suitable domain, in which loss of information due to quantization is more tolerated. The transformed data is then quantized and entropy coded, using arithmetic coding, for example. The transformation maps a point in the original pixel domain, $\boldsymbol{\psi}$ (with number of dimensions equal to the number of pixels), to a point in the transformed domain, $\boldsymbol{\omega}$ (with number of dimensions depending on the transform).

Neural networks operate in continuous domains. They require differentiable operations in order to be trained using gradient descent. Quantization is a real problem for neural networks, because the derivative of the quantization function is zero almost everywhere. Variational image compression is able to circumvent this problem, by essentially replacing quantization with the addition of uniform noise, in the range $[-1/2, 1/2]$, during training.

Let $\hat{\boldsymbol{\omega}}$ (with associated random variable $\hat{\boldsymbol{\Omega}}$) denote the quantized version of the data in transformed domain, and let $\tilde{\boldsymbol{\omega}}$ (with associated random variable $\tilde{\boldsymbol{\Omega}}$) denote the noisy version. Initially, the input image $\boldsymbol{\psi}$ is transformed by a neural network, which outputs $\boldsymbol{\omega}$. Then, uniform noise in the range $[-1/2, 1/2]$ is added, giving $\tilde{\boldsymbol{\omega}}$ with distribution $q_{\tilde{\boldsymbol{\Omega}}|\boldsymbol{\Psi}}(\tilde{\boldsymbol{\omega}}|\boldsymbol{\psi})$. During training, the noisy version of the transformed data is used, but during inference, actual quantization is performed.

A model for the unconditional prior probability distribution of the quantized representation is also required in order to encode it using entropy coding. An important consequence of replacing quantization with the addition of uniform noise, with the same width as the quantization bins, is that the probability density function of the noisy data $p_{\tilde{\boldsymbol{\Omega}}}(\tilde{\boldsymbol{\omega}})$ is a continuous function that interpolates the probability mass function of the quantized data $p_{\hat{\boldsymbol{\Omega}}}(\hat{\boldsymbol{\omega}})$ at integer positions [26]. This is a consequence of the fact that the probability distribution of the sum of two random variables is the convolution of the distributions of the two random variables [91], therefore $p_{\tilde{\boldsymbol{\Omega}}}(\tilde{\boldsymbol{\omega}}) = \int_{\tilde{\omega}-1/2}^{\tilde{\omega}+1/2} p_{\boldsymbol{\Omega}}(\boldsymbol{\omega}) d\boldsymbol{\omega}$. Furthermore, the probability mass function $p_{\hat{\boldsymbol{\Omega}}}(\hat{\boldsymbol{\omega}})$ for a given $\hat{\boldsymbol{\omega}}$ is the integral of $p_{\boldsymbol{\Omega}}(\boldsymbol{\omega})$ over the corresponding quantization bin [26] [27], or $p_{\hat{\boldsymbol{\Omega}}}(\hat{\boldsymbol{\omega}}) = \int_{\hat{\omega}-1/2}^{\hat{\omega}+1/2} p_{\boldsymbol{\Omega}}(\boldsymbol{\omega}) d\boldsymbol{\omega}$. Then $p_{\tilde{\boldsymbol{\Omega}}}(i) = p_{\hat{\boldsymbol{\Omega}}}(i)$, for integer i.

The unconditional prior probability distribution of $\tilde{\boldsymbol{\Omega}}$ is modeled by a non-parametric fully factorized density model $p_{\tilde{\boldsymbol{\Omega}}}(\tilde{\boldsymbol{\omega}})$, and is learned during training [26] [27]. By fully factorized we mean that the components of $\tilde{\boldsymbol{\Omega}}$ are independent from each other. By non-parametric we mean that no assumption is made about the shape of the distribution of each component of $\tilde{\boldsymbol{\Omega}}$. This model is trained to minimize the negative expected likelihood [26]:

$$-\mathbb{E}_{\tilde{\boldsymbol{\Omega}}} \sum_i \left[p_{\tilde{\Omega}_i}(\tilde{\omega}_i; \boldsymbol{\phi}^{(i)})\right] \tag{3.22}$$

Where $p_{\tilde{\Omega}_i}$ is the probability distribution of one $\tilde{\Omega}_i$, which is one component of $\tilde{\boldsymbol{\Omega}}$, and $\boldsymbol{\phi}^{(i)}$ represents the parameter vector for model $p_{\tilde{\Omega}_i}$. The encoder portion of the variational autoencoder is illustrated in the top part of Figure 3.6.

Decoding is also performed with an uncertainty. Given some vector $\tilde{\boldsymbol{\omega}}$ in transformed domain, the probability of the original image being $\boldsymbol{\psi}$ is assumed to be $p_{\boldsymbol{\Psi}|\tilde{\boldsymbol{\Omega}}}(\boldsymbol{\psi}|\tilde{\boldsymbol{\omega}}) = \mathcal{N}(\boldsymbol{\psi}; \tilde{\boldsymbol{\psi}}, (2\lambda)^{-1}\mathbf{I})$, where $\tilde{\boldsymbol{\psi}}$ is the output of a synthesis transform (a neural network) and $\mathcal{N}(\boldsymbol{\psi}; \tilde{\boldsymbol{\psi}}, (2\lambda)^{-1}\mathbf{I})$ denote a multivariate normal distribution with mean vector $\tilde{\boldsymbol{\psi}}$ and covariance matrix $(2\lambda)^{-1}\mathbf{I}$, where $\mathbf{I}$ is the identity matrix and λ is a constant. Again this specific choice of normal distribution is not arbitrary. It is chosen so that

$$\begin{aligned} -\ln(p_{\boldsymbol{\Psi}|\tilde{\boldsymbol{\Omega}}}(\boldsymbol{\psi}|\tilde{\boldsymbol{\omega}})) &= -\ln\left(\text{const.} \times \exp\left(-\frac{1}{2}(\boldsymbol{\psi} - \tilde{\boldsymbol{\psi}})^T \boldsymbol{\Sigma}^{-1}(\boldsymbol{\psi} - \tilde{\boldsymbol{\psi}})\right)\right) \\ &= \lambda\|\boldsymbol{\psi} - \tilde{\boldsymbol{\psi}}\|^2 + \text{const.}, \end{aligned} \tag{3.23}$$

where $\boldsymbol{\Sigma} = (2\lambda)^{-1}\mathbf{I}$ is the covariance matrix. This term appears in the loss function of the variational autoencoder. The decoder portion of the variational autoencoder is illustrated in the bottom part of Figure 3.6.

In variational inference terms [92], the encoder is linked to the inference model, while the decoder is linked to the generative model. The objective of the generative model is to generate the image from the latent representation, and the objective of the inference model is to infer the latent representation from the image. The true posterior distribution $p_{\tilde{\boldsymbol{\Omega}}|\boldsymbol{\Psi}}(\tilde{\boldsymbol{\omega}}|\boldsymbol{\psi})$ of the latent representation is assumed intractable. Therefore, it is approximated by the parametric variational density $q_{\tilde{\boldsymbol{\Omega}}|\boldsymbol{\Psi}}(\tilde{\boldsymbol{\omega}}|\boldsymbol{\psi})$. The variational autoencoder is trained to minimize the Kullback-Leibler divergence between $q_{\tilde{\mathbf{\Omega}}|\mathbf{\Psi}}(\tilde{\boldsymbol{\omega}}|\boldsymbol{\psi})$ and $p_{\tilde{\mathbf{\Omega}}|\mathbf{\Psi}}(\tilde{\boldsymbol{\omega}}|\boldsymbol{\psi})$ for every $\boldsymbol{\psi}$ in the training set, where $\boldsymbol{\psi}$ has distribution $p_{\mathbf{\Psi}}(\boldsymbol{\psi})$:

$$\mathbb{E}_{\mathbf{\Psi}\sim p_{\mathbf{\Psi}}}[D_{\mathrm{KL}}[q_{\tilde{\mathbf{\Omega}}|\mathbf{\Psi}}||p_{\tilde{\mathbf{\Omega}}|\mathbf{\Psi}}]] = \mathbb{E}_{\mathbf{\Psi}\sim p_{\mathbf{\Psi}}}[\mathbb{E}_{\tilde{\mathbf{\Omega}}\sim q_{\tilde{\mathbf{\Omega}}|\mathbf{\Psi}}}[\ln q_{\tilde{\mathbf{\Omega}}|\mathbf{\Psi}}(\tilde{\boldsymbol{\omega}}|\boldsymbol{\psi}) - \ln p_{\tilde{\mathbf{\Omega}}|\mathbf{\Psi}}(\tilde{\boldsymbol{\omega}}|\boldsymbol{\psi})]] =$$
$$\mathbb{E}_{\mathbf{\Psi}\sim p_{\mathbf{\Psi}}}[\mathbb{E}_{\tilde{\mathbf{\Omega}}\sim q_{\tilde{\mathbf{\Omega}}|\mathbf{\Psi}}}[\ln q_{\tilde{\mathbf{\Omega}}|\mathbf{\Psi}}(\tilde{\boldsymbol{\omega}}|\boldsymbol{\psi}) - \ln p_{\mathbf{\Psi}|\tilde{\mathbf{\Omega}}}(\boldsymbol{\psi}|\tilde{\boldsymbol{\omega}}) - \ln p_{\tilde{\mathbf{\Omega}}}(\tilde{\boldsymbol{\omega}})]] + \text{const.} \quad (3.24)$$

The first term of expression (3.24) is zero, because the probability density function $q_{\tilde{\mathbf{\Omega}}|\mathbf{\Psi}}(\tilde{\boldsymbol{\omega}}|\boldsymbol{\psi})$ is the product of several uniform distributions with width 1, and therefore is equal to 1 for every $\tilde{\boldsymbol{\omega}}$ with non-zero density [27]. The second term is the weighted distortion (Equation (3.23)). Finally, the third term $\mathbb{E}_{\mathbf{\Psi}\sim p_{\mathbf{\Psi}}}[\mathbb{E}_{\tilde{\mathbf{\Omega}}\sim q_{\tilde{\mathbf{\Omega}}|\mathbf{\Psi}}}[-\ln(p_{\tilde{\mathbf{\Omega}}}(\tilde{\boldsymbol{\omega}}))]]$ is the *differential* (*continuous*) cross-entropy between the marginal $\mathbb{E}_{\mathbf{\Psi}\sim p_{\mathbf{\Psi}}}[q_{\tilde{\mathbf{\Omega}}|\mathbf{\Psi}}(\tilde{\boldsymbol{\omega}}|\boldsymbol{\psi})]$ and the prior $p_{\tilde{\mathbf{\Omega}}}(\tilde{\boldsymbol{\omega}})$. Both the distortion and cross-entropy terms in expression (3.24) closely approximate the quantization error and the average codeword length of the variational autoencoder [26] [27].

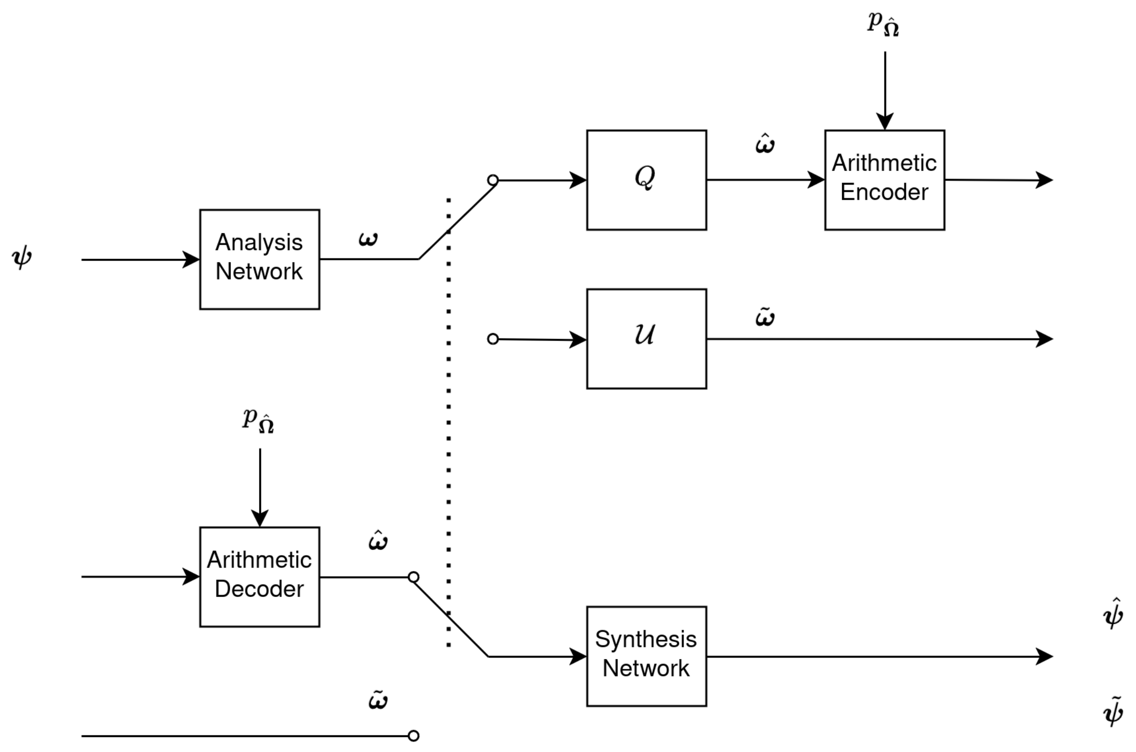

$p_{\hat{\Omega}}$
$\hat{\boldsymbol{\omega}}$
Arithmetic Encoder
Q
$\boldsymbol{\psi}$
Analysis Network
$\boldsymbol{\omega}$
$\mathcal{U}$
$\tilde{\boldsymbol{\omega}}$
$p_{\hat{\Omega}}$
Arithmetic Decoder
$\hat{\boldsymbol{\omega}}$
Synthesis Network
$\hat{\boldsymbol{\psi}}$
$\tilde{\boldsymbol{\omega}}$
$\tilde{\boldsymbol{\psi}}$

Figure 3.6: Block diagram of the variational autoencoder. The input image $\boldsymbol{\psi}$ is initially transformed by an analysis neural network. The output of the analysis network $\boldsymbol{\omega}$ is then quantized during inference, which is represented by the "Q" block, or is added with uniform noise during training, which is represented by the "$\mathcal{U}$" block. The noisy version of the transformed image is represented by $\tilde{\boldsymbol{\omega}}$, while the quantized version is represented by $\hat{\boldsymbol{\omega}}$. The quantized data is arithmetically encoded and then decoded using an also learned probability mass function $p_{\hat{\Omega}}$. Then, a synthesis network converts the data back to pixel domain. The output is either $\hat{\boldsymbol{\psi}}$ or $\tilde{\boldsymbol{\psi}}$, depending if the input was $\hat{\boldsymbol{\omega}}$ or $\tilde{\boldsymbol{\omega}}$.

Chapter 4

Perceptron Coding

4.1 Motivation

Our objective is to propose a neural-based replacement for the LUT, which is used in many data compression standards. We explained the LUT in detail in Section 2.5. In data compression, LUTs are often used as context models for arithmetic coding. Broadly speaking, a context model can be seen as a function $\boldsymbol{f} : \mathbb{R}^M \to \mathbb{R}^S$, where M is the number of values that make up the context, and S is the number of symbols. The context model returns the estimates of the conditional probabilities of the symbols given the context.

The LUT is optimal when applied to the observed data. It gives the MLE of the conditional probabilities, which also translates to achieving the minimum average codeword length. However, the LUT is also maximally overfit. It has a huge number of parameters, one for each combination of symbol and context. The number of parameters of the LUT is $(S^M)S = S^{M+1}$, where S is the number of possible symbols and M is the context size. It suffers from the zero-frequency problem or context dilution [57], which happens when a symbol to be encoded has not been encountered before for a given context. In such a case, the conditional probability of the symbol is zero, and, in theory, it would take an infinite number of bits to encode it, unless any precaution has been taken, such as starting with a count of 1 for every symbol. This is a clear case of overfitting, and it happens more frequently for larger numbers of LUT parameters, or larger context size M.

In theory, neural networks can serve as drop-in replacements for LUTs. Neural networks, such as MLPs, are universal approximators, and can therefore approximate any continuous function. Although convolutional neural networks are also universal approximators [75], they are more stringent in their assumptions about the input data. MLPs are more general and flexible for different types of data. Besides universal approximation, with neural networks it is possible to control the number of parameters, trading off generalization error (or variance, or error in unobserved data) for approximation error (or bias, or error in observed data).

Let the context be represented by a vector $\boldsymbol{\xi} \in \mathbb{R}^M$. We represent the context as a vector of real numbers, even though the symbols are generally integers, because the context is the input of the neural network, and the neural network accepts any real number as input. Let the neural network without activation function be represented by the function $\hat{\boldsymbol{f}}(\boldsymbol{\xi}, \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ represents the network parameters. Then, the neural network output with activation can be represented as $\boldsymbol{f}(\boldsymbol{\xi}, \boldsymbol{\theta}) = \text{Softmax}(\hat{\boldsymbol{f}}(\boldsymbol{\xi}, \boldsymbol{\theta}))$. These are the conditional probability estimates of the symbols given the context $\boldsymbol{\xi}$. In order to train the neural network, we need a way to tell if it is doing a good job or not. In other words, we need a cost function.

In Section 2.2 we have seen that an ideal arithmetic coder compresses a sequence $x_1, x_2, ..., x_N$ with $-\log_2(P(x_1, x_2, ..., x_N))$ bits. This can be expanded as

$$-\log_2(P(x_1, x_2, ..., x_N)) = -\log_2(P(x_1)) - \log_2(P(x_2|x_1)) - ... - \log_2(P(x_N|x_{N-1}, ..., x_1)). \tag{4.1}$$

That is, each symbol in the sequence is compressed with a number of bits equal to the negative log of its conditional probability. Then, we can train the neural network to minimize the negative log of the symbol probabilities output by the network. This can be mathematically expressed as

$$L = -\sum_{n=1}^{N} \log_2(\mathbf{t}_n^T \boldsymbol{f}(\boldsymbol{\xi}_n, \boldsymbol{\theta})), \tag{4.2}$$

where $\mathbf{t}_n$ and $\boldsymbol{\xi}_n$ are the one-hot-encoding and the context for the n-th symbol in the sequence, x_n, and $\boldsymbol{f}(\boldsymbol{\xi}_n, \boldsymbol{\theta})$ are the estimated symbol probabilities given the context $\boldsymbol{\xi}_n$. Note that, by the definition of one-hot-encoding, $\mathbf{t}_n$ is a binary vector with only a single nonzero entry, which is associated with a particular value of x_n. Therefore, $\mathbf{t}_n^T \boldsymbol{f}(\boldsymbol{\xi}_n, \boldsymbol{\theta})$ selects the component of $\boldsymbol{f}(\boldsymbol{\xi}_n, \boldsymbol{\theta})$ associated with the symbol x_n.

Note that this is also equivalent to seeking to maximize the probabilities that are output by the network for the observed symbols. It is also equivalent to seeking the MLE of the conditional probabilities, which, as we know from Section 2.5, are the relative frequencies stored by the LUT. If there are enough parameters in the network, it could learn to exactly reproduce the LUT. For example, a single layer of neurons, with one neuron per conditional probability, could simply learn to store the relative frequencies, one per neuron. However, by limiting the number of parameters of the network, we force it to make more interesting extrapolations about the underlying structure of the data, which can better generalize in the case of unobserved data.

The loss function in (4.2) is the cross-entropy loss. If we group the terms in (4.2) for which the values of x_n are the same, expression (4.2) is then the sum of the negative log of the probability estimates of the possible symbols, multiplied by their absolute frequencies. This is equivalent to the cross-entropy we have seen in Section 2.3 when discussing probability estimation, except for a normalization factor. By minimizing it we minimize the KLD between the observed and estimated probability distributions. It approaches the entropy of the source according to the observed distribution. The cross-entropy loss is also extensively used in machine learning to train classification models. However, the main objective there is not to minimize the average codeword length, but to find a good estimate of the membership probabilities in a more practical way [62].

4.2 Perceptron Coding

We call the neural based method we developed to replace the LUT as perceptron coding (PC). We transform any input data into binary data beforehand. This is so for the same reason as CABAC: in order to simplify context modeling. The simplification comes into two forms: first, with fewer possible symbols, there are fewer possible contexts for the same context size M. This means that the same contexts occur more frequently, and the possibility of coming across unseen contexts is lower. Second, for each context, there are fewer conditional probabilities. These two factors simplify the neural network training and learning.

As neural network architecture, we use MLPs, two hidden layers, and ReLU activation in the hidden neurons. MLPs are universal approximators, and can therefore approximate any continuous function, provided there are enough neurons. CNNs are also universal approximators [75], but MLPs make fewer assumptions about the input data. CNNs were developed for computer vision tasks, and they learn filters which are capable of extracting interesting visual features in the input data. With MLPs, each input may have a different meaning, they do not have to be all neighboring pixels from the same image for example.

We use two hidden layers because, although even one hidden layer would be sufficient, if there were enough neurons in that layer, with two hidden layers we increase the number of ways the neural network can learn the same function [74]. We could further increase the number of layers. However, the training becomes harder the larger the number of layers, because of the vanishing gradient problem [74]. Therefore, we keep the number of hidden layers equal to two as a middle-ground compromise.

In regards to the number of neurons in each layer, in theory, we should define the number of neurons in each layer according to the amount of approximation error that we consider acceptable. However, recall that we do not only want the network to approximate the function on the training set, but we also want it to generalize well to other unseen data. For that, it is important that the number of parameters is not too high to avoid overfitting. In any case, it is expected that the complexity of the underlying function increases with the context size M, and, as a consequence, it is also necessary to increase the number of neurons in each layer to keep the approximation error low. For this reason, we use a number of hidden neurons per hidden layer which is proportional to the context size M.

We use $64M$ units in the first hidden layer and $32M$ units in the second hidden layer. It is important to note that these numbers are somewhat arbitrary. These are numbers which we assume to yield good approximation and generalization errors. Later results confirm this to some extent. Note that, in this section, our focus is not on the neural network design process. We explore the problem of hyperparameter selection in more depth in Section 5.8. Note that we use more neurons in the first hidden layer compared to the second hidden layer. This pyramid structure is based on a common design strategy present in many works that use MLPs, for example [22], [93]. In Section 5.8, we also present some results that corroborate this common design strategy.

Since the data is converted to binary form, the network may go through some simplifications compared to what we have previously discussed. The network only needs to output the probability of the bit 1, since the probability of the bit 0 is its complement. Therefore, the network without activation may be represented by a function $\hat{f}(\boldsymbol{\xi}, \boldsymbol{\theta})$ and its output with activation by $f(\boldsymbol{\xi}, \boldsymbol{\theta}) = \sigma(\hat{f}(\boldsymbol{\xi}, \boldsymbol{\theta}))$, where σ represents the sigmoid activation function. The cost function may be simplified to:

$$L = \sum_{n=1}^{N} -x_n \log_2(f(\boldsymbol{\xi}_n, \boldsymbol{\theta})) - (1 - x_n) \log_2(1 - f(\boldsymbol{\xi}_n, \boldsymbol{\theta})), \tag{4.3}$$

which is known in the machine learning literature as the binary cross entropy loss.

Since the hidden layers have $64M$ and $32M$ neurons, the total number of parameters of the network is $2112M^2 + 128M + 1$. This comes from the sizes of the weight matrices and bias vectors representing each layer. The growth in M sharply contrasts with the growth of parameters in the LUT. For binary symbols, it suffices to store the conditional probabilities of the bit 1. Then, the number of parameters in the LUT is 2^M instead of 2^{M+1}. Note that, for $M > 19$, there are more LUT entries than parameters in the network!

4.3 Adaptive Perceptron Coding

So far, we have described how the MLP can be used in place of the LUT in forward coding. In our description, we have assumed that the networks would be trained using batch gradient descent with a dataset of size N, and either the loss function in (4.2) or (4.3). However, this overlooks a major advantage of the LUT as used in many applications: the LUT is usually adaptive. It is continuously updated after each new sample is coded, it does not require previous training, and it can track changes in source statistics, in the case of non-stationary processes.

It is important to remember that, in the context of this work, the networks are intended to drive arithmetic coding engines. This means that the networks in the encoder and the decoder must estimate the same probabilities, otherwise reconstruction is not possible. Therefore, in order to use the MLP in place of the LUT in online coding, the networks in the encoder and the decoder must go through the same sequence of states. For that, they must be initialized the same way in both the encoder and the decoder, and the network updates must also be the same at each corresponding encoding/decoding step. Next, we describe a method that is able to achieve this. We call our proposed method adaptive perceptron coding (APC).

We have seen in Section 3.2 that training is often carried one batch at a time, instead of one update for the whole data all at once. The loss function in (4.2) may be broken into portions attributed to different batches of data as

$$L = \sum_i L_{\mathcal{B}_i}, \tag{4.4}$$

$$L_{\mathcal{B}_i} = -\sum_{n \in \mathcal{B}_i} \log_2(\mathbf{t}_n^T \boldsymbol{f}(\boldsymbol{\xi}_n, \boldsymbol{\theta})), \tag{4.5}$$

where $\{\mathcal{B}_i\}$ is a partition of the set $\{1, 2, ..., N\}$ such that all $\mathcal{B}_i$ have the same size B, except maybe for one i, in case N is not divisible by B. Also, the elements that compose each batch are consecutive to each other.

In an online coding setting, data coding and stochastic gradient descent can occur in parallel using the following procedure. The neural network is randomly initialized, with care taken to initialize it the same way in both the encoder and the decoder, using the same random seed and the same pseudo random number generator. The first B samples are coded using the randomly initialized model. Since the neural network has been randomly initialized, the initial probability distribution is close to uniform, as can be experimentally verified. After B samples have been coded, one update of the network parameters is performed using the B previously coded samples. The next B samples are coded using the updated model, and, after that, a new update on the network parameters is made. This goes on until there is no more data to be coded.

One characteristic of this approach is that the network does not go through the same data twice. Therefore, every new sample is a different sample from the distribution, and the network continuously learns to approach the true probability distribution, instead of the training set distribution [64]. Even if the distribution changes its characteristics, in the case of a non-stationary process, the network tries to converge to the new statistics.

The important is that the process statistics do not change quicker than the neural network can converge. In fact, we want the neural network to converge faster than the process would change its statistics. In practice, however, there are many factors into play. The learning rate, the neural network size and architecture, the optimization algorithm, the learning rate schedule, if present, all can affect how quickly the network can converge, and how large the generalization error will be. There is also the nature of the data itself to take into account and how quickly it changes its statistics. All this can affect the overall coder performance.

In theory, this method could be used with any S-ary data. However, as before, we assume that any input data has been converted to binary to simplify context modeling.

In this case, the network may have only a single output, the probability of bit 1, and we can use the binary-cross-entropy loss (Equation (4.3)). Also, although the batch size B can be freely chosen, we use it fixed at $B = 1$ in order to maximize adaptivity. The value of B is associated with the tradeoff between running time and adaptivity. It controls the frequency of updates, which affects positively the adaptivity, but negatively the running time. The frequency of updates affect the running time in two ways. Firstly, inference requires only one forward pass in the network, but a training update also requires a backward pass of backpropagation. Therefore, one update requires roughly twice the time of a single inference. Secondly, using multiple samples at once (for inference or training) is more efficient than processing a single sample at a time, because of parallelism. Another factor which affects the running time is the network architecture, since the larger the network architecture, the longer it takes to complete a forward or backward pass.

The method with $S = 2$ and $B = 1$ may be described as follows. Let the neural network without activation function at instant n be represented by $\hat{f}(\boldsymbol{\xi}_n, \boldsymbol{\theta})$. The output with activation function may be represented by $f(\boldsymbol{\xi}_n, \boldsymbol{\theta}) = \sigma(\hat{f}(\boldsymbol{\xi}_n, \boldsymbol{\theta}))$ where $\boldsymbol{\xi}_n$ represents the context at instant n. Since the data has been converted to binary, the contribution of the current sample to the loss function may be represented by

$$L_n = -x_n \log_2(f(\boldsymbol{\xi}_n, \boldsymbol{\theta})) - (1 - x_n) \log_2(1 - f(\boldsymbol{\xi}_n, \boldsymbol{\theta})). \tag{4.6}$$

The network parameters $\boldsymbol{\theta}_n$ may be updated after coding this sample by

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \epsilon(\nabla_{\boldsymbol{\theta}} L_n)_{\boldsymbol{\theta}_n} \tag{4.7}$$

where ϵ is the learning rate.

In APC, we use the same context-size-dependent network architecture that was described in Section 4.2. The initial network is obtained by randomly initializing the network parameters. We guarantee that the initial parameters are the same in both the encoder and the decoder by using the same random number generator seed, and the same pseudo random number generator algorithm. A common initialization method of network parameters is Xavier initialization [94]. We use a slightly modified version of Xavier initialization based on the fact that more diverse initial values in a layer is often better [64]. Let $J^{(d)}$ denote the number of nodes in the d'th layer of a MLP. $J^{(0)} = M$ is the number of inputs and $J^{(d_{\max})} = 1$ is the number of outputs. Instead of initializing the weights or biases of the d'th layer, except the input layer, by sampling a uniform distribution in the interval $(-1/\sqrt{J^{(d-1)}}, 1/\sqrt{J^{(d-1)}})$, we take their values from a random permutation of equally spaced values in this interval. We have experimentally confirmed that this modification leads to consistently better results. The encoding results for the APC are shown in Section 4.7.

4.4 Adaptive Coding with Recurrent Neural Networks

Adaptive context modeling with neural networks has previously been tackled to some extent in a few previous works ([25], [32], [33]), for the particular case of textual data. Because of the sequential nature of this type of data, in general, these methods consisted in predicting next word or character using RNNs. The RNN weights are usually updated after processing each sample.

In [32] and [33], RNN's pre-trained weights are continuously updated during evaluation, in the context of language modeling. The RNN method in [32], consists of training an Elman network with SGD updates at every time step, with fully truncated backpropagation through time (TBPTT). The method in [33], improves the method in [32] essentially by using: LSTMs, updates for slightly larger chunks of sequence (5-20), backpropagation over more time steps, and regularization. In [25], a hybrid model, composed of one pre-trained bootstrap model, and a continuously updated supporter model, is used to drive arithmetic coding in text compression. The context is composed of a fixed number of previous samples. The input sequence is divided into 64 equally sized parts, and predictions are generated for each part in a single batch. Weight updates are performed after encoding/decoding blocks of 20 symbols (per part).

Inspired by these previous works, we developed a compression method based on RNNs to compare with APC. We call the RNN method we developed adaptive RNN (ARNN). In this method the RNN architecture consists of two GRU layers with 650 hidden units each. This architecture is similar to the one used in [33]. Let the output of the RNN without activation be represented by either $\hat{f}(\mathbf{x}^{(n-1)}, \boldsymbol{\theta})$ or $\hat{\boldsymbol{f}}(\mathbf{x}^{(n-1)}, \boldsymbol{\theta})$, and the output with activation by either $f(\mathbf{x}^{(n-1)}, \boldsymbol{\theta}) = \sigma(\hat{f}(\mathbf{x}^{(n-1)}, \boldsymbol{\theta}))$ or $\boldsymbol{f}(\mathbf{x}^{(n-1)}, \boldsymbol{\theta}) = \text{Softmax}(\hat{\boldsymbol{f}}(\mathbf{x}^{(n-1)}, \boldsymbol{\theta}))$, depending if the data is binary or S-ary. In the case of S-ary data, the RNN outputs correspond to the probabilities of the next element in the sequence, x_n, being each symbol. In the case of binary data, the output corresponds to the probability of the next element in the sequence being 1. The probability of 0 can be obtained by the output's complement, in this case.

The network is updated during coding using SGD and BPTT in the following manner. The initial network weights are randomly obtained, with the random seed shared between the encoder and the decoder. The initial prediction for the first $N' < N$ samples is obtained either using the randomly-initialized network or by assuming a uniform distribution. We opted to use the randomly-initialized network. Since it has been randomly initialized, there is no special trend for the output probabilities to favor one symbol or the other. The initial output probability distribution is close to uniform, which was experimentally verified. Then, after N' samples have been encoded/decoded, the sequence of length N' is used to update the neural network weights using BPTT. The next N' samples are predicted using the newly updated model, and so on. This is continued until there are no more samples to encode/decode. The network parameters are updated following the rule:

$$\boldsymbol{\theta}_{i+1} \leftarrow \boldsymbol{\theta}_i - \epsilon \, \nabla_{\boldsymbol{\theta}} \left(\sum_{n=(i-1)N'+1}^{\min\{iN', N\}} \ell(\mathbf{t}_n, \boldsymbol{f}(\mathbf{x}^{(n-1)}, \boldsymbol{\theta})) \right) \Bigg|_{\theta=\theta_i} , \tag{4.8}$$

where the partial derivatives are obtained with BPTT, $i \in \{1, ..., \lceil N/N' \rceil\}$ and

$\ell(\mathbf{t}_n, \boldsymbol{f}(\mathbf{x}^{(n-1)}, \boldsymbol{\theta})) = -\log_2(\mathbf{t}_n^T \boldsymbol{f}(\mathbf{x}^{(n-1)}, \boldsymbol{\theta}))$. In Equation (4.8), we have assumed S-ary data. The update rule for binary data can be obtained by replacing the cross-entropy loss with the binary cross-entropy loss.

The RNN is fed the same data that goes through the arithmetic coder, in the same order. This means that, in theory, all previous samples can be taken under consideration. In other words, it means that the context size is unbounded. In practice, however, the context size is not unbounded due to numerical reasons, and depends on the capabilities of the RNN, with LSTMs and GRUs having longer memories than naive RNNs.

4.5 Binary Image Datasets

This section is devoted to describing the datasets used in the perceptron coding experiments. All datasets here described consist of binary document images, extracted and processed from scientific journals. We defined six datasets, with different pages selected for training, validation and testing.

The pages from the Datasets 1, 2 and 3, were obtained from the paper [65] which appeared in the 26th volume of the IEEE Transactions on Image Processing (IEEE TIP). The pages were converted from portable document format (PDF) to portable network graphics (PNG) with a value of dots per inch of 93, resulting in images with dimensions of 791×1024 pixels. When converting to greyscale, we used the ITU-R 601-2 luma transform, and when converting to bi-level, all values above 40% the maximum value were set to 1, and all values below or equal were set to 0.

In the experiments using these datasets, we discarded the pixels around the borders for which the context was not complete. Restricting ourselves to these smaller images has no impact on the conclusions taken from the experiments. It corresponds to reducing the margins of the documents, since the pixels discarded correspond to only a small portion of the width and height of the image.

The pages from the Datasets 4, 5 and 6 were extracted from the IEEE Signal Processing Letters (IEEE SPL). The extracted pages were converted to binary following a pipeline similar to that of Datasets 1, 2 and 3. However, the resulting images have dimensions of 768×1024 pixels, and the threshold was separately determined for each image, using Otsu's method [95]. Table 4.1 summarizes the pages used in each dataset and for what purpose (training, validation or testing). Sample pages from the datasets can be viewed in Annex I.

In the experiments using Datasets 4, 5 and 6, we considered the pixels outside of the borders of the image to be equal to 1. This permits encoding all image pixels. Recall that we use the common convention that 1 corresponds to white.

The pages from datasets 4 and 5 were selected at random. Their training and validation sets are the same, but the test set from Dataset 4 has all pages from the test set of Dataset 5 and 90 other pages.

Table 4.1: Datasets used in the perceptron coding experiments, all consisting of binary document images, extracted from scientific journals.

Dataset	Journal	Volume	papers	Training pages	Validation pages	Test pages
Dataset 1	IEEE TIP	26	[65]	p. 3	p. 5	pp. 1,2,4,6-11
Dataset 2	IEEE TIP	26	[65]	pp. 1-4,6-11	p. 5	-
Dataset 3	IEEE TIP	26	[65]	-	-	pp. 1-11
Dataset 4	IEEE SPL	27, 28	multiple	10 (Vol. 27)	5 (Vol. 27)	100 (Vol. 28)
Dataset 5	IEEE SPL	27, 28	multiple	10 (Vol. 27)	5 (Vol. 27)	10 (Vol. 28)
Dataset 6	IEEE SPL	27, 28	[96], [97]	-	p. 2 ([96])	pp. 1-5 ([97])

4.6 Forward Coding of Binary Images

This section deals with perceptron coding with pre-trained neural networks. That is, as described in Sections 4.1 and 4.2. More specifically, in the settings discussed in this section, the neural networks are trained beforehand on a training set, and then used to code other data, be it validation or simply test data.

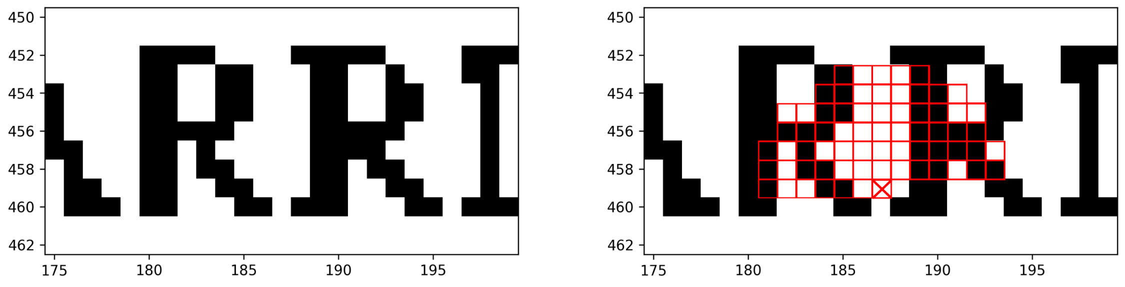


Figure 4.1: To the left: patch of a binary image. To the right: context of size $M = 67$ for the pixel at $(187, 459)$.

In order to train, validate and test our method, we need suitable binary data. It is desirable to use long-memory signals, such as binary images, since we want to test the effects of different context sizes. We opted for binary document images because they are rich in detail and easily accessible. We introduced our binary image datasets in Section 4.5.

We assume that the image pixels are encoded following a forward non-diagonal lexicographical scan order, that is, in a row by row basis, from left to right and from top to bottom. With respect to the context, we selected the M closest pixels in the causal neighborhood to make up the context. These pixels were selected because they have the highest correlation with the current pixel, based on proximity, among the pixels which are available to the decoder. Figure 4.1 shows an illustration of the context used for a specific value of M. Other examples, for other values of M, are depicted in Figure 2.10.

The neural network trainings were done in Pytorch, during 420 epochs, using a learning rate of 10^{-5}, stochastic gradient descent, and a batch size of 2048 [1]. A large number of epochs and small learning rate were selected so the networks converged slowly but steadily. The training and validation losses were monitored to verify convergence. Multiple training and validation trials were performed in order to select the training parameters.

In the first experiment, our intention is to obtain an initial comparison between the MLP and the LUT for different values of context size M. In this section, the LUTs are

[1]All programs used in this work can be accessed at `https://github.com/lucassilvalopes/perceptronac`.

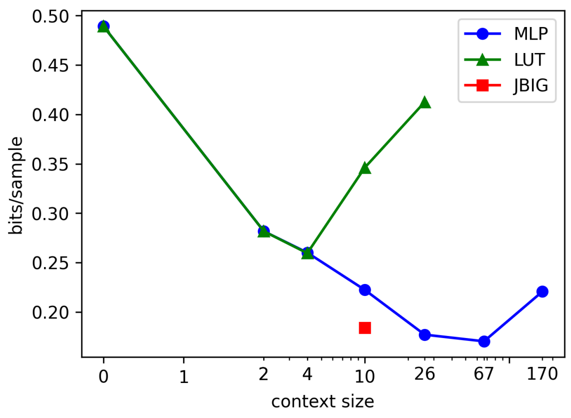
also built prior to encoding, on the same data used to train the neural networks. We use Dataset 1 for this experiment. Figure 4.2 shows the values of bits/sample for exponentially increasing values of context size M, for both the perceptron coding and LUT methods. Both methods use the same context. The results for the LUT method were limited to $M = 26$, because, above this number, the memory requirements become too high. When $M = 0$, the operation corresponds to static AC.

In order to have a reference, we also include values in bits/sample for the pages encoded with JBIG [57]. JBIG is a well-established bi-level image compression standard. It also makes use of context models for binary arithmetic coding, in a similar way to the H.26X standards. In the particular implementation of JBIG that we use [98], the encoder uses 10 neighbor pixels to estimate the probability of the next pixel being zero or one. One out of these 10 pixels is allowed to move up to 8 pixels away horizontally. JBIG also uses other artifices to further enhance its compression capability.

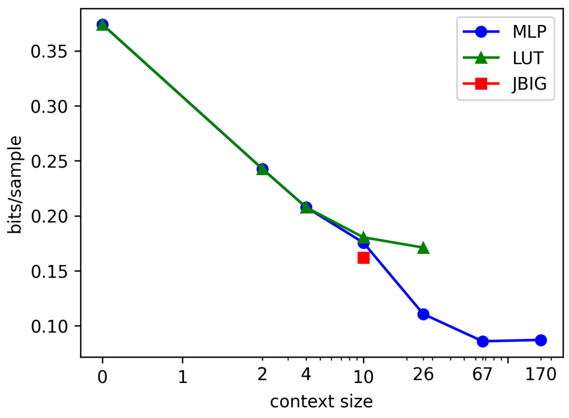
The x-axis in Figure 4.2 is in one of a set of modified logarithmic scales capable of representing values near zero [99]. In this case, the x-axis is linear for values between 0 and 1, and logarithmic for values above 1. This adaptation is necessary because the regular logarithmic scale is not capable of representing values near zero ($\log(x) \to -\infty$ as $x \to 0^+$).

Note how the MLPs approximate the LUT on the training set (Figure 4.2 (c)). The LUT is optimal for the training set and better than the MLP, as expected. For $M = 10$, the MLP lags with respect to the LUT. However, the behavior of the LUT for $M \geq 10$ is a clear indication of overfitting (or context dilution). As M increases, the average codeword length decreases in the training set, but sharply increases in the validation and test sets. The MLPs, on the other hand, only begins overfitting much later, at about $M = 67$. At $M = 67$, the overfitting is still not too noticeable. It is at $M = 170$ that it becomes evident.

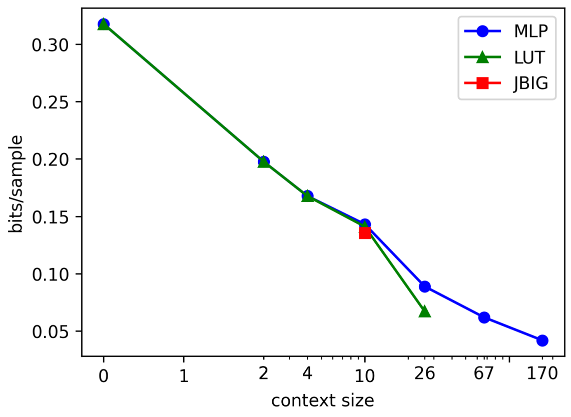
The overfitting of the MLP is probably due to a combination of factors. First, one could argue that for large values of M, the heuristic for the number of hidden units in the



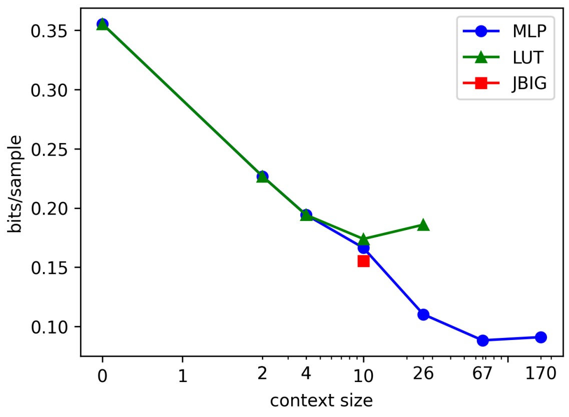
(a) Page 1



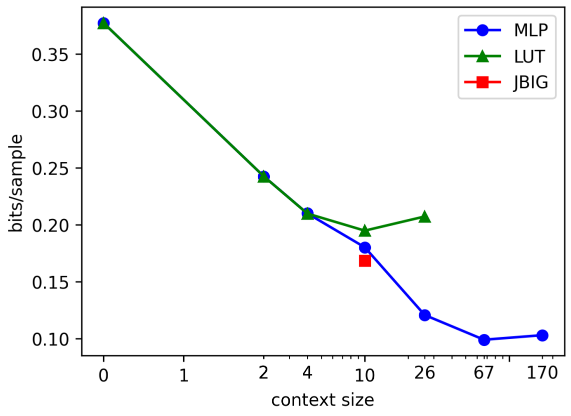
(b) Page 2



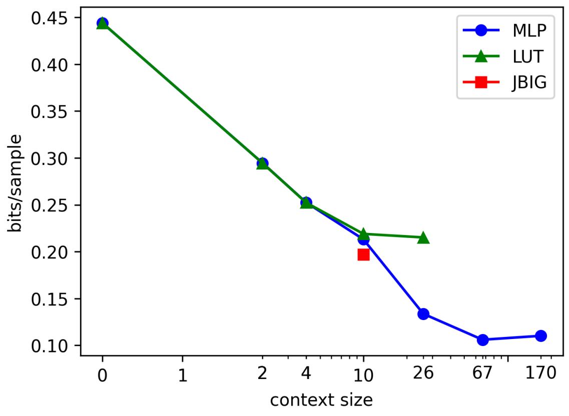
(c) Page 3



(d) Page 4

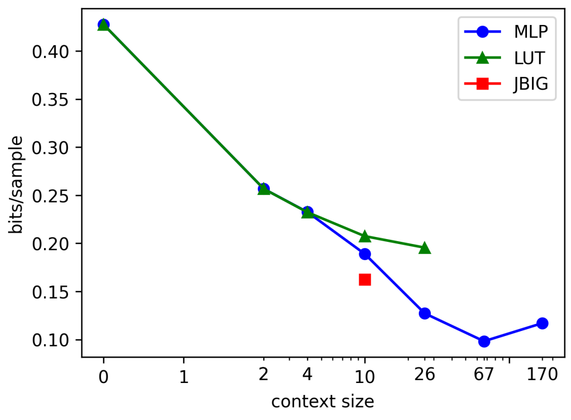


(e) Page 5

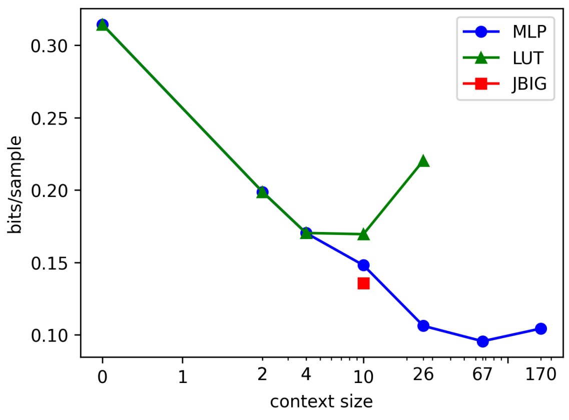


(f) Page 6

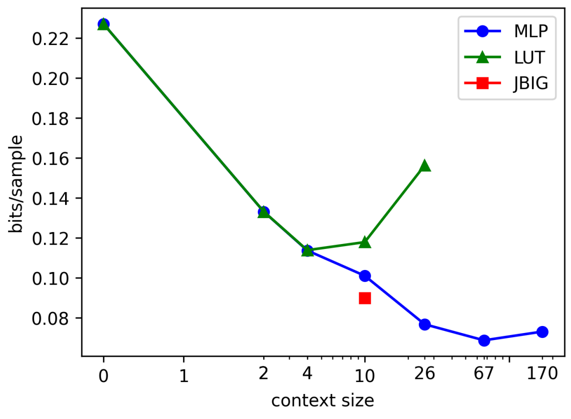
Figure 4.2: ((a) - (f)) Values of bits/sample for the different pages of [65]. Page 3 was used for training and page 5 was used for validation. All other pages were used for testing. The value at zero is the average code length for static binary arithmetic coding (AC). The x-axis is in a modified logarithmic scale capable of representing values near zero [99].



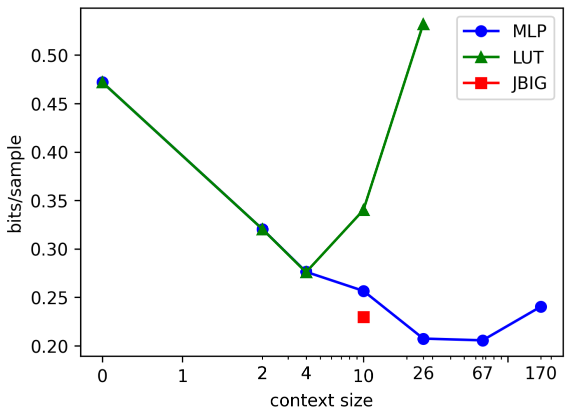
(g) Page 7



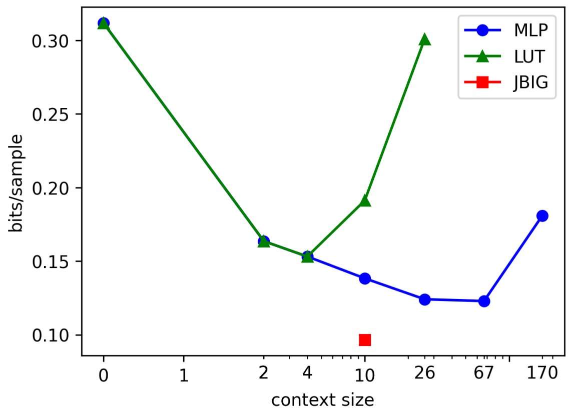
(h) Page 8



(i) Page 9



(j) Page 10



(k) Page 11

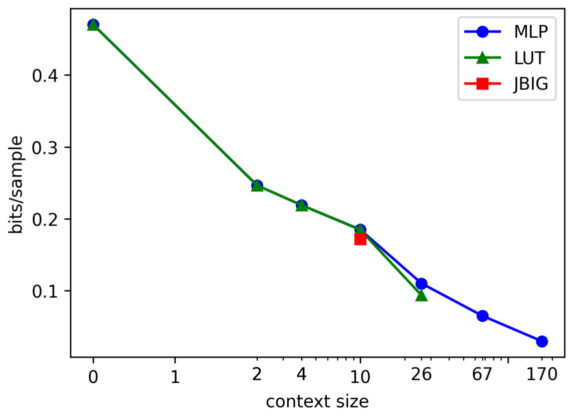
Figure 4.2: ((g) - (k)) Values of bits/sample for the different pages of [65]. Page 3 was used for training and page 5 was used for validation. All other pages were used for testing. The value at zero is the average code length for static binary arithmetic coding (AC). The x-axis is in a modified logarithmic scale capable of representing values near zero [99].

first and second hidden layers leads to too many parameters. This is a valid observation. However, fixing the number of hidden units in the first layer to 2048, and the number of hidden units in the second layer to 1024 for $M > 26$, changes the validation average code lengths only by approximately 10%, and the values of bits per pixel are greater instead of lower. Limiting these values to 1024 and 512, respectively, also has similar effects. Therefore, the number of parameters is not by itself the only reason for this overfitting.

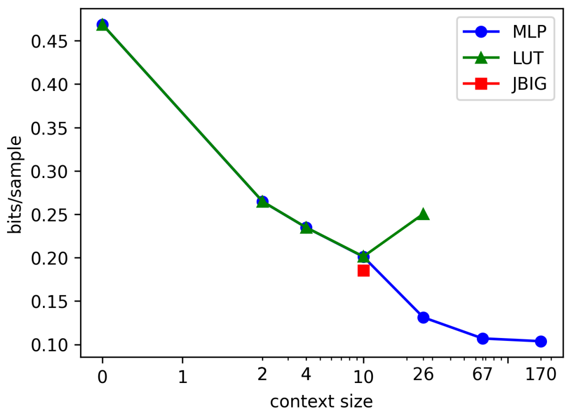
Another factor, which is likely the most significant in this case, is the number of possibilities. This value is a 52-digit number (2^{170}) for $M = 170$, and a 21-digit number (2^{67}) for $M = 67$. There are simply too many possibilities in these two cases. No feasible dataset size would lead to enough significant training examples to prevent the network from overfitting. Considering that one page of 791×1024 pixels leads to 809984 training examples, about 182 trillion pages would be necessary to reach a training set in the order of magnitude of 2^{67}. On the other hand, 2^{26} is only an 8-digit number and, with less than 100 pages, the training set size reaches the order of magnitude of 2^{26}. It is not necessary that all possible contexts are present in the training set, only the ones that are reasonable for the data distribution under consideration need to be present. Nevertheless, their number is also expected to grow exponentially.

We also experimented with more training and test data to verify if the results still hold, for Dataset 4. Figure 4.3 shows the results on the training and test data. The overall appearance of the curves are similar to the observed in the previous experiments, and the relations among the different methods are also similar.

It is well known in the literature that the likelihood of correct generalization depends on the number of networks being considered, the number of networks that give good generalization and the number of training examples [100]. If the size of the network is too large and/or the number of training examples is too small, there will be a vast number of networks consistent with the training data, but only a small portion provides good generalization. In this case, the experiments indicate that the main cause for the observed overfitting is the training set size, which is likely too small for $M > 67$.



(a)



(b)

Figure 4.3: (a) Average code lengths on the 10 randomly selected pages from the 27th Volume of the IEEE Signal Processing Letters used for training. (b) Average code lengths on the 100 randomly selected pages from the 28th Volume of the IEEE Signal Processing Letters used for testing.

In order to study the effect of the training set size on the average code lengths, we progressively increased the number of pages used for training with the neighborhood size fixed at $M = 67$ and verified the average code length on the validation set. We chose $M = 67$ because it was associated with the best results from the previous experiments. The data used was Dataset 2. The results are presented in Figure 4.4. As it can be seen in Figure 4.4, increasing the number of training examples considerably improves the results, until about 7 pages.

What this data is showing is that it becomes increasingly harder to find good examples to use as training data. More and more pages become necessary for the same reduction in bits/sample. Ideally, the number of diverse training samples would be infinite. In this case, the network would continuously learn and would approach the true source probability distribution, instead of the training set distribution [64]. However there are increasing costs in order to keep reducing the average codeword length. A compromise must be made between generalization performance and the investment applied into training and dataset selection.

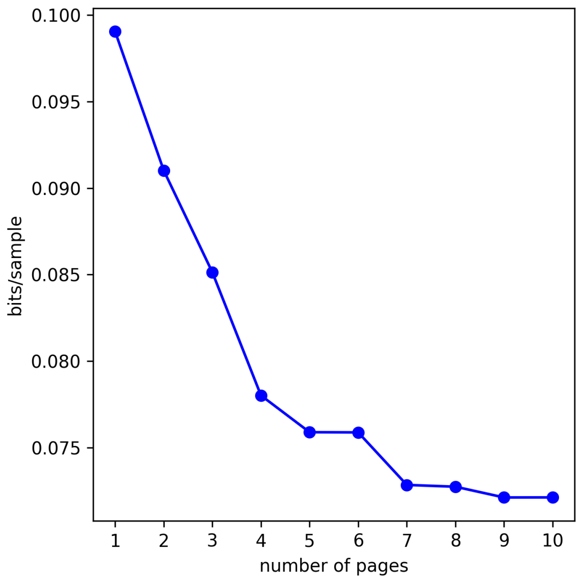


Figure 4.4: Impact of progressively using more pages in order to train the MLP, on the validation average code length . In this figure, the neighborhood size is fixed at 67.

4.7 Backward Adaptive Coding of Binary Images

The previous section showed that the pre-trained MLP can be successfully used in place of the LUT in a forward coding setting. In this section, we compare our proposed method for online coding, APC, with the use of a LUT which is continuously updated based on previously coded samples. We addressed online trained LUTs in Sections 2.5 and 2.6, and the APC method in Section 4.3.

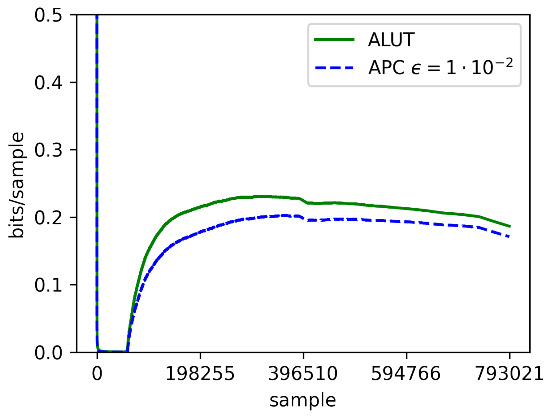
In our particular implementation of the LUT-based method, the probabilities are updated following Equation (2.24), starting the count of every symbol as 1, and therefore with ρ_1 equal to 0.5. Note that the value of α used in H.264 and H.265 in the exponential decay window is very small which leads to very slow adaptation, making it close to the cumulative average of Equation (2.24). Therefore, our method is similar to the one used in H.264 and H.265 in this respect. These standards use specific side information available at the encoder and the decoder to determine the initial probabilities [67] [70], which are not available in our specific application, therefore the best we can do is to assume that the symbols are initially equiprobable. In the following discussion, we call our implementation of the LUT-based method as adaptive LUT, or ALUT, for short.

As before, we opted to test our method on binary document images because of their long-memory, richness in detail and widespreadness. The datasets used in the experiments are described in Section 4.5. Again, the contexts used, in both APC and ALUT, are the M closest pixels in the causal neighborhood. Exemplary contexts are shown in Figures 2.10 and 4.1.

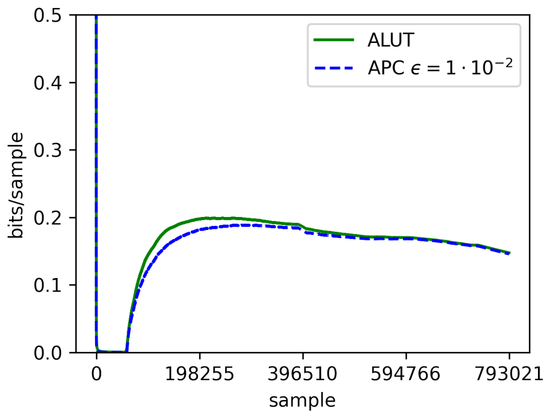
Figure 4.5 shows how APC and ALUT compare to each other, for a particular selection of hyperparameters, when coding the different pages from Dataset 3. The results are for context size $M = 26$ and learning rate $\epsilon = 0.01$. The first pixel of each page has no priors from where to infer probabilities and is encoded with 1 bit. The borders of a page are all white, so that all methods quickly learn to encode whites and the rate drops to nearly 0. As a page progresses, and the encoders encounter text and graphics, they slowly build context models and stabilize after about 200K pixels.

Note how APC outperforms ALUT in the results of Figure 4.5, specially at the beginning of coding. As coding procedes, their average codeword lengths become approximately the same. It is possible to see that, although the LUT is optimal when used on the same data that was used to build it, it is not optimal in online coding. The ALUT has a worse generalization performance than APC, and this is reflected in the results. An important point to be made is that results like these indicate that APC can successfully be used in place of the ALUT for the same context size M. This means that, if one wishes to reduce the average codeword length by means of increasing the context size M, one can use APC instead of ALUT when the context size becomes too large to use the LUT-based method.

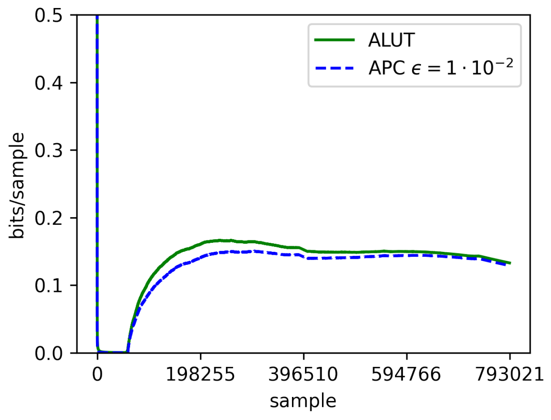
In order to define the learning rate for APC, we compressed the validation page of Dataset 6 with different values of learning rate ϵ. Figure 4.6 shows the evolution of the average codeword length as the page of nearly 787K samples is encoded, with ϵ equal to $10^{-1}, 10^{-2}, 10^{-3}$ and 10^{-4}. As can be seen from the figure, for $\epsilon = 10^{-4}$ the adaptation is too slow, which makes the average codeword length lag compared to the other values of ϵ. For $\epsilon = 10^{-1}$, on the other hand, the adaptation is too fast, and the method overshoots the target, making it hard to recover later. However, for $\epsilon = 10^{-2}$, the adaptation seems



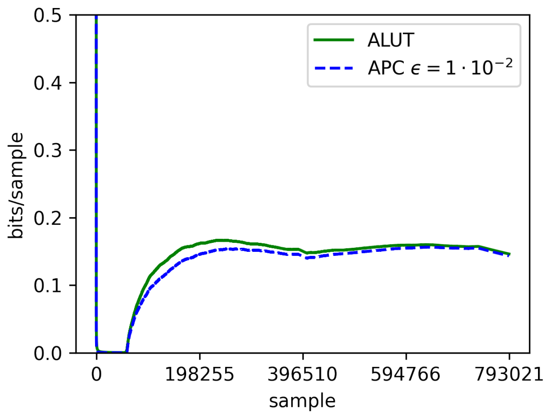
(a) Page 1



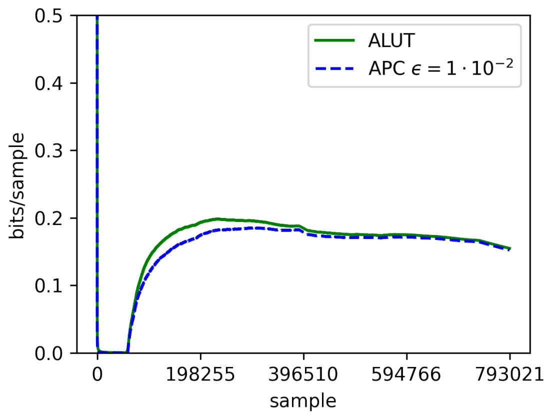
(b) Page 2



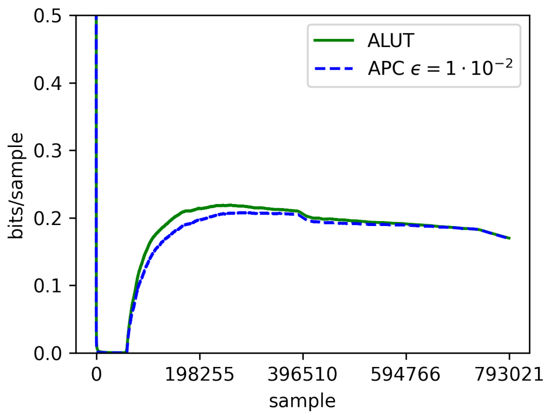
(c) Page 3



(d) Page 4

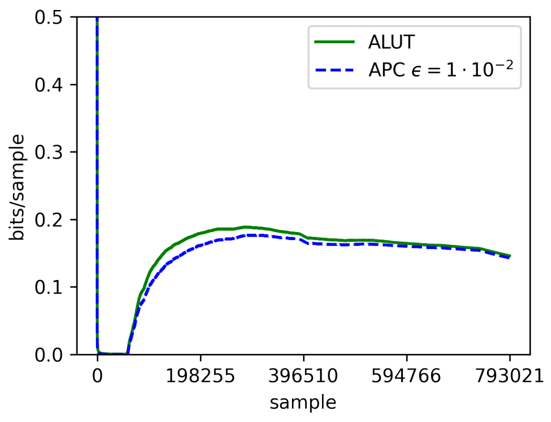


(e) Page 5

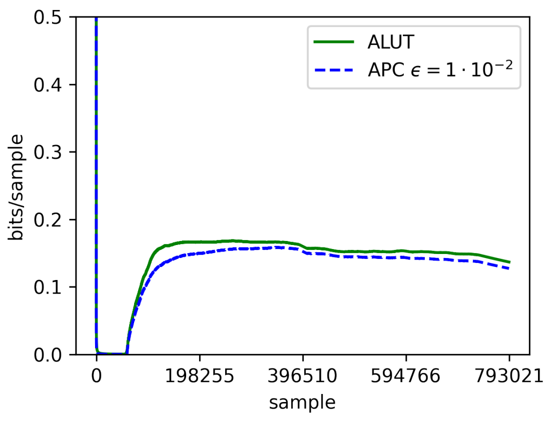


(f) Page 6

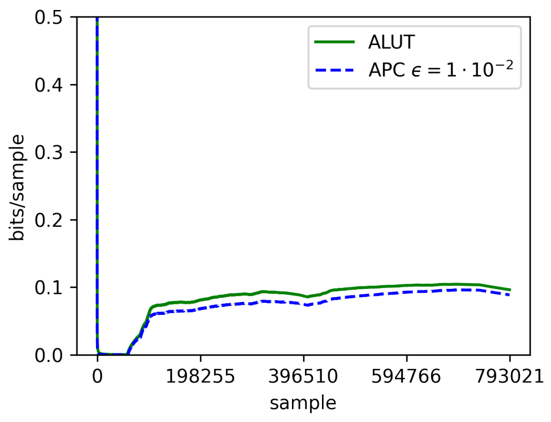
Figure 4.5: ((a) - (f)) Evolution of the bitrate in bits/symbol of APC and ALUT when coding the different pages from the article [65].



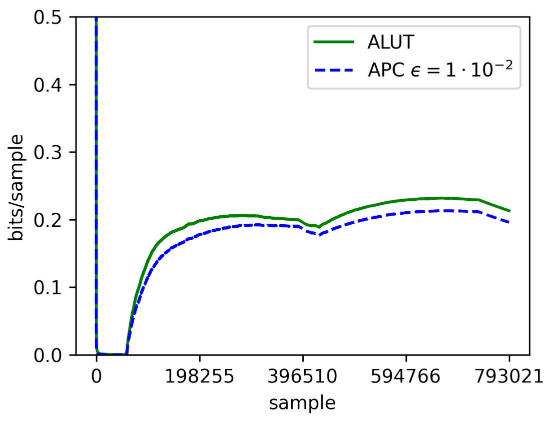
(g) Page 7



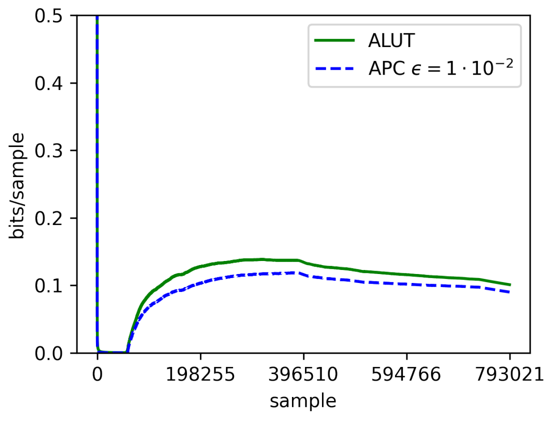
(h) Page 8



(i) Page 9



(j) Page 10



(k) Page 11

Figure 4.5: ((g) - (k)) Evolution of the bitrate in bits/symbol of APC and ALUT when coding the different pages from the article [65].

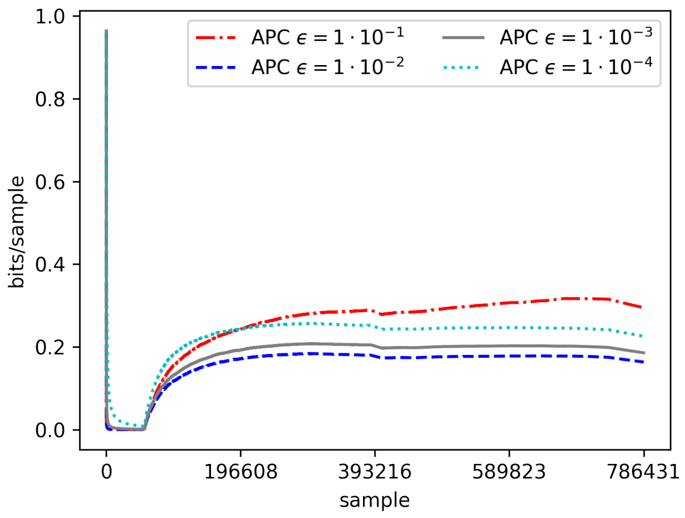


Figure 4.6: Cumulative rate (bits/symbol) when coding page 2 of the article [96] with APC, $M = 26$ and different values of learning rate.

just about right, reaching the optimal value of average codeword length compared to the other values of ϵ. We also tested these different learning rates for M equal to 4,10,26 and 67, with $\epsilon = 0.01$ leading to the best results for every M, therefore we fixed that value of ϵ in other runs of APC.

For completeness, in this section, we also compare APC with an online trained RNN, which we call ARNN. The ARNN method also requires tuning the learning rate. Therefore we also experimented compressing the validation page of Dataset 6 using ARNN with different values of learning rate. We found that the best results were obtained for $\epsilon = 0.01$ and updates after every 64 samples. We also tested ϵ equal to 0.05, 0.005 and 0.001, and updates every 1 and 96 samples. Therefore we fixed those values in subsequent runs of ARNN. Figure 4.7 shows how different learning rates affect the cumulative bitrate of ARNN with updates every 64 samples.

With the hyperparameter tuning process explained, we proceed with more comparisons among methods. We compared APC, ALUT and ARNN to encode the complete 5-page article from the test set of Dataset 6. The results are shown in Figure 4.8. The Figure shows the bitrate at different instants for APC and ALUT with $M = 26$. Note that the cumulative rate keeps decreasing as the methods learn more context patterns.

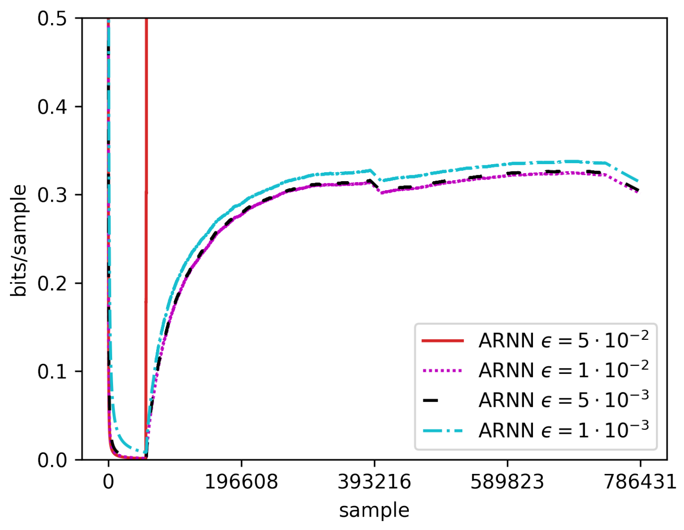


Figure 4.7: Cumulative rate (bits/symbol) when coding page 2 of the article [96] with ARNN, updates every 64 samples, and different values of learning rate.

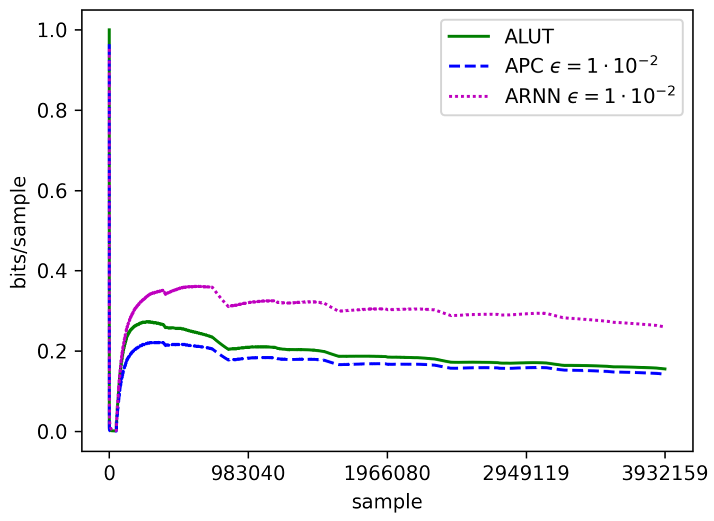


Figure 4.8: Cumulative rate in bits/symbol as APC and ALUT progress to encode a given 5-page paper in the test set of Dataset 6, with $M = 26$. We also included results using ARNN for comparison.

APC also outperforms ARNN. The reason why ARNN performs poorly even though its context size is unbounded is because the effect of previous samples on the RNN fades away with distance. LSTMs and GRUs can capture longer data dependencies, but it is hard to capture very long data dependencies. Samples closest in the path are given more importance, and the ARNN follows the same traversal order as the arithmetic coder, that is, row by row. This means that distant samples from the same row are given more importance than closer ones from a different row. Other orders of traversal would face similar problems. The MLP, on the other hand, is more flexible. Even though its context size is limited, we can design the context size so that it focuses on the samples that matter the most. One way to make the RNN also focus on the samples that matter would be through the use of attention [101].

Tables 4.2 and 4.3 show the results of APC, ALUT, JBIG and the offline methods of Section 4.6 on the 10 test pages from the Dataset 5, and the 100 test pages from the Dataset 4, for different context sizes M. As before, we only include the values of the LUT-based methods up to $M = 26$. For $M > 26$, they become too memory-consuming and impractical. Besides all the advantages of the online-trained methods, due to them not requiring previous training, another advantage, that we mentioned in Section 4.3, is that they approach the true probability distribution, instead of the training data distribution. Therefore, it is our expectation that the online-trained methods should surpass the offline-trained methods, in terms of compression performance. As can be seen from Tables 4.2 and 4.3, this still does not happen for all M for only the 10 test pages from the Dataset 5. However, when the number of test pages is increased to the 100 test pages from the Dataset 4, the online-trained methods are finally able to surpass the offline-trained methods for all values of M. Recall that the training sets of Datasets 5 and 4 are the same. This was the training set used to train the offline methods. The results for the offline methods (and JBIG) on the Table 4.3 are the same as the ones present in Figure 4.3 (b).

As mentioned in Section 4.3, we use a slightly modified version of Xavier initialization with APC. In order to demonstrate that our modifications truly improve the results, we made repeated experiments using the validation page of Dataset 6, ϵ equal to $10^{-1}, 10^{-2}, 10^{-3}$ and 10^{-4}, and M equal to 4, 10, 26 and 67. The result is that the initialization method described in Section 4.3 always yields slightly better results than Xavier initialization. Figure 4.9 shows an example of the evolution of the cumulative bit-rate for the two initializations. The example in Figure 4.9 is for $M = 26$ and $\epsilon = 0.01$.

Table 4.2: Coding rates (bits/symbol) attained for different context sizes M with the offline-trained PC and LUT methods, APC (with $\lambda = 0.01$), ALUT, and JBIG, on the 10 test pages from the Dataset 5.

	Offline				
M	PC	LUT	APC	ALUT	JBIG
0		0.442		0.441	
2	0.262	0.263	0.253	0.260	
4	0.236	0.236	0.228	0.233	
10	0.204	0.204	0.197	0.201	0.188
26	0.132	0.242	0.140	0.152	
67	0.106		0.127		
170	0.105		0.139		

Table 4.3: Coding rates (bits/symbol) attained for different context sizes M with the offline-trained PC and LUT methods, APC (with $\lambda = 0.01$), ALUT, and JBIG, on the 100 test pages from the Dataset 4.

	Offline				
M	PC	LUT	APC	ALUT	JBIG
0		0.469		0.469	
2	0.265	0.265	0.259	0.264	
4	0.235	0.235	0.227	0.235	
10	0.201	0.201	0.194	0.200	0.185
26	0.131	0.250	0.121	0.126	
67	0.107		0.098		
170	0.104		0.103		

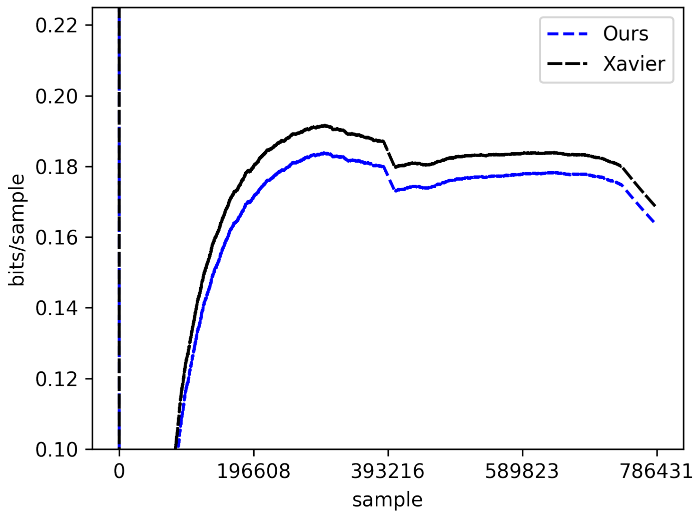


Figure 4.9: Evolution of cumulative rate in bits/symbol for Xavier initialization and our initialization, for $M = 26$ and $\epsilon = 0.01$.

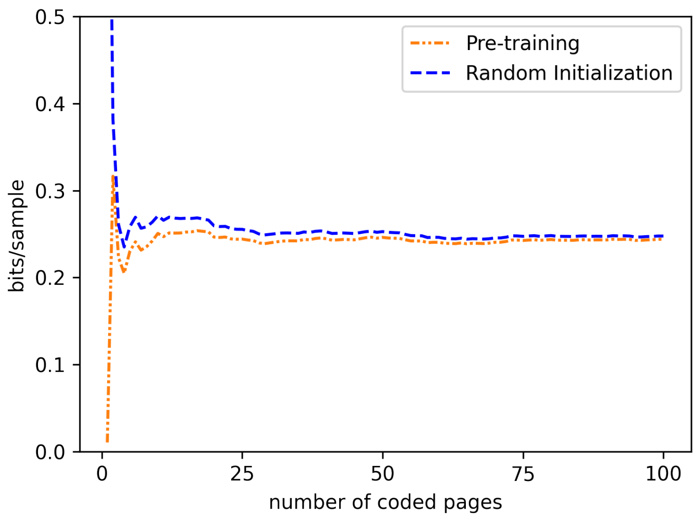


Figure 4.10: Comparison of APC with our custom initialization and with the network initialized with pre-trained weights, for $M = 10$ and $\epsilon = 0.0001$.

We also compared the initialization described in Section 4.3 with pre-training. We used the pre-trained weights and biases obtained in Section 4.6. The test data were the 100 pages from the dataset 4. However, in this case, we removed the margins of the pages. We used a learning rate close to the one used during pre-training, $\epsilon = 0.0001$. Larger learning rates sometimes caused abrupt changes in the network parameters, which were accompanied by large increases in the average codeword length. The result is shown in Figure 4.10. The curves are for $M = 10$.

As can be seen, the effects of initialization are long lasting. Even after 100 pages, the average codeword length of the pre-trained network is still better than the randomly initialized network. However, the average codeword lengths are close to each other. Although both networks are learning the same distribution, the networks are most likely at all times at very different positions in the graph of the loss function. This explains why the average codeword lengths of two differently initialized online-trained networks get close to each other, but may still be different, even after coding many pages.

Chapter 5

Greedy Lower Convex Hull

5.1 Motivation

Machine learning and deep learning have often been associated to a form of art, rather than to a form of science [102]–[104]. One of the reasons for this is that many of the design choices behind the neural network architectures, that feature in research papers, seem arbitrary. They give the impression that the authors have guessed the hyperparameters at random.

That is not too far from the truth, since random-search is one of the most popular algorithms used for hyperparameter selection [56]. However, random search, and other hyperparameter selection algorithms, for that matter, do not solely rely on the generation of neural network architectures. They can actually be seen as a two step process. In the first step, tentative network architectures are obtained. These architectures can be randomly generated, such as in random search, or they can be generated in a smarter way, for example, as in Bayesian optimization. In the second step, the best architecture, or the set of optimal architectures, is obtained, either by picking the one that optimizes the objective, or by using an algorithm to find the Pareto frontier (on the validation set). In other words, the first step may indeed be arbitrary, depending on the case, but the second step actually gives the best options among the ones that were generated.

Therefore, depending on how it is done, hyperparameter selection is not as obscure as it may seem. There is actually objective reasoning behind it. We intend to further develop this aspect of hyperparameter selection, by making the process more direct. Especially in the case of neural-based data compression.

In the case of data compression, contrary to other applications of deep learning, the more complex is not always the better. In many applications, one is only interested in a single objective. This objective may be, for example, image classification accuracy. In these cases, it is often the case that better results can be obtained by increasing the complexity of the network. For example, by increasing the number of layers, or the number of filters in the convolutional layers, of the network. In data compression, however, minimizing the complexity of the coder is almost as important as reducing the average codeword length (the rate). This is so because the coder might often be run in mobile phones or other devices of the sort.

Bearing in mind that increasing a hyperparameter often increases complexity but reduces the rate, we propose an algorithm that traces the LCH, or the Pareto frontier, at the same time that it proposes new architectures. Figure 5.1 illustrates the basic idea of the algorithm. Assume we want to operate at a complexity C_o or rate R_o. The idea behind the algorithm is to build a tree starting from a point with high rate and low complexity, for example the simplest architecture. At each step, we have one node, which we call the parent node, from which we obtain new architectures, which we call the child nodes. We separately increase the values of K hyperparameters from the parent node (in the example of the figure, $K = 3$). with this we expect the complexity to increase and the rate to decrease, but this is not guaranteed, specially for situations where the complexity measurement is noisy. This gives K new options to choose as the next parent node. Then, we pick the child node with the best performance and set it as the new parent node. We recur this procedure until reaching R_o or C_o and the resulting set of all parent nodes is our estimate of the LCH.

We call this algorithm greedy LCH (GLCH) because of the greedy choice of child node

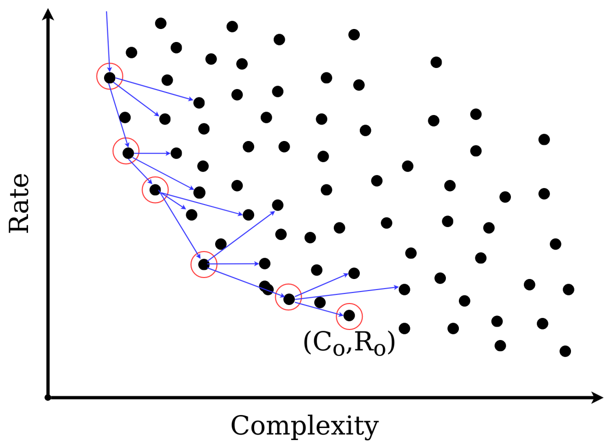


Figure 5.1: Illustration of our algorithm, aiming to track the lower convex hull of a cloud of rate-complexity operating points. Circled dots, or dots at one extreme of an arrow, are visited, or trained, networks. The other dots are all unvisited by the algorithm, that is, the corresponding neural networks did not have to be trained.

made at each step. This algorithm distinguishes itself from Bayesian optimization and other algorithms in that it introduces an ordering to the architectures. It begins with the simplest architecture, and subsequently increases its complexity, allowing one to stop the algorithm as soon as the desired maximum complexity, or a satisfactory rate is achieved.

5.2 The Architecture Graph

The GLCH algorithm assumes a predefined set of architectures connected in a particular way, which we represent by a graph. Consider a family of network architectures which differ only on the values of K hyperparameters. Let the hyperparameters of one architecture be represented by a vector $\mathbf{h} = (h_1, \ldots, h_K)$. Assume that hyperparameter h_k takes values in $v_1^{(k)}, \ldots, v_{T_k}^{(k)}$ or, without loss of generality, in $1, \ldots, T_k$. Then, the number of possible architectures is $\Upsilon = \prod_{k=1}^{K} T_k$, which exponentially grows with the number of hyperparameters K.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph with vertex set $\mathcal{V}$ and edge set $\mathcal{E}$, such that each element of the vertex set corresponds to one of the possible hyperparameter vectors $\mathbf{h}_{\min}, \ldots, \mathbf{h}_{\max}$. Assume that there is one directed edge from every node $\mathbf{h}$ to every other node $\mathbf{h}'$ such that $h'_k = h_k + 1$ for exactly one k, and $h'_k = h_k$ for the other k. Thus, the graph $\mathcal{G}$ corresponds to a K-dimensional, $T_1 \times \cdots \times T_K$ rectangular grid, with links from every node to its immediate next neighbor along each axis. Figure 5.2 (a) illustrates one such rectangular grid for $K = 3$, $T_1 = T_3 = 4$ and $T_2 = 3$.

The following can be said about the graph $\mathcal{G}$:

- It is a directed acyclic graph.
- The out-degree of each node is at most K.
- There is a unique "minimal" node with in-degree 0, namely $\mathbf{h}_{\min} = (1, 1, \ldots, 1)$, which we call the *root* of the graph.
- There is a unique "maximal" node with out-degree 0, namely $\mathbf{h}_{\max} = (T_1, \ldots, T_K)$.
- It is weakly connected.
- There exists a path from the root to any node.
- All paths from the root to a node have the same length, which we call the *depth* of the node.
- A node's depth is given by the Manhattan distance between the hyperparameter vectors of the root and the node.
- The longest path in the graph is from the minimal node to the maximal node.
- The length of the longest path (i.e., the *diameter*) of the graph is $\delta = \sum_{k=1}^{K}(T_k - 1)$, which is the depth of the maximal node.

5.3 The Basic GLCH Algorithm

In Algorithm 1 we show the basic GLCH algorithm. This algorithm essentially operates on a graph $\mathcal{G}$, with the properties described in Section 5.2, and iteratively builds a tree, which is a subgraph of $\mathcal{G}$, starting from the minimal node $\mathbf{h}_{\min}$. At every step, the set of nodes $\mathcal{V}$ is partitioned into three sets: the set of open nodes $\mathcal{O}$, the set of closed nodes $\mathcal{C}$, and the set of unvisited nodes. Initially, only the root node $\mathbf{h}_{\min}$ is in the open set. At every step, we select one node from the open set, move it to the closed set, and add the children of the selected node to the open set. The children of a node are the targets of

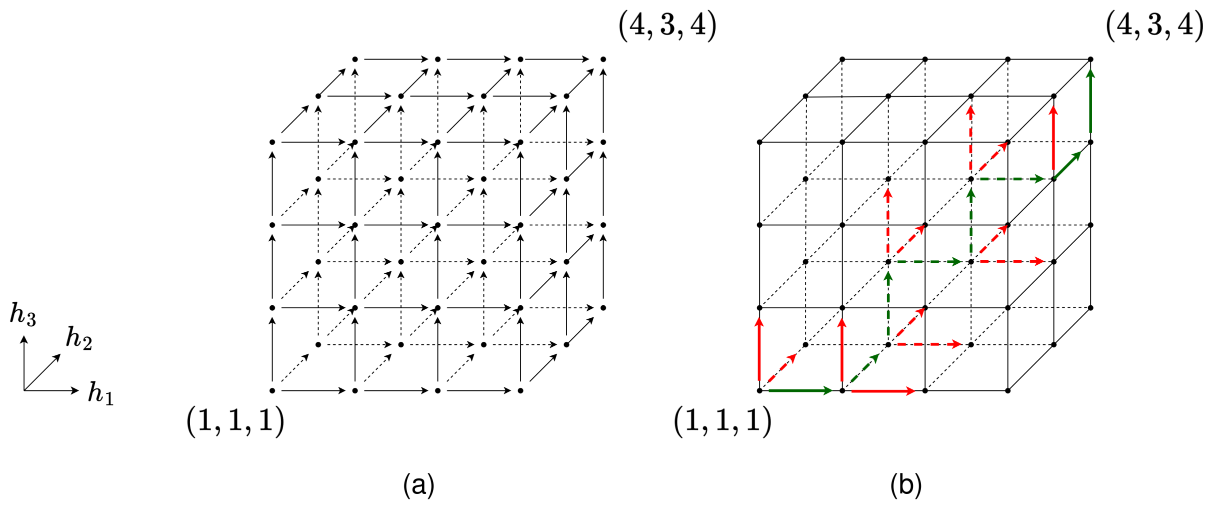


Figure 5.2: (a) Example of architecture graph for three hyperparameters, h_1, h_2, h_3, with number of possible values equal to 4,3 and 4, respectively. (b) Example of path (in green) from the minimum to the maximal node, and example of a tree (green and red) that a constrained GLCH algorithm can generate.

the directed edges which originate from it. We terminate when the maximal node $\mathbf{h}_{\max}$ is reached or an early termination condition is satisfied. Only the architectures in the open and closed sets need to be trained. The nodes that compose the tree are those who are present in the open and closed sets.

Algorithm 1 GLCH Algorithm

Input: the graph $\mathcal{G}$ of all possible hyperparameter vectors

```
1: Set the open and closed sets to the empty set: $\mathcal{O} \leftarrow \emptyset$, $\mathcal{C} \leftarrow \emptyset$
2: Train/evaluate the minimal node $\mathbf{h}_{\min}$, make it the parent node, and add it to the open set:
   $\mathbf{h} \leftarrow \mathbf{h}_{\min}$, $\mathcal{O} \leftarrow \mathcal{O} \cup \{\mathbf{h}\}$
3: repeat
4:     Train/evaluate all children (i.e., out-neighbors) of the parent node $\mathbf{h}$ and add to $\mathcal{O}$
5:     Move the parent node to the closed set: $\mathcal{C} \leftarrow \mathcal{C} \cup \{\mathbf{h}\}$, $\mathcal{O} \leftarrow \mathcal{O} \setminus \{\mathbf{h}\}$
6:     Select a new parent node from the open set: $\mathbf{h} \leftarrow select(\mathcal{O}, \mathcal{C}, \mathbf{h})$
7: until $\mathbf{h}$ is the maximal node $\mathbf{h}_{\max}$, or satisfies an early termination condition
```

Output: the set $\mathcal{O} \cup \mathcal{C}$ of visited nodes, and the history of all past parent nodes $\mathbf{h}$

At the heart of Algorithm 1 is the *select* function in line 6, used to choose the node that is to be moved to the closed set and whose children are to be added to the open set. Different select functions yield different variants of the algorithm. In particular, we propose two broad classes of GLCH algorithms.

In the first class, the *select* function is *constrained* to choose a node $\mathbf{h}$ from $\mathcal{O}$ only if it is among the deepest open nodes $\mathcal{O}' \subset \mathcal{O}$. Then the GLCH algorithm keeps extending the longest path and terminates having visited not more than δK nodes, that is, trained not more than δK network architectures, where δ is the diameter of the graph. Since δK grows quadratically in K, whereas the total number of possible architectures, Υ, grows exponentially in K, there is considerable savings compared to training all the networks. Figure 5.2 (b) illustrates one possible tree that a constrained GLCH algorithm can generate.

In the second class, the *select* function is *unconstrained*, that is, it is allowed to choose any node $\mathbf{h}$ from $\mathcal{O}$. This allows the algorithm to extend other leaves of the tree besides the deepest leaves. This often results in a better LCH approximation, with the cost of more trained networks. In fact, in the worst case scenario, the number of trained networks may even reach Υ, although in practice the algorithm usually terminates much earlier.

5.4 Select Functions

In order to choose one among several alternative network architectures, it is necessary to evaluate the performances of the network architectures on the multiple objectives. The multiple objectives are loss and complexity. We use the term loss instead of rate because the GLCH algorithm may be used with any loss other than rate. Rate is only the loss function in lossless compression.

One immediate approach, for choosing the best architectures, is to rely on other existing algorithms that can find the LCH of a set of known points. Note that the GLCH algorithm aims at finding an *approximate* LCH, of a set of initially *unknown* points, without having to reveal all of them. Algorithms to find the LCH of a set of known points, such as the one described in Section 3.6, assume that the input points are all there is, and ignore the existence of any others, differently from the GLCH algorithm

In Algorithms 2 and 3 we propose two alternative *select* functions, one constrained and another unconstrained. They both depend on other algorithms in order to find the LCH of a subset of points. Let $L_{\mathbf{h}}$ and $C_{\mathbf{h}}$ be the loss and complexity of the neural network associated with the node with hyperparameter vector $\mathbf{h}$. In both *select* functions, we choose the node that has the least complexity $C_{\mathbf{h}}$ among the nodes that are in $\mathcal{O}$ or $\mathcal{O}'$ and that are in the LCH (calculated following the procedure described in Section 3.6) of $\mathcal{O} \cup \mathcal{C}$. The set $\mathcal{O} \cup \mathcal{C}$ consists of all visited networks so far. We pick the node with least complexity, ignoring the nodes with higher complexity but lower loss, because, in subsequent iterations, the algorithm is expected to explore points with increasingly higher complexity and lower loss.

In the event that there are no nodes to choose from on the LCH of $\mathcal{O} \cup \mathcal{C}$, then the functions fall back to choosing among nodes in $\mathcal{O}$ or $\mathcal{O}'$ with performance on the LCH of $\mathcal{O}$ or $\mathcal{O}'$. In this case, the point with most complexity among the available options is selected. Note that none of the available options are on the LCH of all trained networks so far. However, if no node is selected, the algorithm cannot continue. In this case, the point with least complexity is generally not an option because a point with high rate and low complexity, close to the initial point $\mathbf{h}_{\min}$, may be selected, bringing the algorithm back to where it started.

Algorithm 2 Unconstrained Select Function 1

```
Input: $\mathcal{O}, \mathcal{C}$
1: Find node $\mathbf{h}^* \in \mathcal{O}$ with least complexity $C_{\mathbf{h}^*}$ s.t. $(C_{\mathbf{h}^*}, L_{\mathbf{h}^*})$ is on LCH of $\{(C_{\mathbf{h}}, L_{\mathbf{h}}) : \mathbf{h} \in \mathcal{O} \cup \mathcal{C}\}$ if exists, else with most complexity on LCH of $\{(C_{\mathbf{h}}, L_{\mathbf{h}}) : \mathbf{h} \in \mathcal{O}\}$
Output: $\mathbf{h}^*$
```

Algorithm 3 Constrained Select Function 1

```
Input: $\mathcal{O}, \mathcal{C}$
1: Let $\mathcal{O}' \subset \mathcal{O}$ be the subset of $\mathcal{O}$ whose distance from the root (i.e., depth) is largest
2: Find node $\mathbf{h}^* \in \mathcal{O}'$ with least complexity $C_{\mathbf{h}^*}$ s.t. $(C_{\mathbf{h}^*}, L_{\mathbf{h}^*})$ is on LCH of $\{(C_{\mathbf{h}}, L_{\mathbf{h}}) : \mathbf{h} \in \mathcal{O} \cup \mathcal{C}\}$ if exists, else with most complexity on LCH of $\{(C_{\mathbf{h}}, L_{\mathbf{h}}) : \mathbf{h} \in \mathcal{O}'\}$
Output: $\mathbf{h}^*$
```

We experimentally observed that, in the constrained case, when there is more than one node, or no node, on the LCH of $\mathcal{O} \cup \mathcal{C}$, selecting the point closest to the origin, generally gives better results. Even on par with the unconstrained case. We measure the distance to the origin with $C + \mu L$ where μ controls the importance given to the distance in L compared to the distance in C. We use $\mu = (1/\mu')(|C_{\mathbf{h}_{\max}} - C_{\mathbf{h}_{\min}}|/|L_{\mathbf{h}_{\max}} - L_{\mathbf{h}_{\min}}|)$, where $|C_{\mathbf{h}_{\max}} - C_{\mathbf{h}_{\min}}|$ and $|L_{\mathbf{h}_{\max}} - L_{\mathbf{h}_{\min}}|$ are the distances between the minimal and maximal nodes in the C and L axes, and μ' is a constant. That is, we define μ based on the ranges of values in the C and L axes. In our experiments, we found that giving slightly more importance to C yields better results. Therefore, we use $\mu' = 6$.

Algorithm 4 Constrained Select Function 2

```
Input: $\mathcal{O}, \mathcal{C}$
1: Let $\mathcal{O}' \subset \mathcal{O}$ be the subset of $\mathcal{O}$ whose distance from the root (i.e., depth) is largest
2: Find node $\mathbf{h}^* \in \mathcal{O}'$ with least $C_{\mathbf{h}^*} + \mu L_{\mathbf{h}^*}$ s.t. $(C_{\mathbf{h}^*}, L_{\mathbf{h}^*})$ is on LCH of $\{(C_{\mathbf{h}}, L_{\mathbf{h}}) : \mathbf{h} \in \mathcal{O} \cup \mathcal{C}\}$ if exists, else with least $C_{\mathbf{h}^*} + \mu L_{\mathbf{h}^*}$ s.t. $(C_{\mathbf{h}^*}, L_{\mathbf{h}^*})$ is on LCH of $\{(C_{\mathbf{h}}, L_{\mathbf{h}}) : \mathbf{h} \in \mathcal{O}'\}$
Output: $\mathbf{h}^*$
```

5.5 Simplifying the Select Functions

The dependence on a third-party routine, to compute the LCH of a subset of nodes, is not really necessary. We can specify a set of rules, to use in every possible configuration of the child nodes relative to the parent node, and we can create select functions based on these rules.

The LCH of an arbitrary set of known points in 2D can be obtained by starting from the leftmost point and always picking the point that gives the smallest polar angle, with respect to the previously chosen convex hull side, as depicted in Figure 3.4. This algorithm is described in Section 3.6, and works to obtain the LCH of a *known* set of points. When the set of points is initially unknown, but is gradually revealed, there is no guarantee that the remaining points are not to the left and below the lastly selected point. The newly revealed points could be everywhere relative to the lastly selected point.

In the case of the GLCH algorithm, when we increase one hyperparameter of a node $\mathbf{h}$, we expect the complexity to increase and the loss to decrease. However, this is not always the case. Sometimes, the complexity might slightly decrease or the loss might increase. The loss can increase, for example, because of overfitting. The complexity measure, in its turn, can decrease in the case of a noisy complexity measure, such as energy consumption. This creates unexpected situations, that traditional algorithms are not prepared to deal with. Furthermore, differently from traditional algorithms, we do not want to stop when all points are above the lastly selected point. Since the points are gradually revealed, and since the loss and complexity can oscillate, it is perfectly possible that future points fall below the lastly selected point.

Figure 5.3 shows three relevant situations. By dealing with these situations, we can create standalone select functions, that do not depend on third-party routines to compute the LCH of known points. Firstly, if no option has lower loss than $\mathbf{h}$, we do not update $\mathbf{h}$ and we find other node from where to branch off newer nodes [1]. For this purpose, we choose the node with the lowest loss in $\mathcal{O}$ or $\mathcal{O}'$, and if there is more than one node that satisfies this requirement, we choose the one with largest complexity among them. This is a similar rule to the fall-back rule used in Algorithms 2 and 3. Secondly, if one or more nodes are better than $\mathbf{h}$ in both loss and complexity, we pick the one with lowest complexity to replace $\mathbf{h}$, and if there is more than one such node, we pick the one with lowest loss among them. This is natural, since this would be the first point on the

[1]The node $\mathbf{h}$ may be called the "real parent", while the node used to branch off newer nodes may be called the "surrogate parent"

LCH of the points in $\mathcal{O}$ or $\mathcal{O}'$ that is below $\mathbf{h}$. We ignore points above $\mathbf{h}$ to avoid the algorithm from going backwards. Thirdly, if all nodes $\mathbf{h}'$ in $\mathcal{O}$ or $\mathcal{O}'$, that have lower loss than $\mathbf{h}$, also have higher complexity than $\mathbf{h}$, then a node $\mathbf{h}'$ with best performance is one that maximizes $-\Delta L_{\mathbf{h}'}/\Delta C_{\mathbf{h}'}$, where $\Delta L_{\mathbf{h}'} = L_{\mathbf{h}'} - L_{\mathbf{h}}$ and $\Delta C_{\mathbf{h}'} = C_{\mathbf{h}'} - C_{\mathbf{h}}$. If there is more than one node that satisfies such condition, we pick the node with largest $\sqrt{(\Delta L_{\mathbf{h}'})^2 + (\Delta C_{\mathbf{h}'})^2}$ among them. The reason for this is because if we did not pick this point, the unconstrained algorithm could pick it in a subsequent run, but the constrained algorithm would not be able to do so. This third case is essentially the standard rule used by the algorithm described in Section 3.6.

In Algorithms 5 and 6 we show how Algorithms 2 and 3 can be adapted to use these rules. Later we show that the results using Algorithms 5 and 6 are nearly identical to the results using Algorithms 2 and 3.

Algorithm 5 Unconstrained Select Function 2

```
Input: $\mathcal{O}, \mathcal{C}, \mathbf{h}$
1: if all points in $\mathcal{O}$ have larger loss than $\mathbf{h}$ then make $\mathbf{h}^* \leftarrow \mathbf{h}$, find the node in $\mathcal{O}$ with lowest loss then largest complexity, move it from $\mathcal{O}$ to $\mathcal{C}$ and add its children to $\mathcal{O}$
2: else if there is any point in $\mathcal{O}$ with lower or equal loss and lower or equal complexity than $\mathbf{h}$ then find the node $\mathbf{h}^*$ among them with lowest complexity then lowest loss
3: else find the node $\mathbf{h}^* \in \mathcal{O}$, with lower or equal loss than $\mathbf{h}$, such that $-\Delta L_{\mathbf{h}^*}/\Delta C_{\mathbf{h}^*}$ is maximum
4: end if
Output: $\mathbf{h}^*$
```

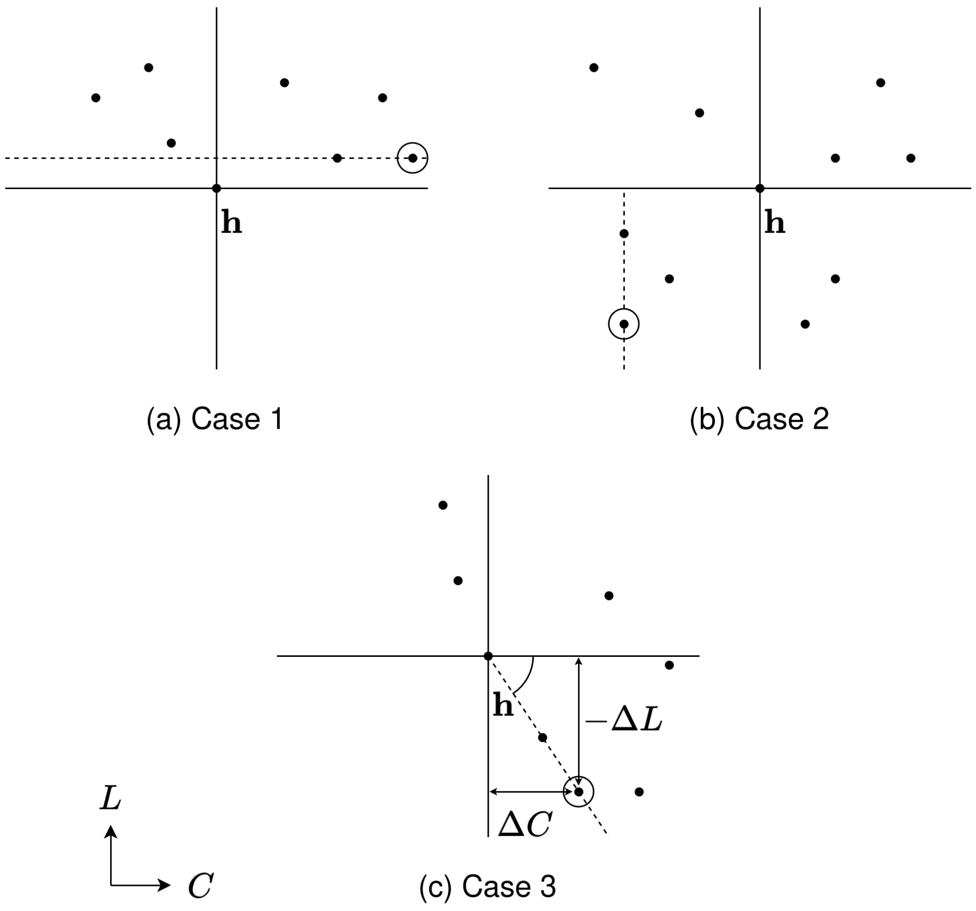


Figure 5.3: The three different configurations of relevance of child nodes relative to the parent node. In each case, the parent node is the node labeled with an **h**. All other nodes are candidates for next parent node. The circled node is the one selected to be the next parent node (cases 2 and 3), or to just branch off newer nodes (case 1).

Algorithm 6 Constrained Select Function 3

```
Input: $\mathcal{O}, \mathcal{C}, \mathbf{h}$
1: Let $\mathcal{O}' \subset \mathcal{O}$ be the subset of $\mathcal{O}$ whose distance from the root (i.e., depth) is largest
2: if all points in $\mathcal{O}'$ have larger loss than $\mathbf{h}$ then make $\mathbf{h}^* \leftarrow \mathbf{h}$, find the node in $\mathcal{O}'$ with lowest loss then largest complexity, move it from $\mathcal{O}$ to $\mathcal{C}$ and add its children to $\mathcal{O}$
3: else if there is any point in $\mathcal{O}'$ with lower or equal loss and lower or equal complexity than $\mathbf{h}$ then find the node $\mathbf{h}^*$ among them with lowest complexity then lowest loss
4: else find the node $\mathbf{h}^* \in \mathcal{O}'$, with lower or equal loss than $\mathbf{h}$, such that $-\Delta L_{\mathbf{h}^*}/\Delta C_{\mathbf{h}^*}$ is maximum
5: end if
Output: $\mathbf{h}^*$
```

5.6 Reasoning behind the Select Functions

The different select functions presented reflect the thought process we went through when designing the GLCH algorithm. We first envisioned the Algorithms 2 and 3, but they rely on subroutines to compute the LCH of a known set of points. Therefore, we proposed Algorithms 5 and 6 to replace them.

Algorithm 4 is an alternative version of the Algorithm 3, which gave better results in our set of examples. However, since this algorithm was based on a heuristic, and has a parameter which was set based on our particular set of experiments, this algorithm is likely overfitted to our examples.

In practice, one would only need Algorithms 5 or 6, depending if one desires the constrained or the unconstrained version of the algorithm. The constrained version of the algorithm always terminates after evaluating a known number of networks. If one wishes a better LCH approximation at the cost of a higher (and unknown) number of trained networks, one may use the unconstrained version of the algorithm.

Algorithm 7 Unconstrained Select Function 3

```
Input: $\mathcal{O}$,$\mathcal{C}$
1: Find node $\mathbf{h}^* \in \mathcal{O}$ such that $R_{\mathbf{h}^*} + \lambda_0 D_{\mathbf{h}^*} + \gamma_0 C_{\mathbf{h}^*}$ is minimum
Output: $\mathbf{h}^*$
```

5.7 Rate-Distortion-Complexity Optimization with the GLCH Algorithm

Neural networks for lossy compression are usually trained to minimize a loss $L = R + \lambda D$, where R is the rate using the network, and D is the distortion. In cross-validation, one would train several networks to minimize this loss for a fixed λ, and would later select the one with minimum L on the validation set. Assuming that R is the horizontal axis, and that D is the vertical axis, this corresponds to finding the first point that touches a line with inclination $-1/\lambda$, or normal vector $[1, \lambda]$, emanating from the origin. The LCH is the set of points that minimize L for all different λ in the interval $(0, \infty)$.

We would also want to take complexity under consideration. In this case, the metric is $R + \lambda D + \gamma C$, and the LCH is the set of points that minimize it for every λ and γ in the interval $(0, \infty)$. Each pair of λ and γ defines a set of parallel planes in 3D, the planes with normal vector $[1, \lambda, \gamma]$, and the minimum for a specific λ and γ is the first point that touches one such plane emanating from the origin.

The GLCH algorithm can be used in all the following three cases: (i) to approximate a specific optimal point, for a specific λ and a specific γ; (ii) to approximate sets of optimal points for *slices* of data; and (iii) to approximate the LCH of the cloud of points. We explore each of these cases in the following:

(i) In the simplest case, one is interested in only the minimum for one λ and one γ. In this case, one can use the GLCH algorithm with the select function shown in Algorithm 7. We would like to find the tracked point as quickly as possible, and then interrupt execution. Constraining the algorithm would not be advantageous, because this could prevent it from exploring leaves, that could more quickly lead closer to the optimal point. For early termination, a condition, such as a maximum number of iterations, can be used.

(ii) By *slice* of data, we mean the points trained for a specific λ. In this case what matters is to minimize $L = R + \lambda D$, irrespective of the values of R and D. Therefore, we can run the GLCH algorithm on the L-C-plane, using any of the select functions in Algs. 2-6.

(iii) When we consider only C and L, and ignore the particular values of R and D, we project the points on the plane formed by the vectors $[0, 0, 1]$ and $[1, \lambda, 0]$, as depicted in Figure 5.4. The inner product of a point $[R, D, C]$ with the vector $[0, 0, 1]$ gives the complexity, $[0, 0, 1] \cdot [R, D, C] = C$, and the inner product of a point $[R, D, C]$ with the vector $[1, \lambda, 0]$ gives the loss, $[1, \lambda, 0] \cdot [R, D, C] = R + \lambda D = L$. Interestingly, points on the LCH of the projected points, are also on the LCH of the original cloud of points. This is so because, consider the minimum for a fixed λ and γ, that is $\arg\min_{\mathbf{h}} R_{\mathbf{h}} + \lambda D_{\mathbf{h}} + \gamma C_{\mathbf{h}}$. When λ is fixed, the specific values of $R_{\mathbf{h}}$ and $D_{\mathbf{h}}$ do not matter. What matters is the value of $L_{\mathbf{h}} = R_{\mathbf{h}} + \lambda D_{\mathbf{h}}$. Therefore, this same minimum can be found by $\arg\min_{\mathbf{h}} L_{\mathbf{h}} + \gamma C_{\mathbf{h}}$. Therefore, when we minimize $L + \gamma C$ in the projected space for a specific γ (and λ), we are also minimizing $R + \lambda D + \gamma C$ in the original space for a particular λ and γ. In this process, we ignore the networks not trained for this λ, because it is unlikely that they have a lower L for this given λ. What all this means is that the LCH of the cloud of points can be approximated by running the GLCH algorithm for the *slices* corresponding to several λ.

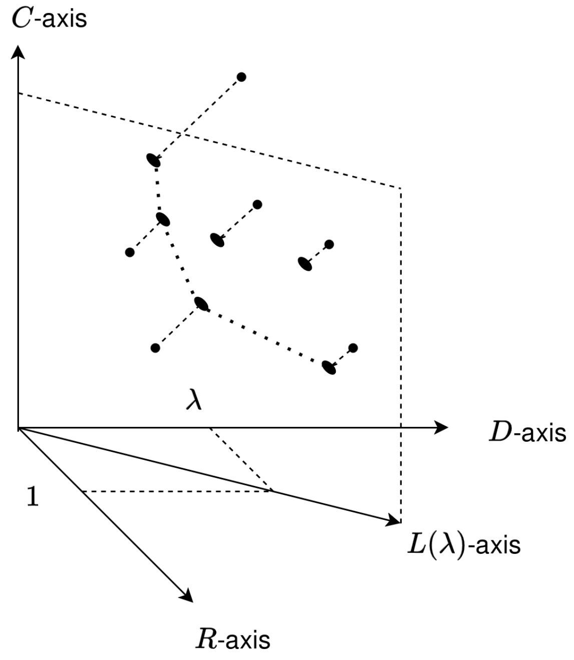


Figure 5.4: Illustration of what happens when we ignore the particular values of R and D, and compute the LCH of only C and $L = R + \lambda D$, in a rate-distortion-complexity optimization problem. We find the LCH of the projected points on the C-$L(\lambda)$ plane, where $L(\lambda)$ is the axis with the same direction as the vector $[1, \lambda, 0]$.

5.8 Rate-Complexity Optimization in Lossless Image Compression

The GLCH algorithm can be used to select neural network architectures, from a set of possible architectures, without having to evaluate the performances of all network architectures. We particularly developed the GLCH algorithm in order to take complexity into account when selecting neural networks, instead of only accounting for the performance metric it was trained to minimize. The optimal network architectures are the ones that lie on the LCH of the cloud of points. The GLCH algorithm visits neural network architectures, and therefore builds the LCH, in increasing order of complexity. In this way, the algorithm can be stopped once the highest level of acceptable complexity, or the superior level of desired network loss, is achieved.

In this section, we use the GLCH algorithm to select networks for lossless compression of binary images. We assume that the images are to be compressed using a method of forward coding with context modeling for arithmetic coding, similar to the PC method we described in Section 4.2. The context is composed of the $M = 32$ closest causal pixels, which gives a context similar to the ones shown in Figures 2.10 and 4.1 for other values of M.

We assume that the set of possible architectures consists of MLPs with two hidden layers, having either 10, 20, 40, 80, 160, 320, or 640 hidden units, each. Thus, in this case, the number of hyperparameters is $K = 2$, and the number of possible values per hyperparameter is $T_k = 7$ for $k = 1, 2$. The total number of architectures is $\Upsilon = 49$.

For the purpose of evaluating the GLCH algorithm, we train all possible networks. The networks are trained on an NVidia GTX 1080Ti GPU to minimize the binary cross entropy loss on the training set of Dataset 5 (described in Section 4.5). We use stochastic gradient descent, 100 epochs, a learning rate of 0.0001, and a batch size of 1024 samples.

We assume one is interested in two forms of forward coding. One is the more traditional forward coding scheme, in which the networks are trained on data from one set and used to compress data from another set. We use the validation set of Dataset 5 to select networks for this form of coding. Therefore, when selecting the networks for this form of coding, the performances reported are measured while encoding the validation set. The complexity is measured in either Joules/pixel or MAC/pixel.

The other form of forward coding, that we assume one is interested in, is based on algorithmic information theory [57]. The basic idea of compression based on algorithmic information theory is to describe a program which can be used to regenerate the data. One approach views the construction of such program as data modeling [57]. For example, the data could be exactly represented by some model and the residual between the model and the data. Optimality is achieved when the sum of the size of the model and the size of the residual is minimized. This is the minimum description length [57]. In one possible adaptation of this form of coding, the network takes the place of the model, and the compressed data takes the place of the residual, and both are included in the compressed file. In this case, the network is trained and evaluated on the data to be compressed.

It does not have to generalize to other data. There is no need for three separate sets (training, validation and test sets). Only one set is necessary. We assume that this set is the training set of Dataset 5. When we select networks for this form of coding, we measure complexity in number of encoded model bits.

Note that, in this case, all that matters is the minimum description length, or $R+\lambda C$, with $\lambda = 1$, where C and R are the model bits and the data bits divided by the number of samples. However, since the rate-complexity points are initially unknown, finding the LCH is still beneficial. The minimum description length operating point lies on the LCH, and can be searched using the GLCH algorithm. The algorithm can be stopped once $\lambda = 1$ is reached.

The networks are initially trained using 32-bit floating point arithmetic. Therefore, the size of the network in bits is, initially, its number of parameters times 32 bits. To create more network alternatives for the second form of coding, we also quantize the network parameters to 8 and 16 bits. This corresponds to including a third hyperparameter h_3, with $T_3 = 3$ possible values (8, 16 and 32), increasing the total number of network alternatives to $\Upsilon = T_1 T_2 T_3 = 147$.

We do not include points for different quantization levels when selecting networks for the first form of coding (traditional forward coding), because we assume that the number of Joules/pixel and the number of MAC/pixel do not change with quantization. Since the network is trained using 32 bits, then quantized, we also assume that the bitrate does not improve, but actually gets worse, with quantization. Therefore, in these cases, we do not expect the quantized networks to be on the LCH.

In order to summarize, we measure complexity in three different ways: (1) MAC per pixel, (2) Joules per pixel, and (3) model bits. It is important to describe how we obtain the values for each of these three metrics. We explain this in the following three sections.

5.8.1 Multiply-accumulate operations per pixel

The values of MAC/pixel and the number of parameters of an MLP are essentially the same. First of all, the network is run once per pixel, therefore the values of MAC/pixel and MAC are the same. Second of all, at each neuron, the input values are multiplied by the weights then added (together with the bias), which amounts to a number of MAC equal to the number of weights. If we count the number of weights in all neurons, the total number of weights is an estimate of the number of MAC. The number of parameters of the MLP also includes the biases, and the activation functions also amount to some operations, but these numbers are generally negligible compared to the number of weights. Therefore, we can generally consider the values of MAC/pixel and network parameters as almost equivalent.

5.8.2 Joules per pixel

We estimate Joules by measuring the power consumption in watts every second, using the Nvidia-smi Toolkit®, then adding up all values. The samples are 1 second apart. Therefore, it approximately corresponds to the integral of the power consumption during the period of the evaluation. Since the energy consumption taken this way is very noisy, we repeat this process 200 times. We average the results to obtain the final energy consumption estimate, and divide by the total number of pixels in the validation set to obtain the value of Joules per pixel.

5.8.3 Encoded model bits

The size of the network in bits is its number of parameters times the number of quantization bits. As we said, we consider network architectures for numbers of quantization bits h_3 equal to 8, 16 and 32. We determine the quantization step Δ from the range of values of all parameters in the network, that is $\Delta = (\theta_{\max} - \theta_{\min})/2^{h_3}$, where $\theta_{\max}$ is the maximum and $\theta_{\min}$ is the minimum of all network parameters. Then we use midtread uniform quantization after subtracting the parameter value by $\theta_{\min} + \Delta/2$ or, in other words,

$$\theta_q = \text{round}\left(\frac{\theta - \theta_{\min} - \Delta/2}{\Delta}\right), \tag{5.1}$$

where θ_q represents a quantized parameter. We make sure that the extremes $\theta_{\min}$ and $\theta_{\max}$ are rounded to 0 and $2^{h_3} - 1$, respectively. We do not retrain the networks after quantization. For $h_3 = 32$, nothing needs to be done, since the networks are already trained using 32-bit floating point numbers.

It is important to note that we are assuming a 32-bit platform and that, regardless of the data width (if it is either 8,16 or 32) we are assuming that it will be processed with a 32-bit MAC. This is also why we assume that the number of Joules/pixel and the number of MAC/pixel do not change with quantization.

5.8.4 Results

Figure 5.5 shows the final states of the GLCH algorithm with the select functions 5 and 6 for complexity measured in MAC/pixel, μJoules per pixel, and model bits. The other select functions find trees only slightly different from the ones shown in Figure 5.5 (see Annex II for the trees found by the other select functions). Table 5.1 summarizes the number of visited networks for all variants of the GLCH algorithm. The select function in Algorithm 4 was the one which, in general, visited the least number of networks.

If we consider that the networks with same h_1 and h_2, but different h_3, that is, with same numbers of hidden units, but different quantization bits, share the same training, then the numbers of trained networks, when complexity is measured in model bits, are reduced to the numbers shown in Table 5.2. Note that, in this work, we did not retrain the neural networks for different numbers of quantization bits. However, in other works, the networks could be separately trained using 8-, 16- and 32-bit-floating-point arithmetic.

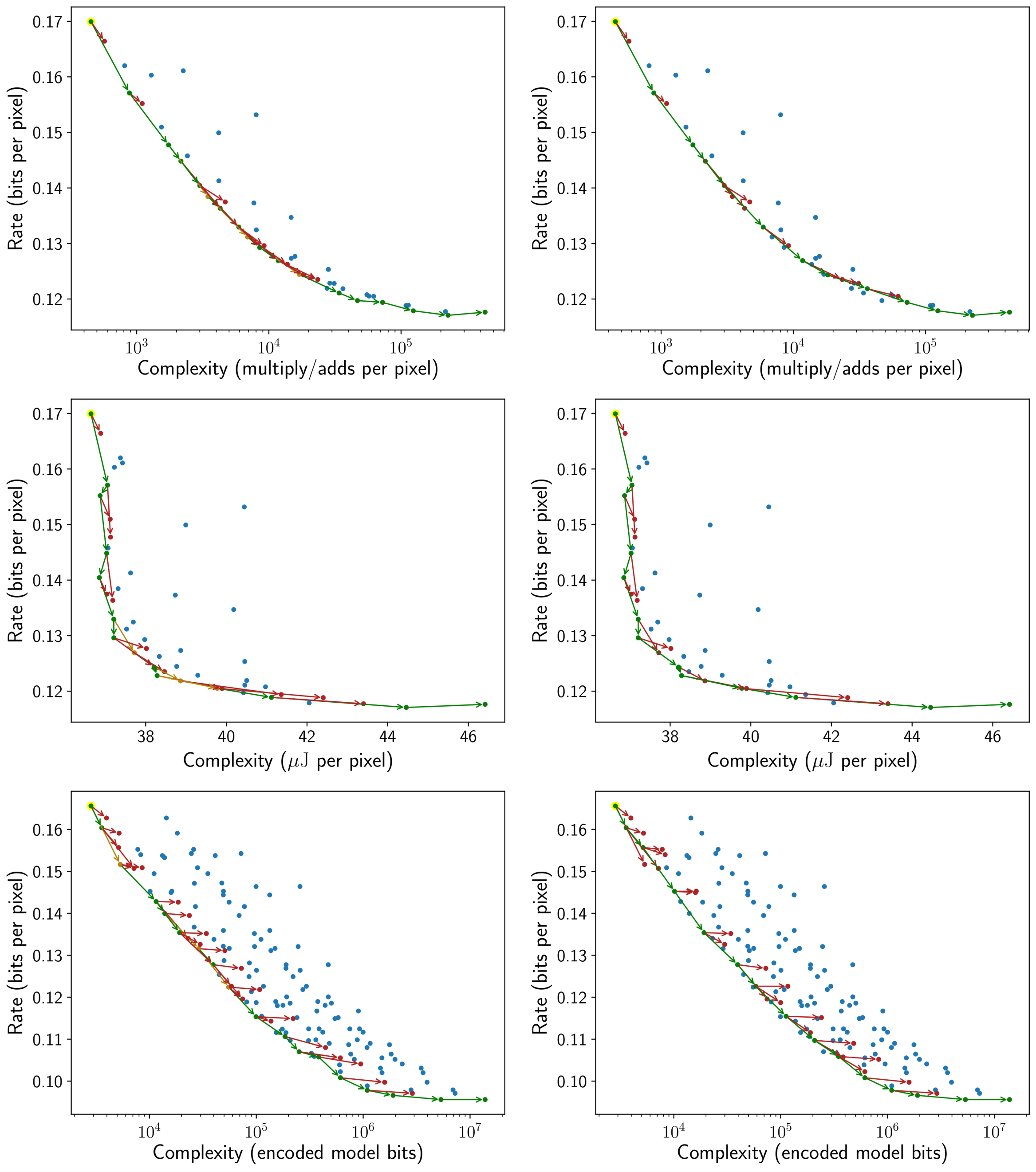


Figure 5.5: Clouds of rate-complexity points and final trees for the GLCH algorithm with the select functions 5 (left column) and 6 (right column), and with complexity measured in multiply/add operations per pixel (top row), μJoules per pixel (middle row), and encoded model bits (bottom row). Dots are rate-complexity performance of hyperparameters/nodes. At termination, blue nodes are never visited, red nodes have been visited (moved to the open set $\mathcal{O}$) but never selected, green and yellow nodes have been visited and selected (moved to the closed set $\mathcal{C}$). Arrows show parent-child relationships from green or yellow parents to red, green, or yellow children, and take the child color. A green arrow and child indicate that the child is selected in the step immediately following its parent's selection, while yellow indicates a gap between the parent's and child's selection.

Table 5.1: Numbers of visited networks for all variants of the GLCH algorithm and for complexity measured in: (a) MAC/pixel; (b) μJ/pixel and (c) model bits.

	complexity measure		
variant	(a)	(b)	(c)
Alg. 2	25	26	40
Alg. 3	22	24	37
Alg. 4	21	24	33
Alg. 5	25	26	40
Alg. 6	22	24	37

Table 5.2: Actual numbers of trained networks for all variants of the GLCH algorithm, complexity measured in MAC/pixel (a), μJ/pixel (b) and model bits (c), if we consider that the networks with same numbers of hidden units (h_1, h_2), but different numbers of quantization bits (h_3), share the same training.

	complexity measure		
variant	(a)	(b)	(c)
Alg. 2	25	26	23
Alg. 3	22	24	23
Alg. 4	21	24	19
Alg. 5	25	26	23
Alg. 6	22	24	23

The constrained select functions find sets of points close to the LCH while only visiting a fraction of the Υ networks. However, Figure 5.6 demonstrates the potential shortcomings of the constrained algorithms. Since they are constrained to only select nodes from the subset $\mathcal{O}' \subset \mathcal{O}$ of deepest open nodes, they cannot select previously visited nodes that are not in $\mathcal{O}'$, even if they are better options. The unconstrained algorithms, on the other hand, do not have this problem, and can find sets of points that more closely approximate the set of optimal points. However, this comes at the cost of a higher number of trained networks.

By analyzing the networks that make up the LCH approximations, we find grounding for a popular MLP design strategy. The GLCH tree nodes, for the different select functions, are present in the tables of Annex II. The nodes that make up the LCH approximations are marked in boldface. The set of all possible network architectures included

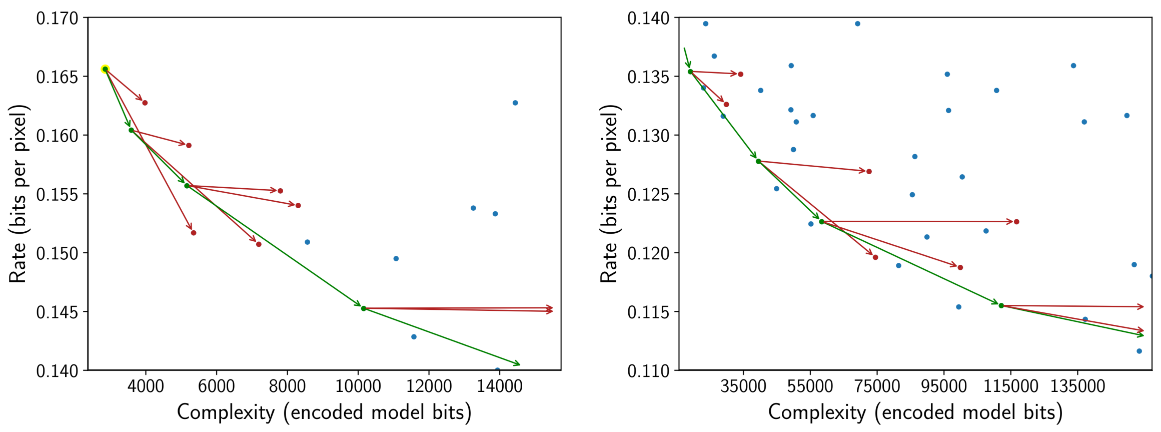


Figure 5.6: Zoomed portions of the bottom right plot of Figure 5.5, which demonstrate the shortcomings of a constrained select function. Because of the constraint of only selecting nodes among the current deepest open nodes, when they find more than one point that make part of the LCH, they must select one, and the other cannot be selected later in a future iteration. The unconstrained select functions do not suffer from this issue.

examples of pyramid, inverted pyramid, and rectangular structure. Nevertheless, the networks on the LCH approximations ended up having a pyramid structure. This seems to confirm the common pyramid design strategy many practitioners use when designing MLPs. We have also used such strategy in Section 4.2.

Regarding the feasibility of including the neural network along with the data in the compressed file: note that the number of pixels in the training set is $10 \times 768 \times 1024 = 7864320$, while the number of model bits is in the range from 10^3 to 10^7. In general, the number of model bits per data sample is much lower than the number of data bits per sample. This means that, in many cases, it is feasible to include the network along with the data in the compressed file. In some cases, it may even be beneficial, if compared to other compression alternatives.

So far, we have only qualitatively evaluated the GLCH algorithm. We can evaluate it quantitatively by computing the hypervolume of the visited points, and comparing it with the hypervolume considering all Υ architectures. We defined the hypervolume in Section 3.7. To recapitulate, the hypervolume of a set of points is the volume of the space Pareto-dominated by those points. A point Pareto-dominates another when it is better or equal in all objectives, and strictly better in at least one objective. Note that, in this work, we assume that an objective is better when it is lower. If this is not the case for one objective, we can make it the case, by multiplying the objective by -1. When computing the hypervolume, it is necessary to specify a reference point, otherwise the dominated hypervolume would always be infinite. We set each coordinate of the reference point to a value 10% higher than the highest value in that coordinate. When obtaining the highest value in each coordinate, we include all the points involved in the hypervolume calculations with respect to that reference point. The dominated hypervolume is then computed as the volume of the dominated space, bounded from above by this reference point (as depicted in Figure 3.5). The maximum possible hypervolume is only achieved by sets which contain the Pareto set, that is, the set of pareto optimal solutions. Evidently, the set of all possible Υ network architectures contains the Pareto set, therefore the hypervolume for the Υ network architectures is the maximum. The LCH and the Pareto set are *almost* equivalent (as discussed in Section 3.6, and depicted in Figure 3.3). Therefore, sets that contain the LCH have close to maximum dominated hypervolume. In other words, a method to compute the LCH is good when the distance from its hypervolume to the maximum possible hypervolume is small. We refer to this quantity as the hypervolume difference.

A common way to compare multi-objective hyperparameter optimization methods, when the maximum possible hypervolume is known, is to compare their hypervolume differences, as the number of visited networks increases. The best method, for a given number of visited networks, is the one which gives the lowest value of hypervolume difference. The number of points that can be visited depends of the amount of budget that is available. The method that gives the best LCH approximation might differ from one number of visited networks to another.

Figure 5.7 shows the log hypervolume difference of several methods, relative to the hypervolume of the Υ architectures, as the number of visited networks increases. Besides the five variants of the GLCH algorithm, we also included results for three other methods, obtained using a popular platform for multiobjective optimization [105]. The three methods are Sobol, qNEHVI and qNParEGO. The first one essentially corresponds to the random search method described in Section 3.5. However, it specifically selects points based on the Sobol sequence [106], which provides a more uniform coverage of the sample space. The qNEHVI and qNParEGO are noisy parallel versions of the EHVI and ParEGO methods mentioned in Section 3.8.

We run these methods for a number of iterations equal to the maximal number of iterations of the GLCH methods. However, note that the Bayesian Optimization methods operate in continuous domains. The suggested hyperparameters in a given iteration are continuous. After rounding the hyperparameters, the suggested architecture in a given iteration may be one already suggested in a previous iteration. In the plots of Figure 5.7, the horizontal axis is the number of *distinct* networks that have been visited. This applies to all methods. Note that a child node at a given iteration of the GLCH algorithm, with an unconstrained select function, may also correspond to a node that has already been explored before. This is because a node can be reached from different paths in the architecture graph. In that case, the repeated node is ignored, and is not counted as a newly explored network. In the plots of Figure 5.7, we have already accounted for this, and removed duplicates when counting the number of visited networks reported in the horizontal axis.

We run each of the Sobol, qNEHVI and qNParEGO methods 25 times, each time with a different set of initialization points. The first six points in each run are initialization points, and are shared between these three methods. The maximum number of visited networks is different in each run of each method. For each method, we take the average of the HV curves over the different runs. When computing the average HV for a given number of visited networks, it is evident that we can only consider the runs in which the number of visited networks have reached that particular value. We only keep the averages computed using at least 10 runs. That is one of the reasons why the curves in the plots of Figure 5.7 do not go up to the same number of visited networks. Other reason is that the GLCH methods naturally terminate with different numbers of visited networks.

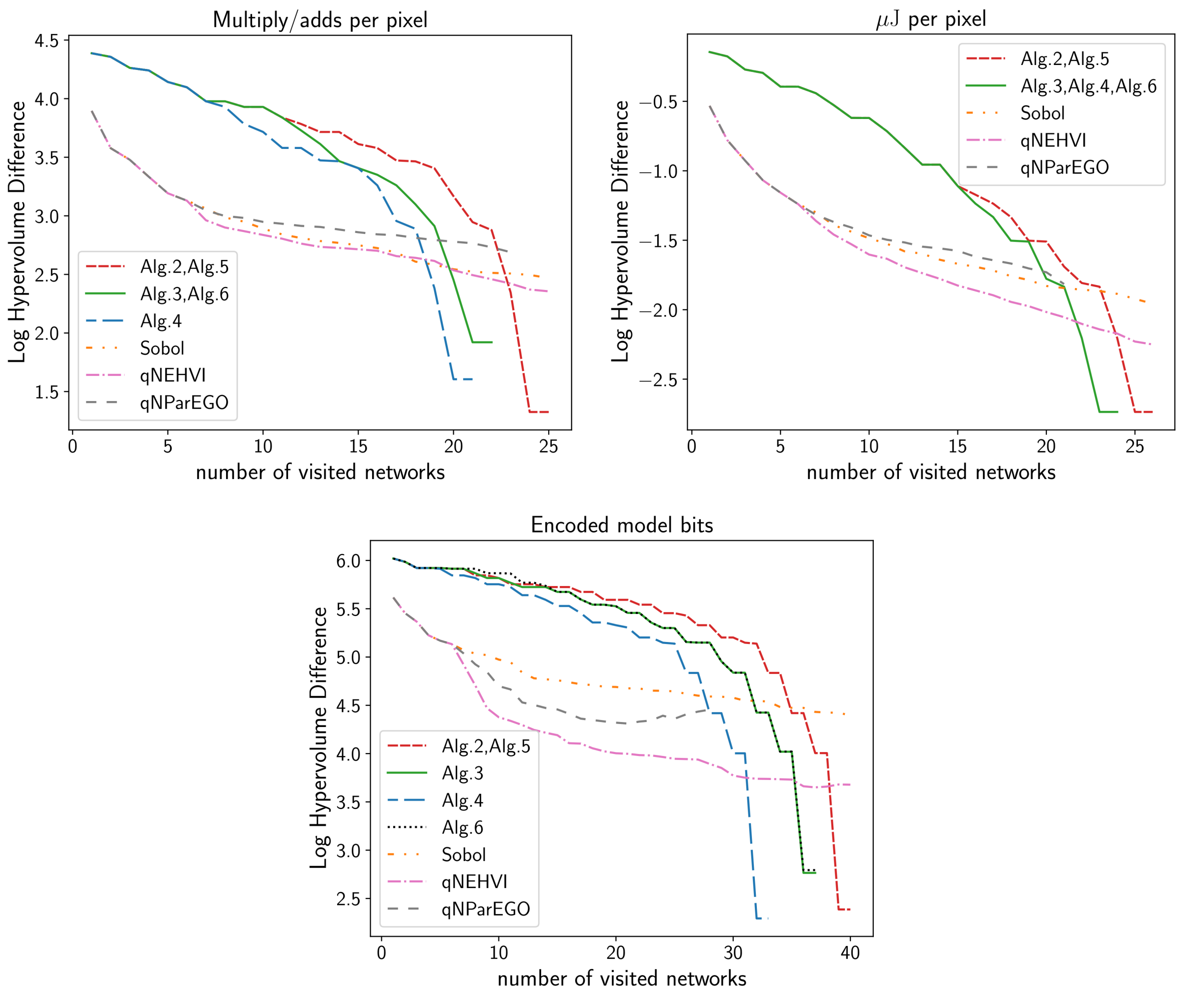


Figure 5.7: Comparison between the different versions of the GLCH algorithm and other MOHPO methods. The methods are compared in terms of log hypervolume difference with respect to the maximal hypervolume, which is obtained considering all combinations of hyperparameters.

Analyzing Figure 5.7 we can see that, for low numbers of visited networks, Sobol, qNEHVI and qNParEGO provide a better approximation of the overall pareto front. However, when the GLCH methods are close to terminate execution, they surpass the other methods for the same number of visited networks. This is a consequence of the fact that the GLCH methods build the LCH in increasing order of complexity.

If, instead of considering all points, we only consider the points up to the complexity level already achieved by the GLCH method, the advantages of the GLCH method become clear. In Figure 5.8, we show the results for the select function in Algorithm 5. Similar results hold for the other select functions. To obtain the plots of Figure 5.8, we consider the history of visited networks by the methods. The GLCH method visits the networks in increasing order of complexity. When the GLCH method reaches a given level of complexity, we compare its suggested networks with the suggested networks by the other methods, for the same number of visited networks. We calculate the hypervolume of the points with complexity lower than the GLCH complexity level. Then we take its log difference with respect to the maximum hypervolume. A lower value indicates a better LCH approximation for the region up to that complexity level. As before, the curves of Sobol, qNEHVI and qNParEGO are averages over multiple runs. We use the same runs used to generate the plots of Figure 5.7. Again, we only keep curve points which are computed by averaging results from at least 10 runs.

Therefore, if one is only interested on the LCH *up to a given level of complexity* one can largely benefit from the GLCH algorithm. It visits networks in increasing order of complexity, giving a better LCH approximation than the other methods up to the currently explored complexity level. Even if one is interested on the LCH *for all levels of complexity*, the GLCH can still give better LCH approximations for specific numbers of visited networks.

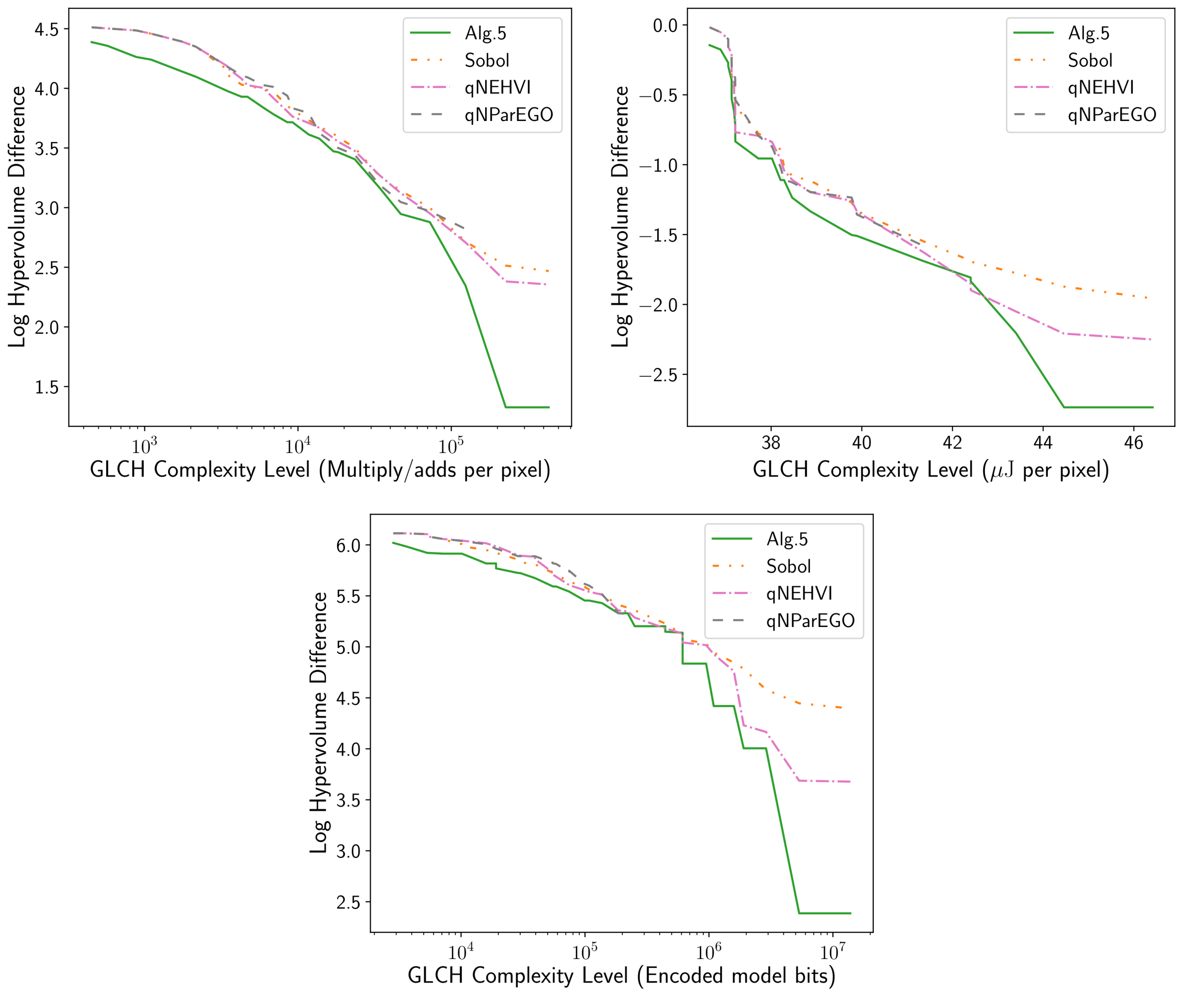


Figure 5.8: Comparison between the GLCH algorithm and other MOHPO methods. The methods are compared in terms of hypervolume difference up to the complexity level achieved by the GLCH method.

5.9 Rate-Distortion-Complexity Optimization in Lossy Image Compression

In Section 5.8, we have shown that the GLCH method can effectively be used to select neural networks for lossless compression. However, the applicability of the GLCH method actually extends to more than simply lossless compression. It can also be used in lossy compression with much effectiveness.

This section is devoted to testing the GLCH method in lossy compression. More specifically, we use the GLCH algorithm to optimize complexity in lossy image compression. This problem is already inherently a multi-objective optimization problem of rate and distortion. We also want to take complexity into account.

We assume that the compromise between rate and distortion is already included in the loss function of the neural network, but the complexity is not. Specific network architectures may lead to overfitting, or may be excessively complex, therefore it is of interest to consider multiple architectures. After considering several architectures, the best one, or the best ones, must be selected. The GLCH algorithm tackles both of the problems of finding the best architectures among the available options, and of deciding which architectures to train. In this way, it finds a good approximation of the overall LCH, even if we include the points which were not visited by the GLCH algorithm. It returns an approximate set of the optimal architectures for the different tradeoffs between rate, distortion and complexity.

In order to evaluate the GLCH method in a lossy setting, we consider the compression of colored images with the VAE compression approach of Section 3.9. For the purpose of the experiments of this section, we use, as training data, a small sub-set, composed of 240 frames, of the Vimeo-90K dataset [14], and, as validation data, the entire Kodak dataset [3], composed of 24 images.

We consider variants of the *bmshj2018-factorized* model from [107], which is based on [27]. Figure 5.9 illustrates the model template, and the hyperparameters we vary.

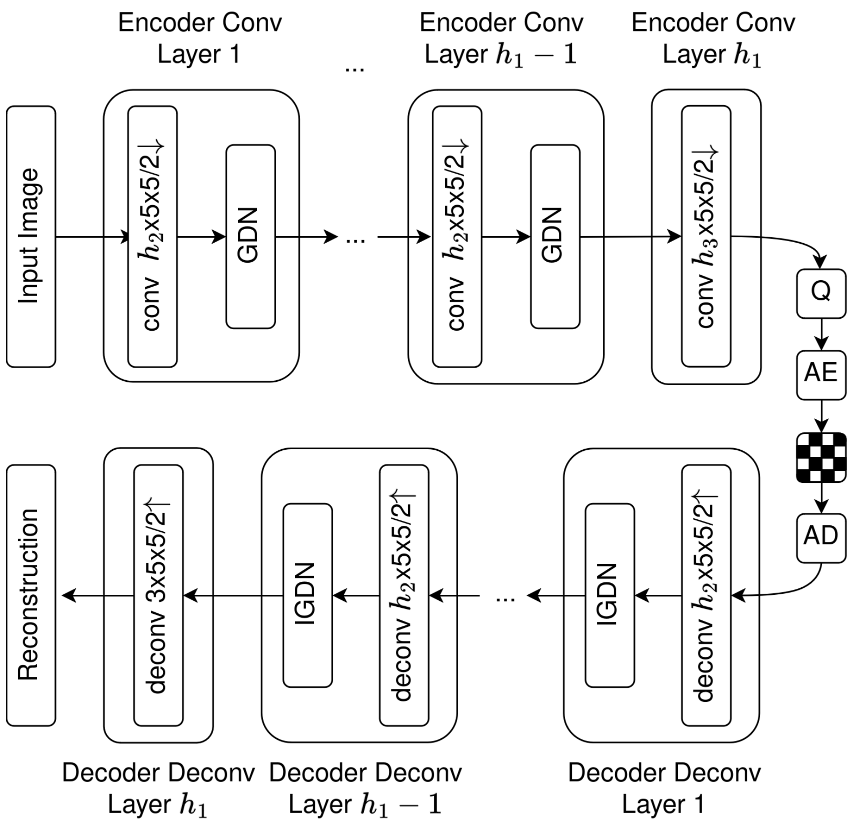


Figure 5.9: Template of the considered VAE architectures. GDN and IGDN indicate the activation functions in the encoder and the decoder [27]. Conv h_2x5x5/2↓ indicate a convolutional layer of h_2 kernels (filters) with width and height equal to 5 pixels, and a stride of 2. Deconv h_2x5x5/2↑ indicate a transposed convolutional layer of h_2 kernels (filters) with width and height equal to 5 pixels, and a stride of 2. See [108] for an introduction on the convolutional and the transposed convolutional layers. Based on Figure 4 from [27].

Those are: the number of layers of the encoder and the decoder (h_1), the number of filters of all convolutional layers of the encoder and the decoder, except for the last convolutional layers (h_2), and the number of filters of the last convolutional layer of the encoder (h_3). The values considered for h_1 are 3 and 4, while h_2 varies from 32 to 224 in steps of 32, and h_3 varies from 32 to 320 in steps of 32. In other words, in this case the number of hyperparameters is $K = 3$, the numbers of hyperparameter values are $T_1 = 2, T_2 = 7, T_3 = 10$, and the total number of architectures is $\Upsilon = T_1T_2T_3 = 140$.

The networks are trained for 10000 epochs, with a learning rate of 0.0004 and a batch size of 16. We consider values of $\lambda = 255^2\lambda'$, for $\lambda' = 0.005$, 0.01 and 0.02, with rate measured in bits per pixel, and distortion measured in mean squared error (MSE), with the pixel values normalized between 0 and 1. Therefore, if we also consider the λ as a hyperparameter (h_4), we have $K = 4$ hyperparameters, $T_1 = 2, T_2 = 7, T_3 = 10, T_4 = 3$, and $\Upsilon = T_1T_2T_3T_4 = 420$ possible networks. However, we do not consider λ as an hyperparameter in the GLCH algorithm for lossy compression, and treat each λ separately. It is only after finding the LCH approximations on the LC-planes for the different λ that the results are combined to obtain the overall LCH approximation on the RDC-space. We measure complexity either as the number of parameters of the network, or in number of floating-point operations (FLOPs). The numbers of FLOPs are obtained using the ptflops tool [109].

Figure 5.10 shows the clouds of loss-complexity points for the different λ, and the LCH approximations found by the GLCH algorithm, using the select function in Algorithm 6. We only show the parent nodes, which can be seen as the approximation of the LCH. The graphs for the other variants of the GLCH algorithm are shown in Annex III. The plots are arranged as follows. The top row corresponds to $\lambda' = 0.005$, the middle row corresponds to $\lambda' = 0.01$ and the bottom row to $\lambda' = 0.02$. The left column corresponds to complexity measured in number of parameters, and the right column corresponds to complexity measured in number of FLOPs.

Combining the points found for the different λ, we obtain an estimate of the LCH for the rate-distortion-complexity cloud of points. Figure 5.11 shows the parent nodes found by the GLCH algorithm for the different λ, whose combination can be seen as an approximation of the LCH on the RDC-space. The results are for the select function in Algorithm 6.

Figure 5.12 shows the hypervolume differences of the GLCH methods as the number of trained networks increases. In Figure 5.12, the reference hypervolume is the hypervolume in the RDC-space of the 420 networks, for the different h_1, h_2, h_3 and the different λ. We include results for Sobol, qNEHVI and qNParEGO as well. The plots in Figure 5.12 are similar to the plots in Figure 5.7.

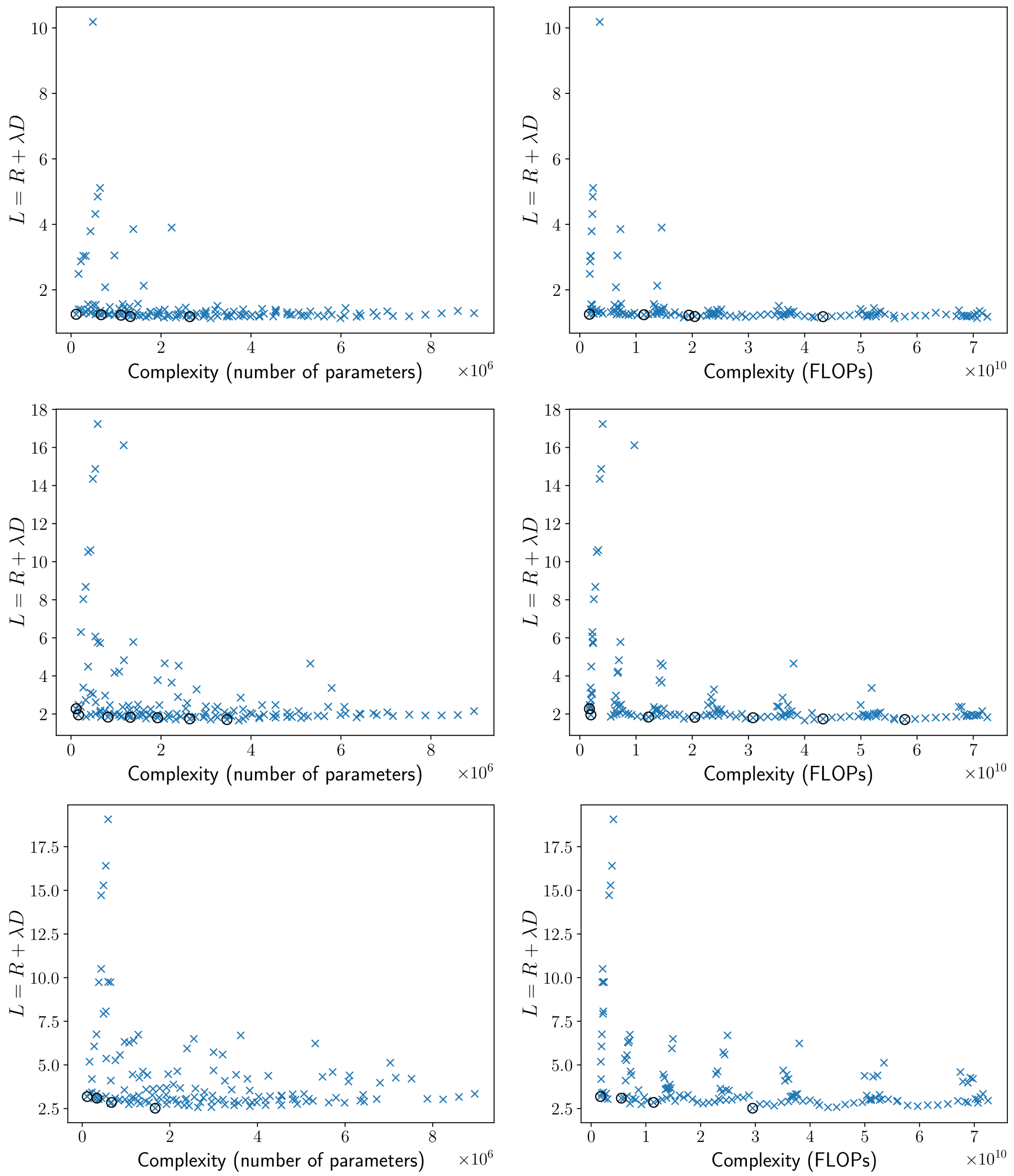


Figure 5.10: Clouds of loss-complexity points, with emphasis on the parent nodes found by the GLCH algorithm, using the select function in Algorithm 6. The loss is equal to $R + \lambda D$, where λ is equal to $255^2 \times 0.005$ (top row), $255^2 \times 0.01$ (middle row) and $255^2 \times 0.02$ (bottom row), R is the rate in bits per pixel, and D is the distortion in mean squared error, with the pixel values normalized between 0 and 1. Complexity is given in number of parameters (left column), and in number of FLOPs (right column).

The GLCH methods present competitive results, particularly for a number of trained networks lower than 60, in which case they dominate the other methods. For more than 60

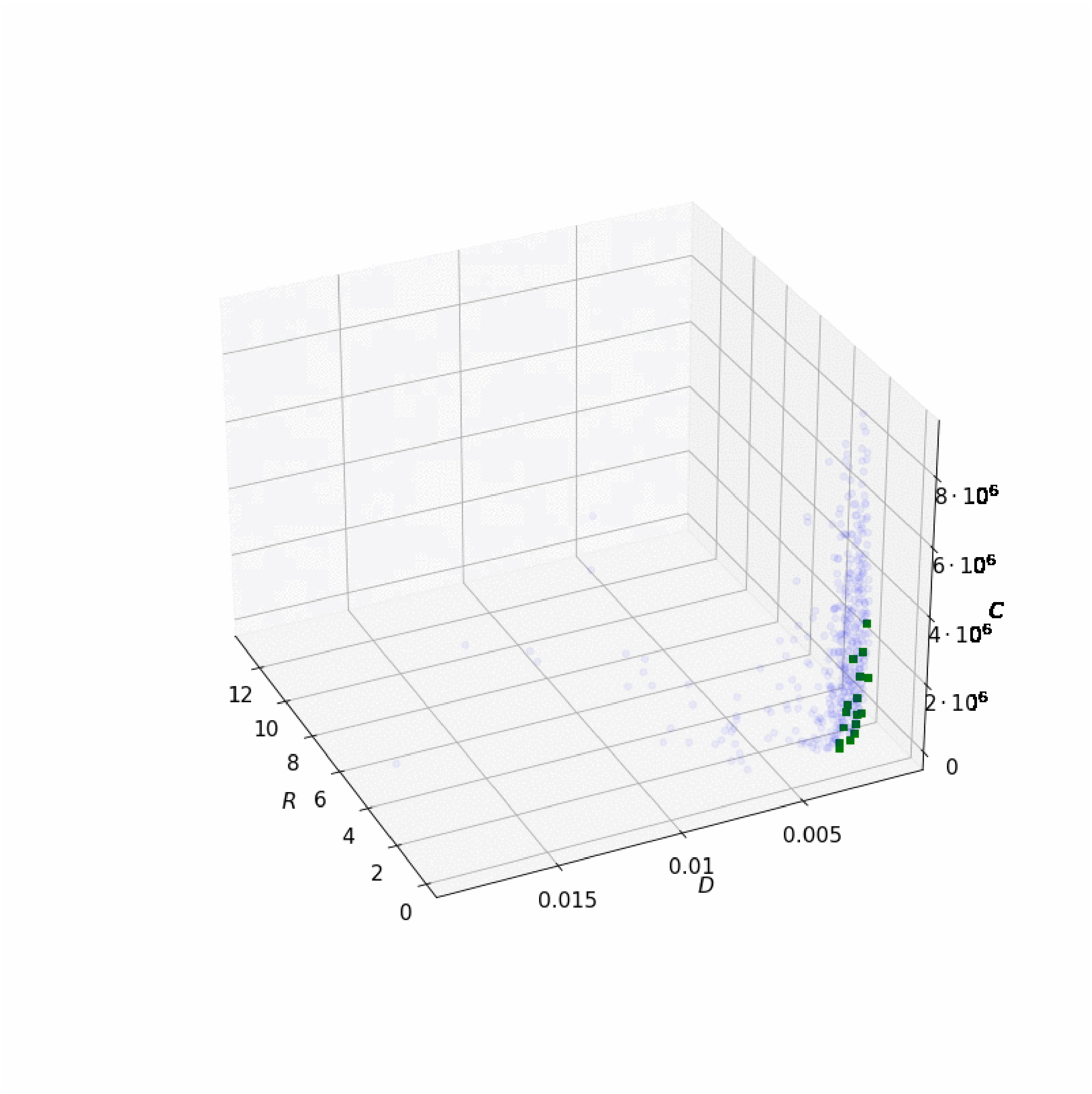


Figure 5.11: Animation of the rate-distortion-complexity cloud of points, and an LCH approximation, found by the GLCH algorithm, using the select function in Algorithm 6. In this figure, we only highlight the nodes which were parent nodes during the execution of the GLCH algorithm. These points are represented as green cubes. The other points are represented as blue spheres. To view this animation, open this PDF file in Acrobat Reader.

trained networks, qNEHVI catches up with the GLCH methods, for complexity measured in number of parameters. It seems that it would show better results for larger numbers of trained networks. The qNEHVI curve stops at around 60 trained networks. This is

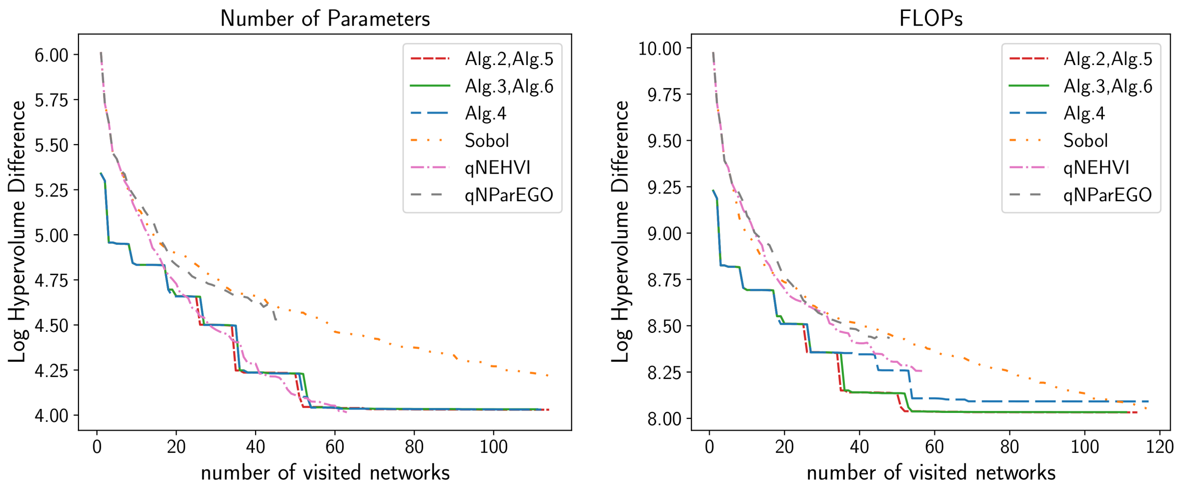


Figure 5.12: Performances of the different versions of the GLCH algorithm, and other MOHPO methods, on the rate-distortion-complexity optimization problem. The methods are compared in terms of log hypervolume difference with respect to the maximal hypervolume, which is obtained considering all of the 420 networks, for the different h_1, h_2, h_3 and the different λ.

so, because in the 25 times we ran this method and for a number of iterations matching the number of iterations of the GLCH methods, it only reached 60 trained networks less than 10 times. Note that we only keep averages computed with more than 10 samples. However, if we do include the remainder of the curve, with values computed using less than 10 samples, the qNEHVI method does indeed surpass the GLCH methods for more than 60 trained networks. However, it is important to note that the results for qNEHVI are averages. The results for a specific run may be better or worse. In the GLCH case, if the rate-complexity-distortion performance of the points are pre-defined, the result is the same at every run.

Chapter 6

Conclusion

In this work, our focus was to investigate neural-based adaptive context modeling, while also considering the neural network design process. In order to achieve this goal, we broke down this main objective into two more specific secondary objectives: 1) to propose and investigate a neural-based replacement for the look-up table (LUT) and 2) to propose and investigate a method of MOHPO more specific to data compression.

We tackled the first objective with the proposal of the adaptive perceptron coding (APC) method. Differently from the other currently available alternatives, the APC method does not require pre-training and continuously adapts to the signal statistics. Continuous adaptation is an attractive quality of most context modeling approaches using LUTs, and it is of interest to have a neural-based method with this characteristic. Such a method combines the approximation power of neural networks with the adaptivity of most LUT-based methods. Since every new sample is different, it continuously learns the signal statistics, and approaches the true probability distribution of the data, instead of the training data probability distribution. We compared the APC method to alternatives using LUTs and RNNs, and showed that the APC method is able to overcome the shortcomings of the LUT-based methods. Since, in principle, a LUT requires a separate storage space for every conditional probability estimate for every context size, and the growth of the number of possible contexts is exponential with respect to the context size, memory consumption is a real issue with LUT-based methods. With APC, we are able to increase the context size much further, and to consequently decrease the average codeword length.

We tackled the second objective with the proposal of the greedy lower convex hull (GLCH) method. We developed the GLCH method to minimize rate, or rate and distortion, while also minimizing complexity. However, it could also be used to minimize other objectives, even outside of the data compression field. The output of the GLCH method is an approximation of the LCH, or the Pareto front, for the multiple objectives considered. An attractive quality of the GLCH method is that it progressively builds this approximation. It first builds the approximation up to a given value of one objective, in the case of our applications, up to a complexity value, then to a higher value, and so on. In MOHPO, it is generally the case that the degradation of one objective is associated with the improvement of other objective, and vice versa [53]. For example, an increase in complexity generally causes a reduction in average codeword length. The progressive construction of the LCH by the GLCH algorithm allows one to stop execution once the maximum, or the minimum, acceptable value of one objective is achieved. As our comparison results demonstrate, other MOHPO methods build the entire LCH for the entire domain from the very beginning, and do not have this advantage. This means that the LCH approximation with the GLCH method is generally better, up to a given value of complexity. In data compression, we are often interested in solutions only up to a certain level of complexity, and the GLCH method permits focusing on those solutions. The GLCH method also showed competitive overall results compared to state-of-the-art methods present in a popular MOHPO platform.

Other minor contributions of our work include a brief analysis of the neural networks that compose the LCH, which gave more groundings to the pyramid design strategy commonly used when designing MLPs. That is, designs with more hidden units in the initial hidden layers, compared to the final hidden layers. We also proposed a variant of the Xavier initialization method, which consistently gave better results, and embedded it into APC.

6.1 Future work

The online-trained neural-based methods are still much slower than the LUT-based methods. However, we did not investigate how faster it can become with dedicated hardware, which could be tested with, for example, Field-Programmable Gate Arrays (FPGAs). Furthermore, it is expected that, in the long run, with the advancement of hardware speed and processing power, online-trained neural-based methods will eventually become more appealing.

The APC can also be tested for the task of encoding any type of data on the fly. Since in APC, just like in CABAC, all data is converted to binary, and then encoded using context modeling and arithmetic coding, in principle it can be used to encode any type of data. Adjustments would have to be made to make the context as agnostic as possible. Since the context modeling in APC is adaptive, it can continuously approach the statistics of the different sources, as the data type changes from one type to another. The fact that the APC can be used with larger context sizes, compared to LUT-based methods, can make it better for this task as well.

The GLCH method can be explored in different areas of application other than data compression. Our application of the GLCH method to rate-distortion-complexity optimization explored the fact that a network for lossy compression is generally trained to minimize a loss of the form $L = R + \lambda D$. Therefore it was natural to apply GLCH to the LC-planes for different λ, and then combine the results to obtain an estimate of the LCH of the RDC-space. However, the question remains if this approach would also be effective in other areas of application.

Bibliography

[1] G. J. J. Verhoeven, "It's all about the format - unleashing the power of raw aerial photography," *International Journal of Remote Sensing*, vol. 31, no. 8, pp. 2009–2042, 2010.

[2] D. Vatolin, I. Seleznev, and M. Smirnov, "Lossless video codecs comparison '2007," Graphics and Media Lab Video Group, Moscow State University, Moscow, Russia, Tech. Rep., Mar. 2007.

[3] R. Franzen. "Kodak lossless true color image suite." (2013), [Online]. Available: `http://r0k.us/graphics/kodak/` (visited on 03/17/2024).

[4] J. Ballé, G. Toderici, L. Versari, *et al.*, "6th challenge on learned image compression," compression.cc, Accessed: March 17, 2024. [Online]. Available: `https://compression.cc/`.

[5] M. Mahoney, "Large text compression benchmark," mattmahoney.net, Accessed: March 17, 2024. [Online]. Available: `https://www.mattmahoney.net/dc/text.html`.

[6] K.-C. Chen, W.-H. Peng, and C. G. G. Lee, "Overview of intelligent signal processing systems," *APSIPA Transactions on Signal and Information Processing*, vol. 12, Jan. 2023.

[7] G. Sullivan and J.-R. Ohm, "Meeting report of the 22nd meeting of the joint video experts team (jvet), by teleconference, 20-28 april 2021," JVET ISO/IEC JTC 1/SC 29/WG 5, Apr. 2021.

[8] A. D. Thompson. "Inside language models (from gpt to olympus)." (2024), [Online]. Available: `https://lifearchitect.ai/models/` (visited on 06/11/2024).

[9] compression.ai, "World's best image compression," compression.ai, Accessed: March 22, 2024. [Online]. Available: `https://compression.ai/`.

[10] JPEG, "Overview of jpeg ai," jpeg.org, Accessed: March 22, 2024. [Online]. Available: `https://jpeg.org/jpegai/`.

[11] G. Martin-Cocher, "Proposed final requirements for learned-based point cloud coding," WG 07 MPEG-I, Apr. 2024.

[12] N. Guerin, R. Silva, M. Oliveira, *et al.*, "Rate-constrained learning-based image compression," *Signal Processing: Image Communication*, vol. 101, p. 116 544, Nov. 2021.

[13] B. Cottier, "Trends in the dollar training cost of machine learning systems," epochai.org, Accessed: March 23, 2024. [Online]. Available: `https://epochai.org/blog/trends-in-the-dollar-training-cost-of-machine-learning-systems`.

[14] T. Xue, B. Chen, J. Wu, D. Wei, and W. T. Freeman, "Video enhancement with task-oriented flow," *International Journal of Computer Vision*, vol. 127, no. 8, pp. 1106–1125, Feb. 2019.

[15] M. Hobbhahn and T. Besiroglu, "Trends in gpu price-performance," epochai.org, Accessed: April 9, 2024. [Online]. Available: `https://epochai.org/blog/trends-in-gpu-price-performance`.

[16] B. Maltinsky, J. Gallagher, and J. Taylor, "Feasibility of training an agi using deep rl: A very rough estimate," Median Group, Tech. Rep., Mar. 2019.

[17] Y. Sun, N. B. Agostini, S. Dong, and D. Kaeli, "Summarizing cpu and gpu design trends with product data," 2020. arXiv: `1911.11313`.

[18] Q. Liu, Y. Xu, and Z. Li, “Decmac: A deep context model for high efficiency arithmetic coding,” in *2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, 2019, pp. 438–443.

[19] M. Goyal, K. Tatwawadi, S. Chandak, and I. Ochoa, “Deepzip: Lossless data compression using recurrent neural networks,” in *2019 Data Compression Conference (DCC)*, 2019, pp. 575–575.

[20] R. Song, D. Liu, H. Li, and F. Wu, “Neural network-based arithmetic coding of intra prediction modes in hevc,” *2017 IEEE Visual Communications and Image Processing (VCIP)*, pp. 1–4, 2017.

[21] C. Ma, D. Liu, X. Peng, Z.-J. Zha, and F. Wu, “Neural network-based arithmetic coding for inter prediction information in hevc,” in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2019, pp. 1–5.

[22] E. C. Kaya and I. Tabus, “Neural network modeling of probabilities for coding the octree representation of point clouds,” *2021 IEEE 23rd International Workshop on Multimedia Signal Processing (MMSP)*, pp. 1–6, 2021.

[23] L. Huang, S. Wang, K. Wong, J. Liu, and R. Urtasun, “Octsqueeze: Octree-structured entropy model for lidar compression,” in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 1310–1320.

[24] J. Schmidhuber and S. Heil, “Sequential neural text compression,” *IEEE Transactions on Neural Networks*, vol. 7, no. 1, pp. 142–146, 1996.

[25] M. Goyal, K. Tatwawadi, S. Chandak, and I. Ochoa, “Dzip: Improved general-purpose loss less compression based on novel neural network modeling,” in *2021 Data Compression Conference (DCC)*, 2021, pp. 153–162.

[26] J. Ballé, V. Laparra, and E. Simoncelli, “End-to-end optimized image compression,” in *International Conference on Learning Representations*, 2016.

[27] J. Ballé, D. Minnen, S. Singh, S. J. Hwang, and N. Johnston, "Variational image compression with a scale hyperprior," in *International Conference on Learning Representations*, 2018.

[28] D. Minnen, J. Ballé, and G. D. Toderici, "Joint autoregressive and hierarchical priors for learned image compression," in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31, Curran Associates, Inc., 2018.

[29] Z. Cheng, H. Sun, M. Takeuchi, and J. Katto, "Learned image compression with discretized gaussian mixture likelihoods and attention modules," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 7936–7945.

[30] E. Agustsson, D. Minnen, N. Johnston, J. Ballé, S. J. Hwang, and G. Toderici, "Scale-space flow for end-to-end optimized video compression," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 8500–8509.

[31] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *2nd International Conference on Learning Representations (ICLR)*, 2014.

[32] T. Mikolov, M. Karafiát, L. Burget, J. Cernocký, and S. Khudanpur, "Recurrent neural network based language model," in *Proceedings of the 11th Annual Conference of the International Speech Communication Association, INTERSPEECH 2010*, vol. 2, Makuhari, Chiba, Japan, Jan. 2010, pp. 1045–1048.

[33] B. Krause, E. Kahembwe, I. Murray, and S. Renals, "Dynamic evaluation of neural sequence models," in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, Stockholm, Sweden: PMLR, Oct. 2018, pp. 2766–2775.

[34] Y. Gao, R. Feng, Z. Guo, and Z. Chen, "Exploring the rate-distortion-complexity optimization in neural image compression," 2023. arXiv: `2305.07678`.

[35] J. Guo, D. Xu, and G. Lu, "Cbanet: Toward complexity and bitrate adaptive deep image compression using a single network," *IEEE Transactions on Image Processing*, vol. 32, pp. 2049–2062, 2023.

[36] H. Louati, S. Bechikh, A. Louati, A. Aldaej, and L. B. Said, "Joint design and compression of convolutional neural networks as a bi-level optimization problem," *Neural Computing and Applications*, vol. 34, pp. 15 007–15 029, 2022.

[37] M. Tonin and R. L. de Queiroz, "On quantization of image classification neural networks for compression without retraining," in *2022 IEEE International Conference on Image Processing (ICIP)*, 2022, pp. 916–920.

[38] L. Qin and J. Sun, "Model compression for data compression: Neural network based lossless compressor made practical," in *2023 Data Compression Conference (DCC)*, 2023, pp. 52–61.

[39] J.-H. Kim, J.-H. Choi, J. Chang, and J.-S. Lee, "Efficient deep learning-based lossy image compression via asymmetric autoencoder and pruning," in *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2020, pp. 2063–2067.

[40] T. A. da Fonseca and R. L. de Queiroz, "Energy-constrained real-time h.264/avc video coding," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 1739–1743.

[41] T. A. Fonseca and R. L. de Queiroz, "Towards greener computing systems for video compression," *Journal of Communication and Information Systems*, vol. 30, no. 1, May 2015.

[42] E. Hosseini, F. Pakdaman, M. R. Hashemi, and M. Ghanbari, "Fine-grain complexity control of hevc intra prediction in battery-powered video codecs," *Journal of Real-Time Image Processing*, vol. 18, pp. 603–618, 2021.

[43] C. Herglotz, F. Brand, A. Regensky, F. Rievel, and A. Kaup, “Processing energy modeling for neural network based image compression,” in *2023 IEEE International Conference on Image Processing (ICIP)*, 2023, pp. 2390–2394.

[44] Z. He, Y. Liang, L. Chen, I. Ahmad, and D. Wu, “Power-rate-distortion analysis for wireless video communication under energy constraints,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 5, pp. 645–658, 2005.

[45] A. Ortega and K. Ramchandran, “Rate-distortion methods for image and video compression,” *IEEE Signal Processing Magazine*, vol. 15, no. 6, pp. 23–50, 1998.

[46] G. Sullivan and T. Wiegand, “Rate-distortion optimization for video compression,” *IEEE Signal Processing Magazine*, vol. 15, no. 6, pp. 74–90, 1998.

[47] P. Chou and Z. Miao, “Rate-distortion optimized streaming of packetized media,” *IEEE Transactions on Multimedia*, vol. 8, no. 2, pp. 390–404, 2006.

[48] T. Yue, “Overview of rate control algorithms in mainstream video coding standards,” in *Signal and Information Processing, Networking and Computers*, Y. Wang, L. Xu, Y. Yan, and J. Zou, Eds., Singapore: Springer Singapore, 2021, pp. 696–703.

[49] Q. Wang, Z. Cheng, X. Pan, and Y. Chen, “Rate control technology in video coding,” in *Signal and Information Processing, Networking and Computers*, Y. Wang, M. Fu, L. Xu, and J. Zou, Eds., Singapore: Springer Singapore, 2020, pp. 889–895.

[50] Y. Shoham and A. Gersho, “Efficient bit allocation for an arbitrary set of quantizers (speech coding),” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, no. 9, pp. 1445–1453, 1988.

[51] P. Chou, T. Lookabaugh, and R. Gray, “Optimal pruning with applications to tree-structured source coding and modeling,” *IEEE Transactions on Information Theory*, vol. 35, no. 2, pp. 299–315, 1989.

[52] S. W. Wu and A. Gersho, “Rate-constrained picture-adaptive quantization for jpeg baseline coders,” in *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, 1993, 389–392 vol.5.

[53] F. Karl, T. Pielok, J. Moosbauer, *et al.*, “Multi-objective hyperparameter optimization in machine learning – an overview,” *ACM Transactions on Evolutionary Learning and Optimization*, vol. 3, no. 4, pp. 1–50, Dec. 2023.

[54] J. Bergstra, D. Yamins, and D. Cox, “Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures,” in *Proceedings of the 30th International Conference on Machine Learning*, S. Dasgupta and D. McAllester, Eds., ser. Proceedings of Machine Learning Research, vol. 28, Atlanta, Georgia, USA: PMLR, 17–19 Jun 2013, pp. 115–123.

[55] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, “Algorithms for hyper-parameter optimization,” in *Advances in Neural Information Processing Systems*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, Eds., vol. 24, Curran Associates, Inc., 2011.

[56] P. Liashchynskyi and P. Liashchynskyi, “Grid search, random search, genetic algorithm: A big comparison for nas,” 2019. arXiv: `1912.06059`.

[57] K. Sayood, *Introduction to Data Compression, Fifth Edition*, 5th. Cambridge, MA, USA: Morgan Kaufmann Publishers Inc., 2017.

[58] D. Salomon, *Coding for data and computer communications.* Springer, 2005.

[59] T. M. Cover and J. A. Thomas, *Elements of Information Theory.* Wiley, 2006.

[60] R. Gray and L. Davisson, *An Introduction to Statistical Signal Processing* (An Introduction to Statistical Signal Processing). Cambridge University Press, 2010.

[61] C. E. Shannon, “Prediction and entropy of printed english,” *The Bell System Technical Journal*, vol. 30, no. 1, pp. 50–64, 1951.

[62] C. M. Bishop, *Neural Networks for Pattern Recognition.* USA: Oxford University Press, Inc., 1995.

[63] F. Cady, “Maximum likelihood estimation and optimization,” in *The Data Science Handbook*. John Wiley & Sons, Ltd, 2017, ch. 23, pp. 345–356.

[64] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016.

[65] R. L. de Queiroz and P. A. Chou, “Transform coding for point clouds using a gaussian process model,” *IEEE Transactions on Image Processing*, vol. 26, no. 7, pp. 3507–3517, 2017.

[66] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing.* Cambridge, Massachusetts: The MIT Press, 1999.

[67] D. Marpe, H. Schwarz, and T. Wiegand, “Context-based adaptive binary arithmetic coding in the h.264/avc video compression standard,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 620–636, 2003.

[68] H. Schwarz, M. Coban, M. Karczewicz, *et al.*, “Quantization and entropy coding in the versatile video coding (vvc) standard,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, no. 10, pp. 3891–3906, 2021.

[69] D. G. Fernández, G. Botella, A. A. D. Barrio, C. García, M. Prieto-Matías, and C. Grecos, “Hevc optimization based on human perception for real-time environments,” *Multimedia Tools and Applications*, vol. 79, pp. 16 001–16 033, 2020.

[70] V. Sze and D. Marpe, “Entropy coding in hevc,” in *High Efficiency Video Coding*, 2014.

[71] T. Nguyen, X. Xu, F. Henry, *et al.*, “Overview of the screen content support in vvc: Applications, coding tools, and performance,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, no. 10, pp. 3801–3817, 2021.

[72] C. Holt, “Forecasting seasonals and trends by exponential weighted moving averages,” *International Journal of Forecasting*, vol. 20, pp. 5–10, Mar. 2004.

[73] S. Haykin, *Neural Networks and Learning Machines.* Pearson, 2008.

[74] M. A. Nielsen, *Neural Networks and Deep Learning.* Determination Press, 2015.

[75] G. Strang, *Linear Algebra and Learning from Data.* Wellesley-Cambridge Press, 2019.

[76] G. Garrigos and R. M. Gower, "Handbook of convergence theorems for (stochastic) gradient methods," 2023. arXiv: `2301.11235`.

[77] K. P. Murphy, *Probabilistic Machine Learning: An introduction.* MIT Press, 2022.

[78] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Pearson, 2020.

[79] P. Werbos, "Backpropagation through time: What it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[80] S. Raschka and V. Mirjalili, *Python machine learning: Machine learning and deep learning with Python, scikit-learn, and TensorFlow 2.* Packt Publishing Ltd., 2019.

[81] M. Emmerich, K. Giannakoglou, and B. Naujoks, "Single- and multiobjective evolutionary optimization assisted by gaussian random field metamodels," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 4, pp. 421–439, 2006.

[82] P. Godfrey, R. Shipley, and J. Gryz, "Algorithms and analyses for maximal vector computation," *The VLDB Journal*, vol. 16, pp. 5–28, Jan. 2007.

[83] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* Cambridge, MA, USA: MIT Press, 2001.

[84] P. I. Frazier, "A tutorial on bayesian optimization," 2018. arXiv: `1807.02811`.

[85] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the human out of the loop: A review of bayesian optimization," *Proceedings of the IEEE*, vol. 104, no. 1, pp. 148–175, 2016.

[86] B. Letham, B. Karrer, G. Ottoni, and E. Bakshy, "Constrained Bayesian Optimization with Noisy Experiments," *Bayesian Analysis*, vol. 14, no. 2, pp. 495–519, 2019.

[87] M. Balandat, B. Karrer, D. R. Jiang, *et al.*, "Botorch: A framework for efficient monte-carlo bayesian optimization," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20, Vancouver, BC, Canada: Curran Associates Inc., 2020.

[88] S. Daulton, M. Balandat, and E. Bakshy, "Parallel bayesian optimization of multiple noisy objectives with expected hypervolume improvement," in *Neural Information Processing Systems*, 2021.

[89] J. Knowles, "Parego: A hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems," *IEEE Transactions on Evolutionary Computation*, vol. 10, no. 1, pp. 50–66, 2006.

[90] S. Daulton, M. Balandat, and E. Bakshy, "Differentiable expected hypervolume improvement for parallel multi-objective bayesian optimization," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS '20, Vancouver, BC, Canada: Curran Associates Inc., 2020.

[91] J. Cai, "Convolutions of distributions," in *Encyclopedia of Actuarial Science*. John Wiley & Sons, Ltd, 2006.

[92] D. P. Kingma and M. Welling, "An introduction to variational autoencoders," *Foundations and Trends® in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019.

[93] Z. Chen and H. Zhang, "Learning implicit fields for generative shape modeling," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2019, pp. 5932–5941.

[94] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Y. W. Teh and M. Titterington, Eds., ser. Proceedings of Machine Learning Research, vol. 9, Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256.

[95] R. Gonzalez and R. Woods, *Digital Image Processing.* Pearson, 2018.

[96] Q. Zou, “A pde model for smooth surface reconstruction from 2d parallel slices,” *IEEE Signal Processing Letters*, vol. 27, pp. 1015–1019, 2020.

[97] S. Yan, P. Addabbo, C. Hao, and D. Orlando, “Adaptive detection of dim maneuvering targets in adjacent range cells,” *IEEE Signal Processing Letters*, vol. 28, pp. 633–637, 2021.

[98] M. Kuhn. “Jbig-kit.” (2018), [Online]. Available: `https://www.cl.cam.ac.uk/~mgk25/jbigkit/` (visited on 03/17/2024).

[99] J. B. W. Webber, “A bi-symmetric log transformation for wide-range data,” *Measurement Science and Technology*, vol. 24, no. 2, p. 027 001, Dec. 2012.

[100] Y. LeCun, “Generalization and network design strategies,” in *Connectionism in perspective*, R. Pfeifer, Z. Schreter, F. Fogelman, and L. Steels, Eds. Elsevier, 1989.

[101] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, *et al.*, Eds., vol. 30, Curran Associates, Inc., 2017.

[102] A. Bundy, “Artificial intelligence: Art or science?” *RSA Journal*, vol. 136, no. 5384, pp. 557–569, 1988.

[103] H. A. Simon, “Artificial intelligence: An empirical science,” *Artificial Intelligence*, vol. 77, no. 1, pp. 95–127, 1995.

[104] R. Baraniuk, D. Donoho, and M. Gavish, “The science of deep learning,” *Proceedings of the National Academy of Sciences*, vol. 117, no. 48, pp. 30 029–30 032, 2020.

[105] Facebook. “Adaptive experimentation platform.” (2024), [Online]. Available: `https://ax.dev/` (visited on 06/05/2024).

[106] I. Sobol’, “On the distribution of points in a cube and the approximate evaluation of integrals,” *USSR Computational Mathematics and Mathematical Physics*, vol. 7, no. 4, pp. 86–112, 1967.

[107] J. Bégaint, F. Racapé, S. Feltman, and A. Pushparaja, “Compressai: A pytorch library and evaluation platform for end-to-end compression research,” 2020. arXiv: `2011.03029`.

[108] V. Dumoulin and F. Visin, “A guide to convolution arithmetic for deep learning,” 2018. arXiv: `1603.07285`.

[109] V. Sovrasov. “Ptflops: A flops counting tool for neural networks in pytorch framework.” (2018-2023), [Online]. Available: `https://github.com/sovrasov/flops-counter.pytorch` (visited on 03/17/2024).

Annex I

Dataset Samples

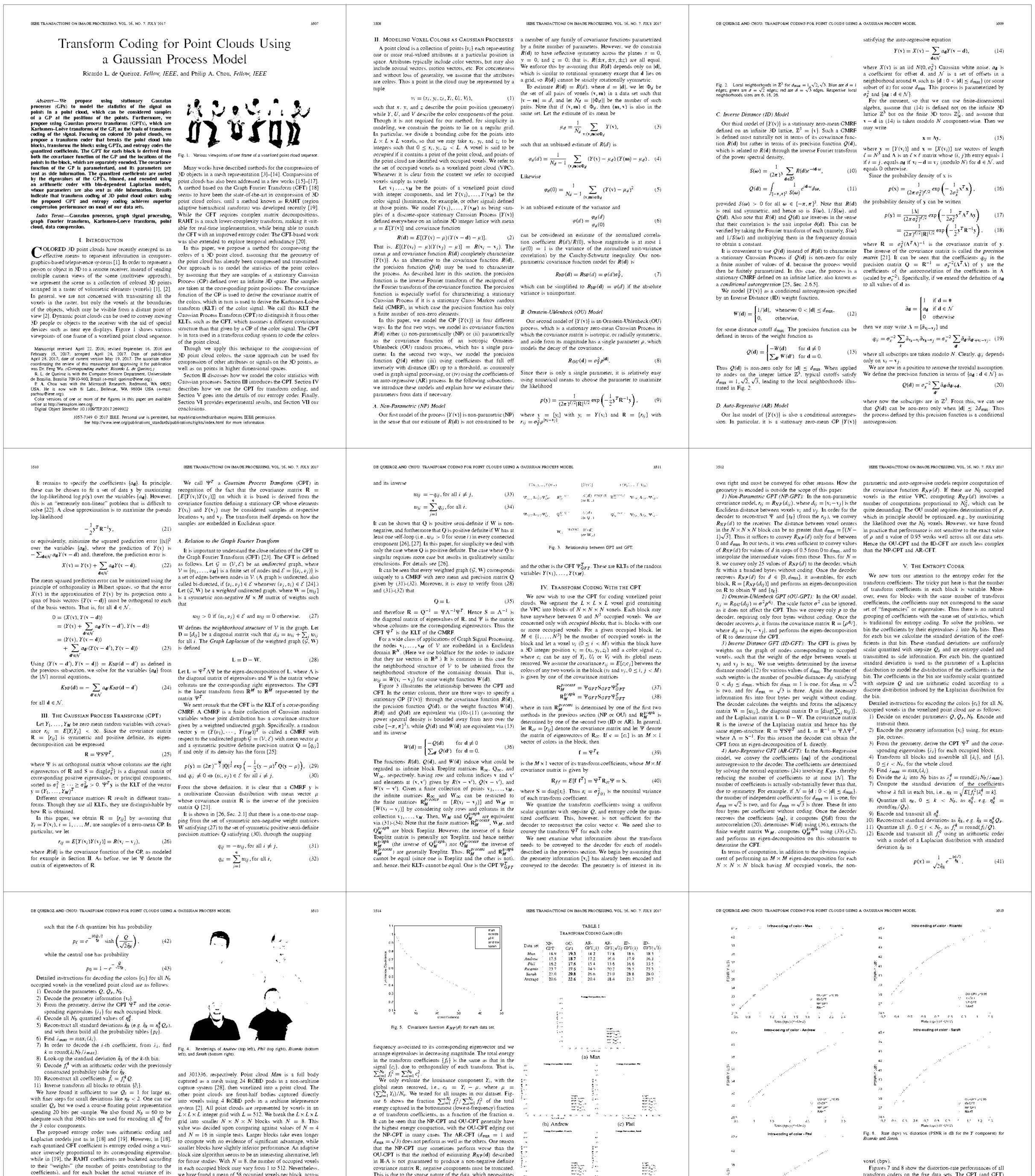


Figure I.1: First nine pages from the paper [65], which are used in Datasets 1, 2 and 3 for different purposes.

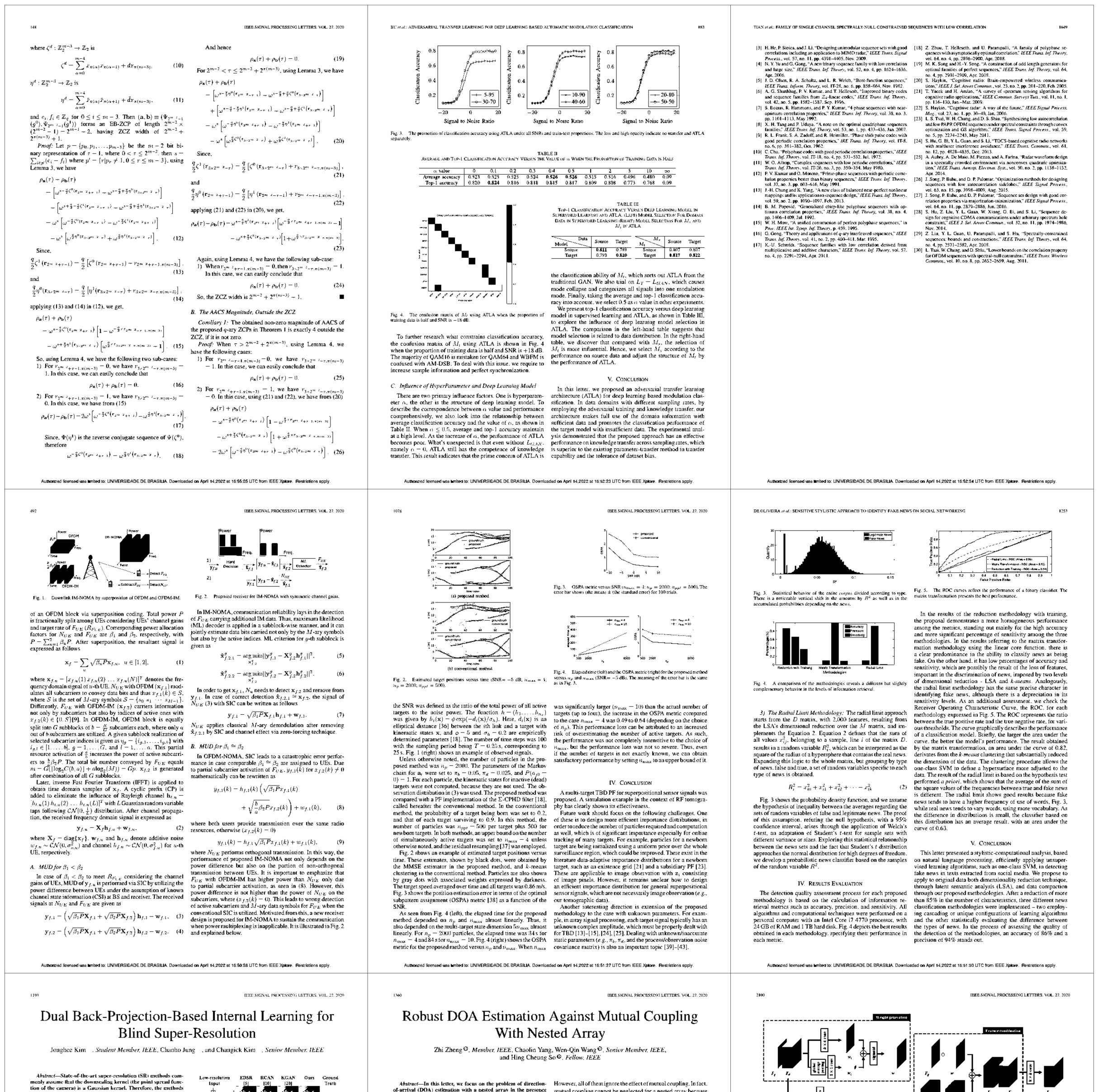


Figure I.2: Nine pages from the training set of Dataset 5, which is also the training set of Dataset 4.

Figure I.3: Four pages from the validation set of Dataset 5, which is also the validation set of Dataset 4. The first page is also the only page from the validation set of Dataset 6.

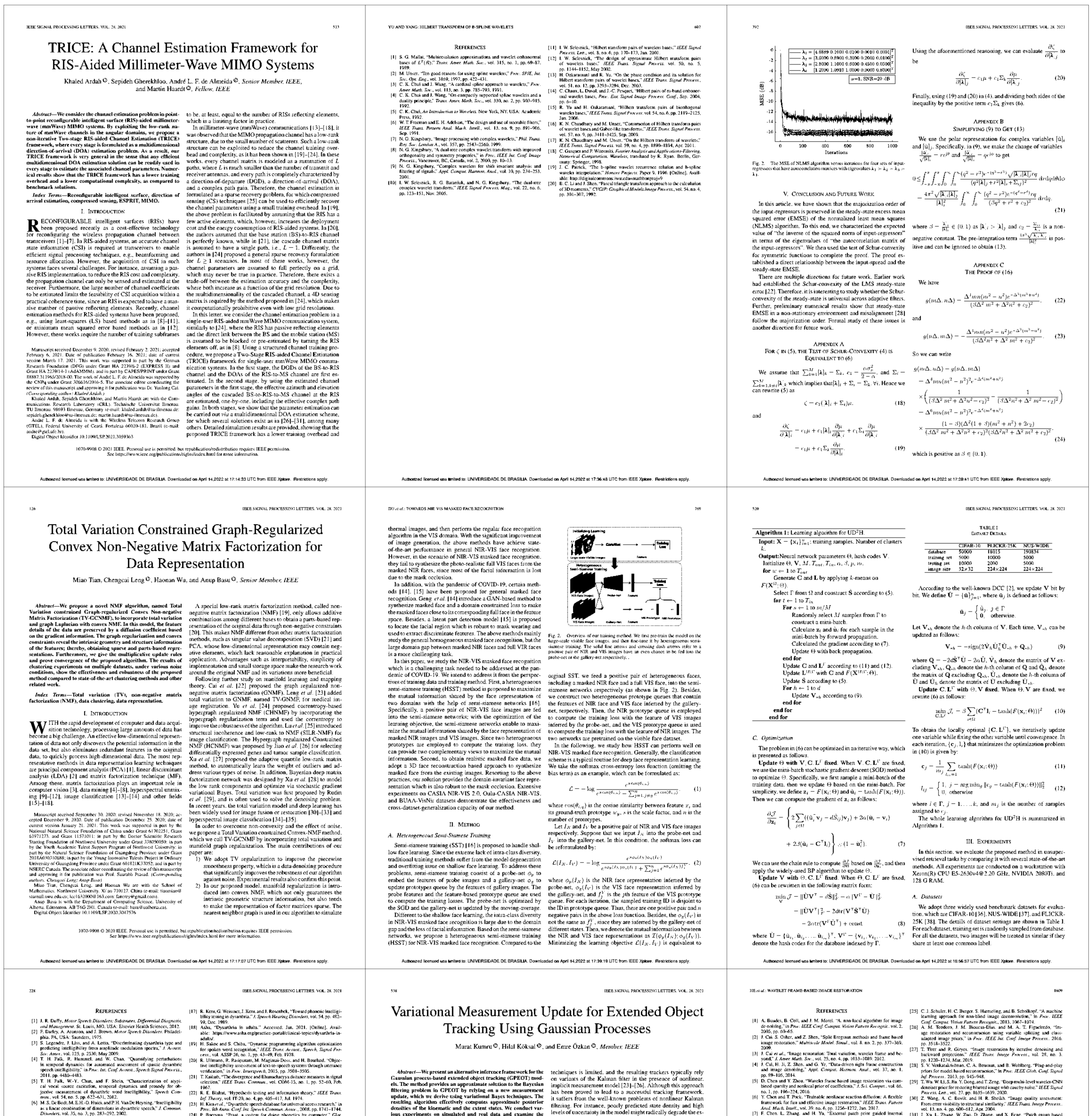


Figure I.4: Nine pages from the test set of Dataset 5, which are also present in the test set of Dataset 4.

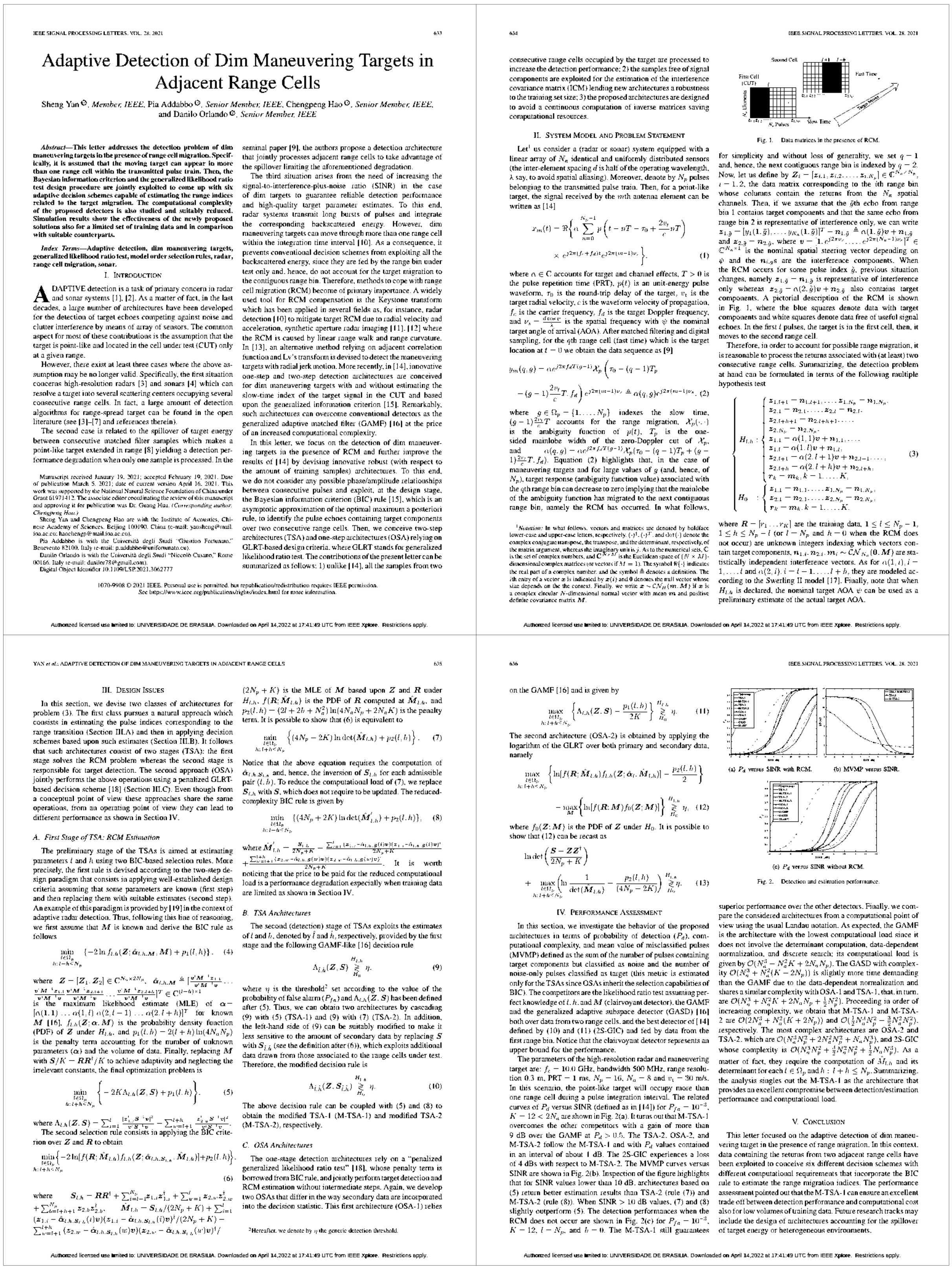


Figure I.5: First four pages from the test set of Dataset 6.

Annex II

GLCH Complementary Results (Lossless)

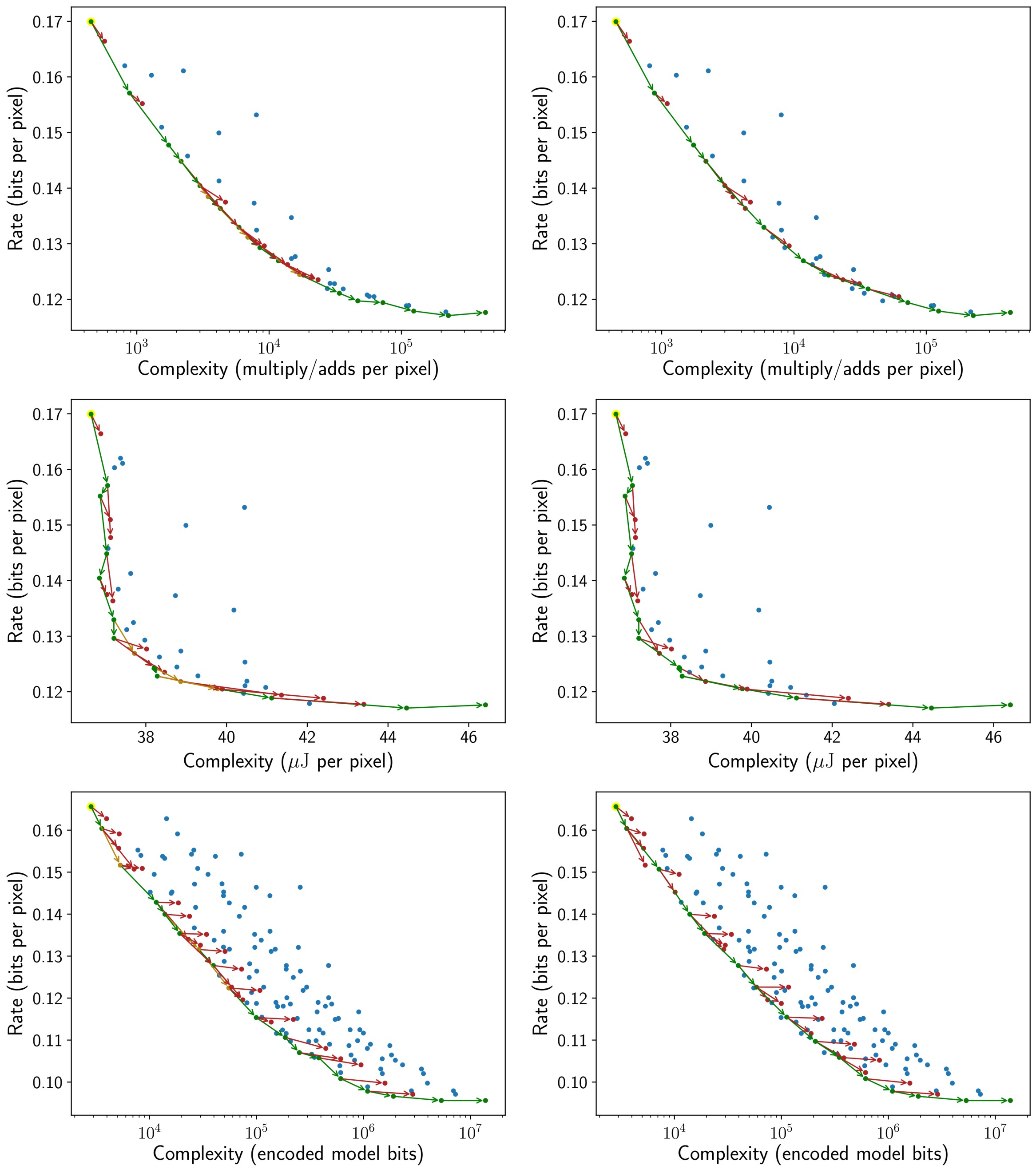


Figure II.1: Clouds of rate-complexity points with execution and final state of the GLCH algorithm for the select functions: Alg. 2 (left column) and Alg. 3 (right column). Bitrate is in bits per pixel and complexity is in multiply/add operations per pixel (top row), μJoules per pixel (middle row), and encoded model bits (bottom row).

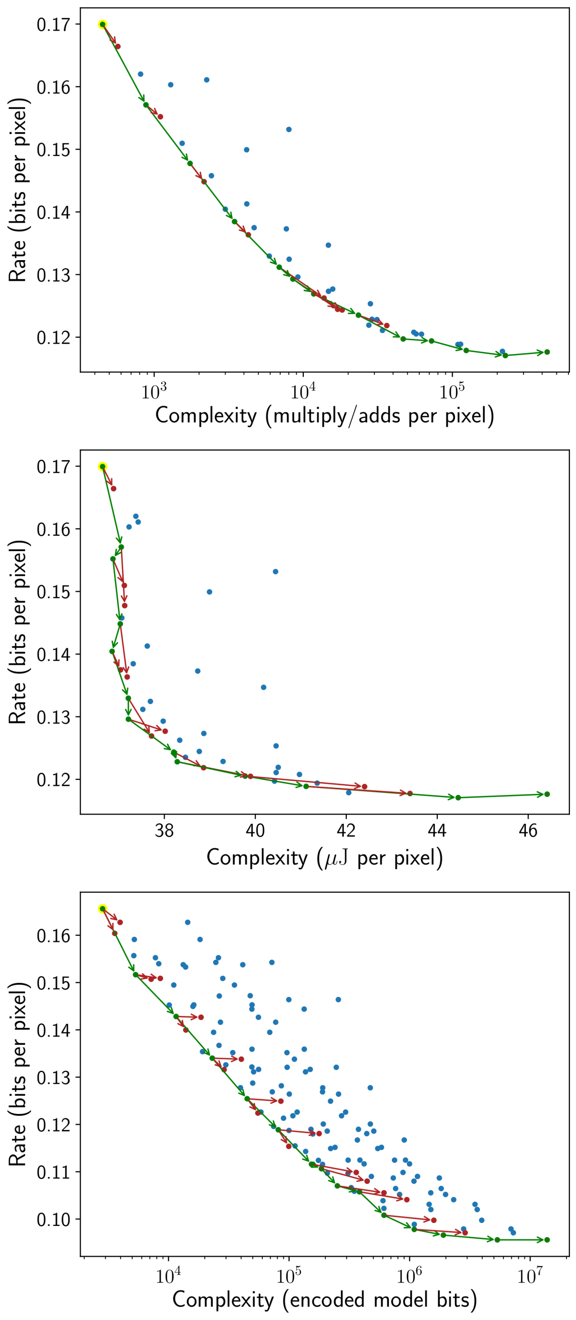


Figure II.2: Clouds of rate-complexity points with execution and final state of the GLCH algorithm for the select function Alg. 4. Bitrate is in bits per pixel and complexity is in multiply/add operations per pixel (top row), μJoules per pixel (middle row), and encoded model bits (bottom row).

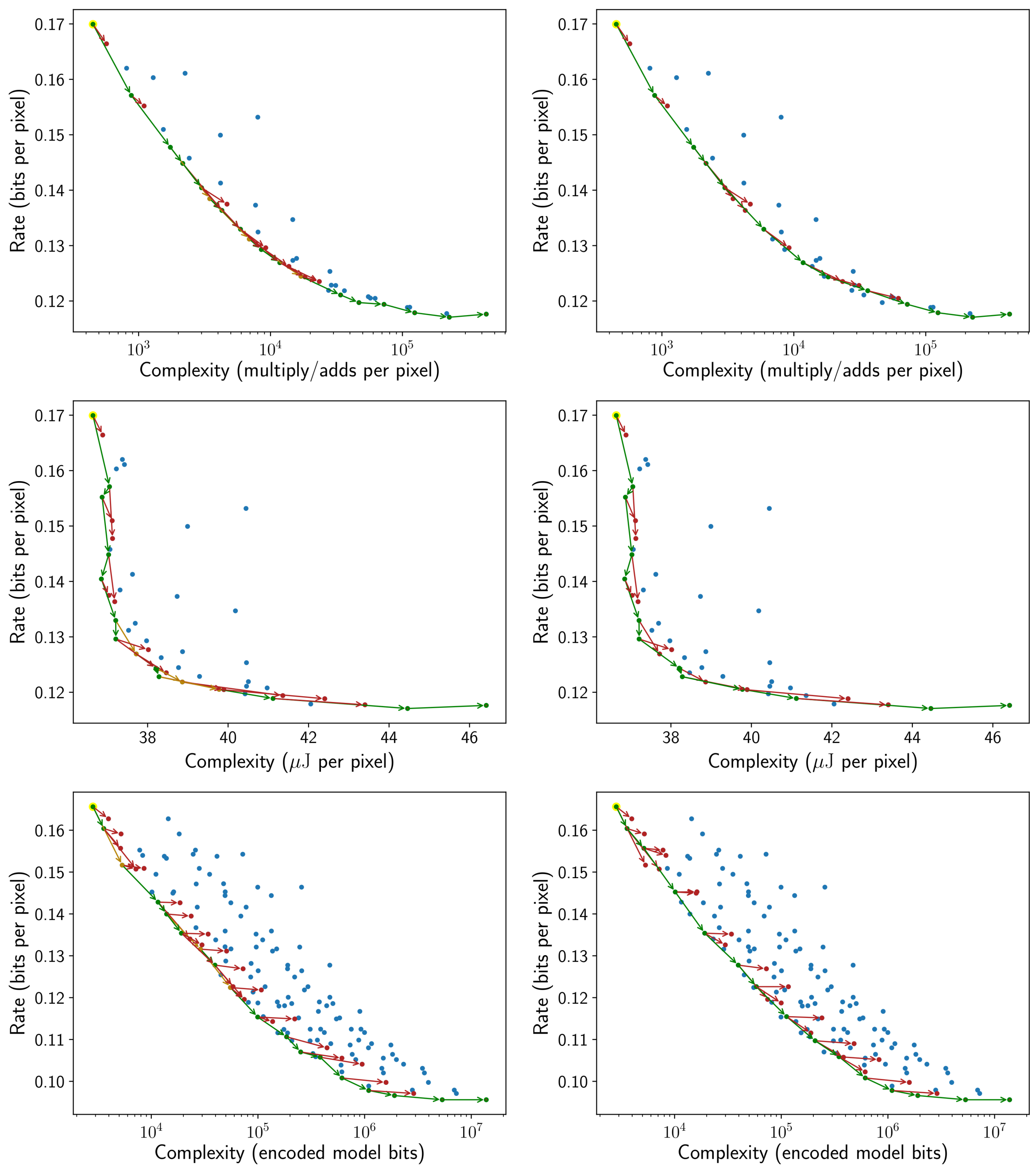


Figure II.3: Clouds of rate-complexity points with execution and final state of the GLCH algorithm for the select functions in Alg. 5 (left column) and Alg. 6 (right column). Bitrate is in bits per pixel and complexity is in multiply/add operations per pixel (top row), μJoules per pixel (middle row), and encoded model bits (bottom row).

Table II.1: Final trees of the GLCH algorithm with the select functions Alg. 2 (a,c,e) and Alg. 3 (b,d,f) for the different complexity measures: MAC/pixel (a,b), μJ/pixel (c,d) and encoded model bits (e,f). Each node is represented by its hyperparameters in the order: units in the first hidden layer, units in the second hidden layer and quantization bits, if applicable (continued on the next pages).

(a)		
parent	child 1	child 2
10,10	**20,10**	10,20
20,10	**40,10**	20,20
40,10	80,10	**40,20**
40,20	80,20	**40,40**
40,40	80,40	40,80
80,10	160,10	**80,20**
80,20	160,20	**80,40**
80,40	160,40	80,80
160,10	320,10	**160,20**
160,20	320,20	**160,40**
160,40	320,40	160,80
320,20	**640,20**	320,40
640,20		**640,40**
640,40		**640,80**
640,80		**640,160**
640,160		**640,320**
640,320		**640,640**

(b)		
parent	child 1	child 2
10,10	**20,10**	10,20
20,10	**40,10**	20,20
40,10	80,10	**40,20**
40,20	80,20	**40,40**
40,40	**80,40**	40,80
80,40	**160,40**	80,80
160,40	320,40	**160,80**
160,80	**320,80**	160,160
320,80	**640,80**	320,160
640,80		**640,160**
640,160		**640,320**
640,320		**640,640**

(c)		
parent	child 1	child 2
10,10	**20,10**	10,20
20,10	40,10	**20,20**
20,20	**40,20**	20,40
40,20	80,20	**40,40**
40,40	**80,40**	40,80
80,40	160,40	**80,80**
80,80	160,80	80,160
160,40	320,40	**160,80**
160,80	320,80	**160,160**
160,160	320,160	160,320
320,80	640,80	320,160
160,320	**320,320**	160,640
320,320	**640,320**	320,640
640,320		**640,640**

(d)		
parent	child 1	child 2
10,10	**20,10**	10,20
20,10	40,10	**20,20**
20,20	**40,20**	20,40
40,20	80,20	**40,40**
40,40	**80,40**	40,80
80,40	160,40	**80,80**
80,80	**160,80**	80,160
160,80	320,80	**160,160**
160,160	320,160	**160,320**
160,320	**320,320**	160,640
320,320	**640,320**	320,640
640,320		**640,640**

(e)			
parent	child 1	child 2	child 3
10,10,8	20,10,8	**10,20,8**	10,10,16
10,20,8	20,20,8	10,40,8	10,20,16
20,10,8	**40,10,8**	20,20,8	20,10,16
40,10,8	80,10,8	**40,20,8**	40,10,16
40,20,8	80,20,8	**40,40,8**	40,20,16
40,40,8	80,40,8	40,80,8	40,40,16
80,20,8	160,20,8	**80,40,8**	80,20,16
80,40,8	160,40,8	80,80,8	80,40,16
160,20,8	**320,20,8**	160,40,8	160,20,16
320,20,8	**640,20,8**	320,40,8	320,20,16
640,20,8		**640,40,8**	640,20,16
640,40,8		**640,80,8**	640,40,16
640,80,8		**640,160,8**	640,80,16
640,160,8		**640,320,8**	640,160,16
640,320,8		**640,640,8**	640,320,16
640,640,8			**640,640,16**
640,640,16			**640,640,32**

(f)			
parent	child 1	child 2	child 3
10,10,8	20,10,8	**10,20,8**	10,10,16
10,20,8	**20,20,8**	10,40,8	10,20,16
20,20,8	**40,20,8**	20,40,8	20,20,16
40,20,8	80,20,8	**40,40,8**	40,20,16
40,40,8	**80,40,8**	40,80,8	40,40,16
80,40,8	160,40,8	**80,80,8**	80,40,16
80,80,8	**160,80,8**	80,160,8	80,80,16
160,80,8	**320,80,8**	160,160,8	160,80,16
320,80,8	640,80,8	**320,160,8**	320,80,16
320,160,8	**640,160,8**	320,320,8	320,160,16
640,160,8		**640,320,8**	640,160,16
640,320,8		**640,640,8**	640,320,16
640,640,8			**640,640,16**
640,640,16			**640,640,32**

Table II.2: Final trees of the GLCH algorithm with the select function Alg. 4 for the different complexity measures: (a) MAC/pixel, (b) μJ/pixel and (c) encoded model bits. Each node is represented by its hyperparameters in the order: units in the first hidden layer, units in the second hidden layer and quantization bits, if applicable.

(a)		
parent	child 1	child 2
10,10	**20,10**	10,20
20,10	**40,10**	20,20
40,10	**80,10**	40,20
80,10	**160,10**	80,20
160,10	320,10	**160,20**
160,20	320,20	**160,40**
160,40	**320,40**	160,80
320,40	**640,40**	320,80
640,40		**640,80**
640,80		**640,160**
640,160		**640,320**
640,320		**640,640**

(b)		
parent	child 1	child 2
10,10	**20,10**	10,20
20,10	40,10	**20,20**
20,20	**40,20**	20,40
40,20	80,20	**40,40**
40,40	**80,40**	40,80
80,40	160,40	**80,80**
80,80	**160,80**	80,160
160,80	320,80	**160,160**
160,160	320,160	**160,320**
160,320	**320,320**	160,640
320,320	**640,320**	320,640
640,320		**640,640**

(c)			
parent	child 1	child 2	child 3
10,10,8	**20,10,8**	10,20,8	10,10,16
20,10,8	**40,10,8**	20,20,8	20,10,16
40,10,8	**80,10,8**	40,20,8	40,10,16
80,10,8	**160,10,8**	80,20,8	80,10,16
160,10,8	**320,10,8**	160,20,8	160,10,16
320,10,8	**640,10,8**	320,20,8	320,10,16
640,10,8		**640,20,8**	640,10,16
640,20,8		**640,40,8**	640,20,16
640,40,8		**640,80,8**	640,40,16
640,80,8		**640,160,8**	640,80,16
640,160,8		**640,320,8**	640,160,16
640,320,8		**640,640,8**	640,320,16
640,640,8			**640,640,16**
640,640,16			**640,640,32**

Table II.3: Final trees of the GLCH algorithm with the select functions Alg. 5 (a,c,e) and Alg. 6 (b,d,f) for the different complexity measures: MAC/pixel (a,b), μJ/pixel (c,d) and encoded model bits (e,f). Each node is represented by its hyperparameters in the order: units in the first hidden layer, units in the second hidden layer and quantization bits, if applicable (continued on the next pages).

(a)		
parent	child 1	child 2
10,10	**20,10**	10,20
20,10	**40,10**	20,20
40,10	80,10	**40,20**
40,20	80,20	**40,40**
40,40	80,40	40,80
80,10	160,10	**80,20**
80,20	160,20	**80,40**
80,40	160,40	80,80
160,10	320,10	**160,20**
160,20	320,20	**160,40**
160,40	320,40	160,80
320,20	**640,20**	320,40
640,20		**640,40**
640,40		**640,80**
640,80		**640,160**
640,160		**640,320**
640,320		**640,640**

(b)		
parent	child 1	child 2
10,10	**20,10**	10,20
20,10	**40,10**	20,20
40,10	80,10	**40,20**
40,20	80,20	**40,40**
40,40	**80,40**	40,80
80,40	**160,40**	80,80
160,40	320,40	**160,80**
160,80	**320,80**	160,160
320,80	**640,80**	320,160
640,80		**640,160**
640,160		**640,320**
640,320		**640,640**

(c)		
parent	child 1	child 2
10,10	**20,10**	10,20
20,10	40,10	**20,20**
20,20	**40,20**	20,40
40,20	80,20	**40,40**
40,40	**80,40**	40,80
80,40	160,40	**80,80**
80,80	160,80	80,160
160,40	320,40	**160,80**
160,80	320,80	**160,160**
160,160	320,160	160,320
320,80	640,80	320,160
160,320	**320,320**	160,640
320,320	**640,320**	320,640
640,320		**640,640**

(d)		
parent	child 1	child 2
10,10	**20,10**	10,20
20,10	40,10	**20,20**
20,20	**40,20**	20,40
40,20	80,20	**40,40**
40,40	**80,40**	40,80
80,40	160,40	**80,80**
80,80	**160,80**	80,160
160,80	320,80	**160,160**
160,160	320,160	**160,320**
160,320	**320,320**	160,640
320,320	**640,320**	320,640
640,320		**640,640**

(e)			
parent	child 1	child 2	child 3
10,10,8	20,10,8	**10,20,8**	10,10,16
10,20,8	20,20,8	10,40,8	10,20,16
20,10,8	**40,10,8**	20,20,8	20,10,16
40,10,8	80,10,8	**40,20,8**	40,10,16
40,20,8	80,20,8	**40,40,8**	40,20,16
40,40,8	80,40,8	40,80,8	40,40,16
80,20,8	160,20,8	**80,40,8**	80,20,16
80,40,8	160,40,8	80,80,8	80,40,16
160,20,8	**320,20,8**	160,40,8	160,20,16
320,20,8	**640,20,8**	320,40,8	320,20,16
640,20,8		**640,40,8**	640,20,16
640,40,8		**640,80,8**	640,40,16
640,80,8		**640,160,8**	640,80,16
640,160,8		**640,320,8**	640,160,16
640,320,8		**640,640,8**	640,320,16
640,640,8			**640,640,16**
640,640,16			**640,640,32**

(f)			
parent	child 1	child 2	child 3
10,10,8	20,10,8	**10,20,8**	10,10,16
10,20,8	20,20,8	**10,40,8**	10,20,16
10,40,8	**20,40,8**	10,80,8	10,40,16
20,40,8	**40,40,8**	20,80,8	20,40,16
40,40,8	**80,40,8**	40,80,8	40,40,16
80,40,8	160,40,8	**80,80,8**	80,40,16
80,80,8	**160,80,8**	80,160,8	80,80,16
160,80,8	**320,80,8**	160,160,8	160,80,16
320,80,8	640,80,8	**320,160,8**	320,80,16
320,160,8	**640,160,8**	320,320,8	320,160,16
640,160,8		**640,320,8**	640,160,16
640,320,8		**640,640,8**	640,320,16
640,640,8			**640,640,16**
640,640,16			**640,640,32**

Annex III

GLCH Complementary Results (Lossy)

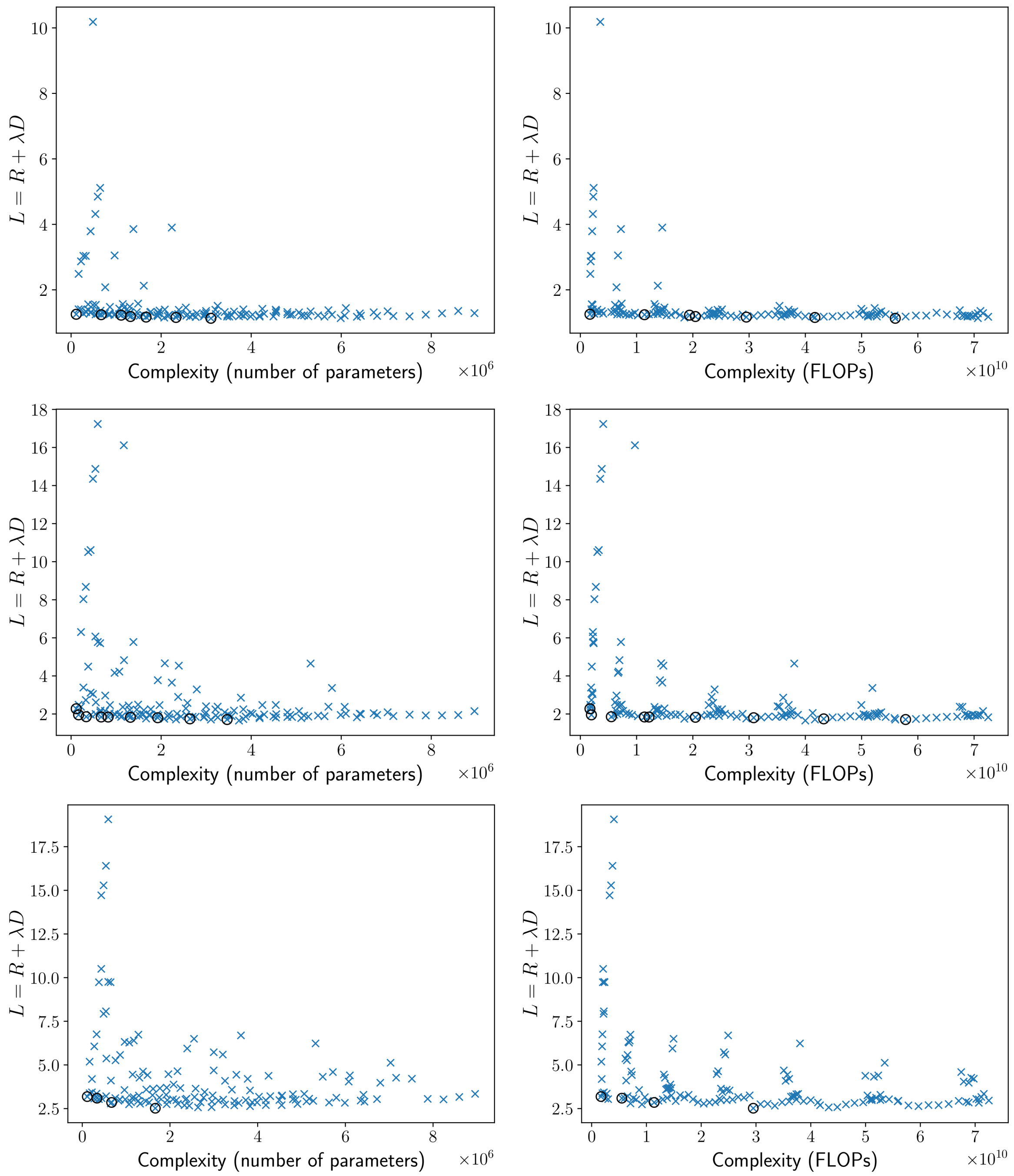


Figure III.1: Clouds of loss-complexity points, with emphasis on the parent nodes found by the GLCH algorithm, using the select function in Algorithm 2. The loss is equal to $R + \lambda D$, where λ is equal to $255^2 \times 0.005$ (top row), $255^2 \times 0.01$ (middle row) and $255^2 \times 0.02$ (bottom row), R is the rate in bits per pixel, and D is the distortion in mean squared error, with the pixel values normalized between 0 and 1. Complexity is given in number of parameters (left column), and in number of FLOPs (right column).

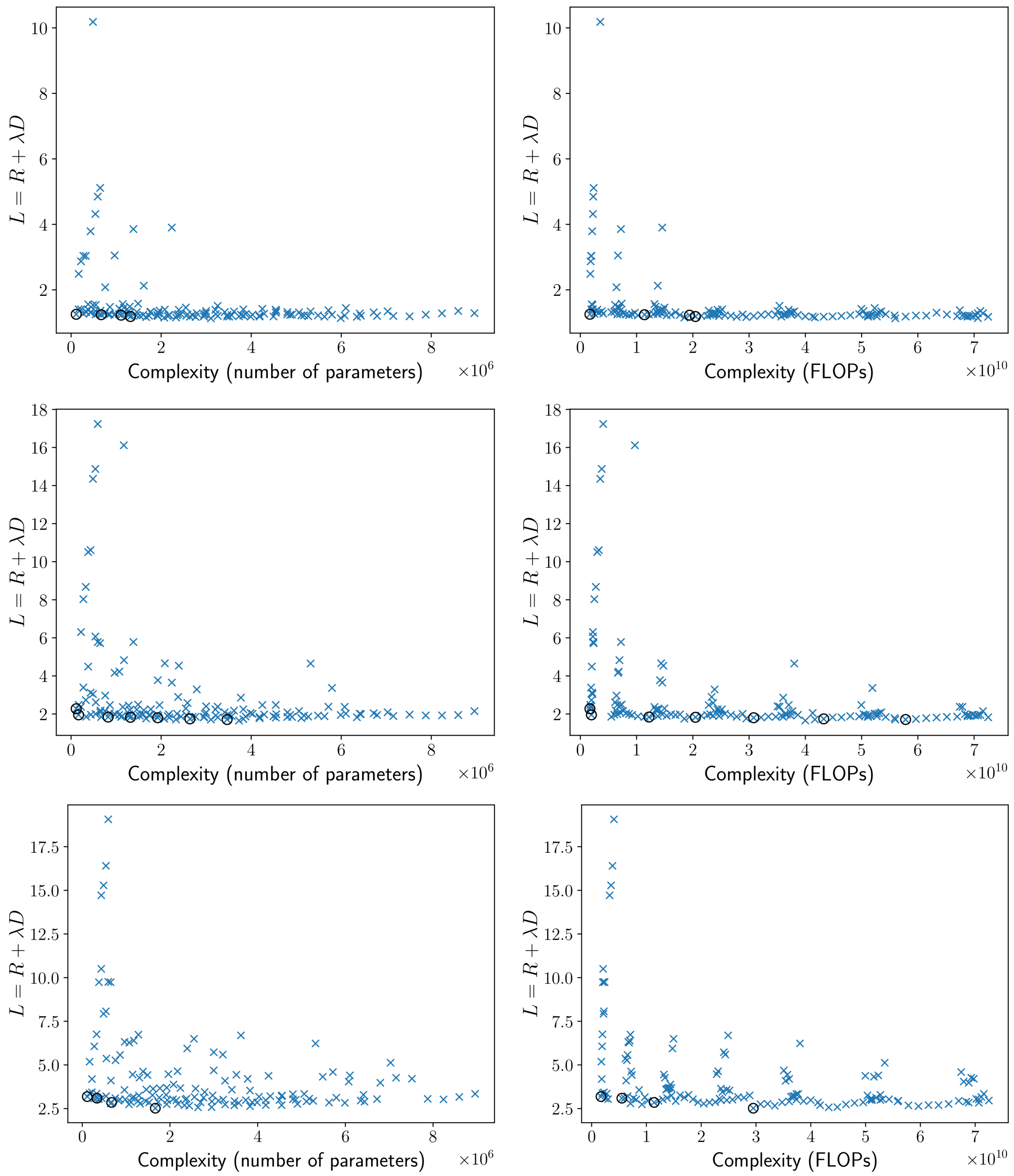


Figure III.2: Clouds of loss-complexity points, with emphasis on the parent nodes found by the GLCH algorithm, using the select function in Algorithm 3. The loss is equal to $R + \lambda D$, where λ is equal to $255^2 \times 0.005$ (top row), $255^2 \times 0.01$ (middle row) and $255^2 \times 0.02$ (bottom row), R is the rate in bits per pixel, and D is the distortion in mean squared error, with the pixel values normalized between 0 and 1. Complexity is given in number of parameters (left column), and in number of FLOPs (right column).

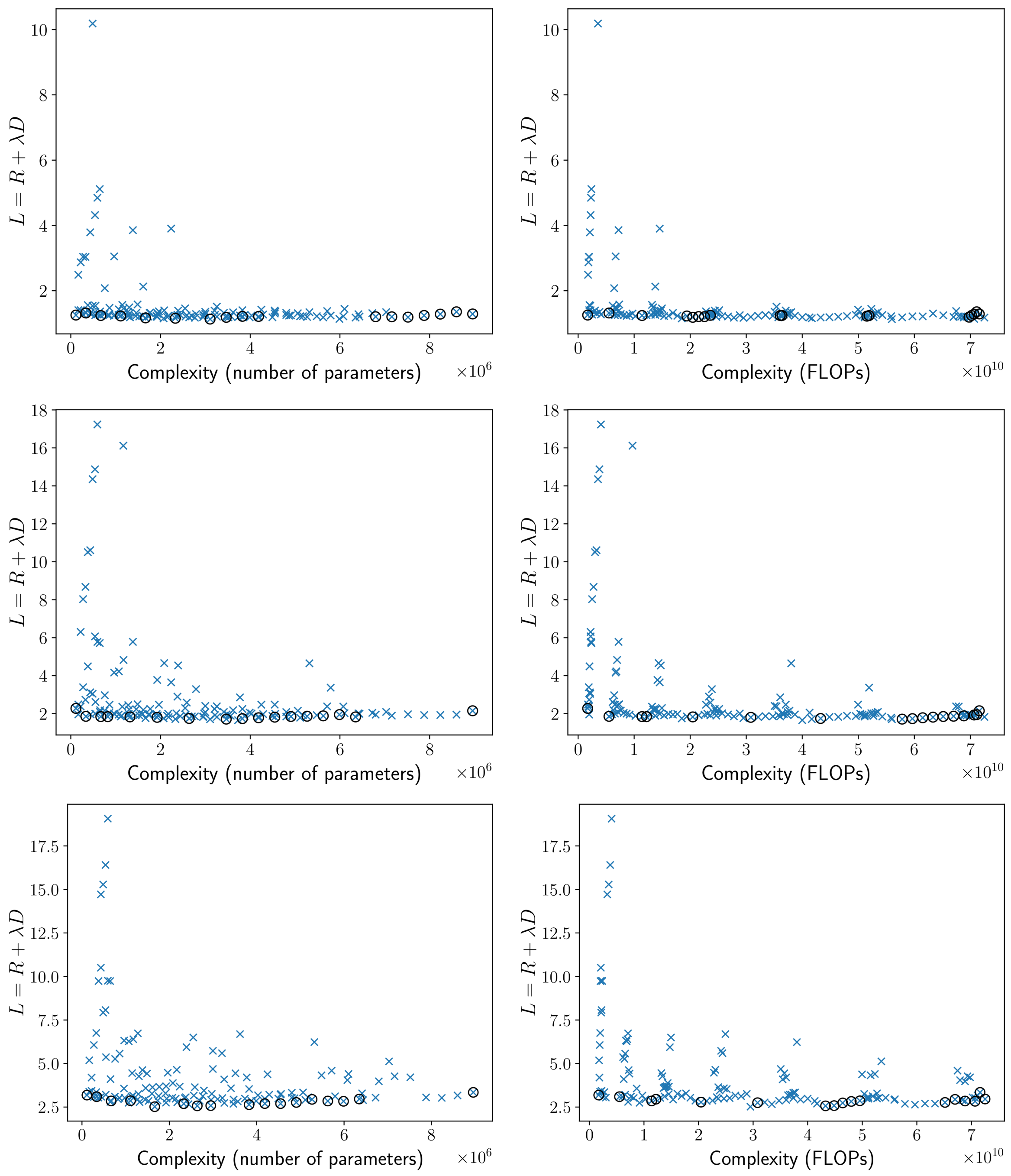


Figure III.3: Clouds of loss-complexity points, with emphasis on the parent nodes found by the GLCH algorithm, using the select function in Algorithm 4. The loss is equal to $R + \lambda D$, where λ is equal to $255^2 \times 0.005$ (top row), $255^2 \times 0.01$ (middle row) and $255^2 \times 0.02$ (bottom row), R is the rate in bits per pixel, and D is the distortion in mean squared error, with the pixel values normalized between 0 and 1. Complexity is given in number of parameters (left column), and in number of FLOPs (right column).

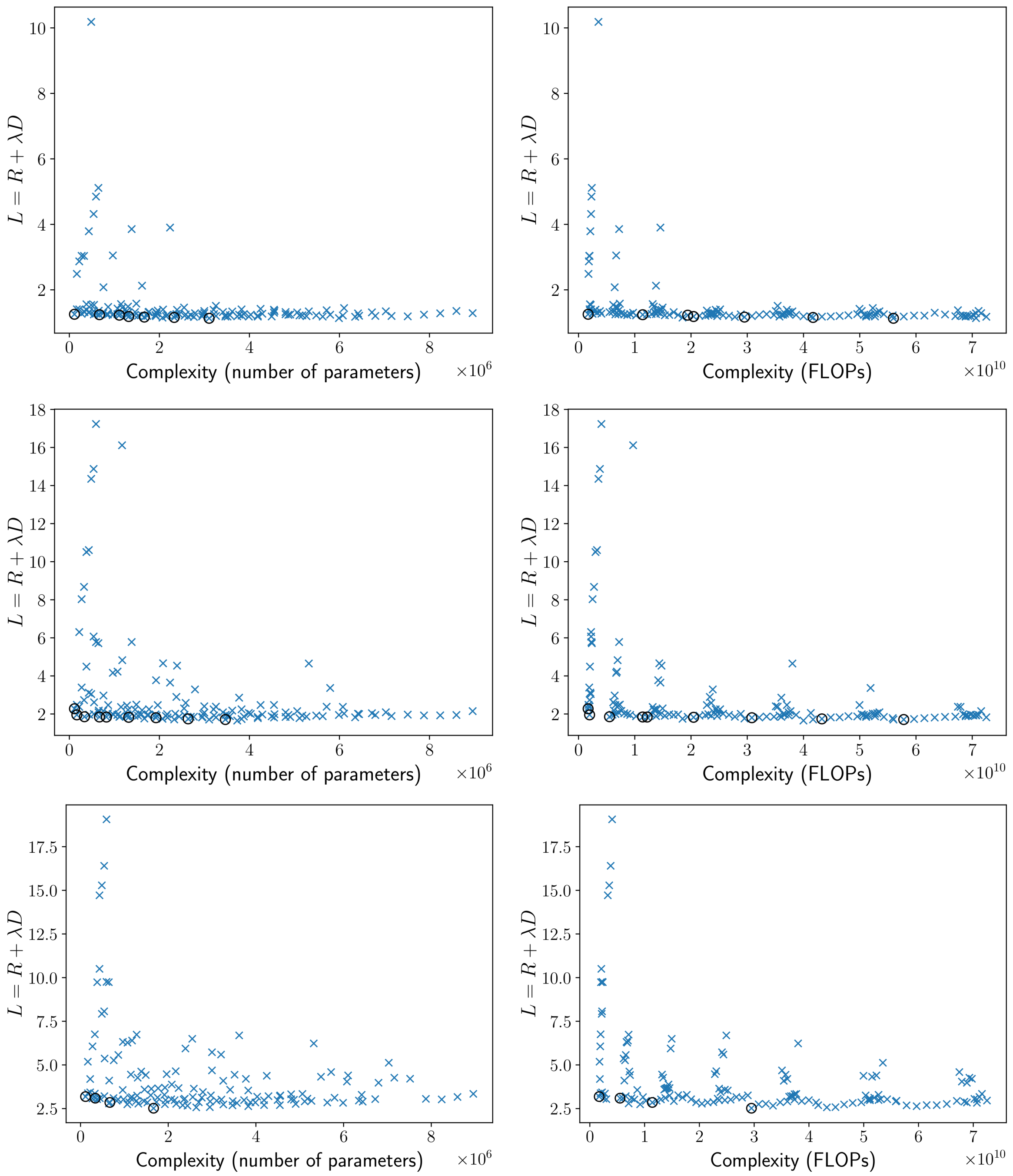


Figure III.4: Clouds of loss-complexity points, with emphasis on the parent nodes found by the GLCH algorithm, using the select function in Algorithm 5. The loss is equal to $R + \lambda D$, where λ is equal to $255^2 \times 0.005$ (top row), $255^2 \times 0.01$ (middle row) and $255^2 \times 0.02$ (bottom row), R is the rate in bits per pixel, and D is the distortion in mean squared error, with the pixel values normalized between 0 and 1. Complexity is given in number of parameters (left column), and in number of FLOPs (right column).

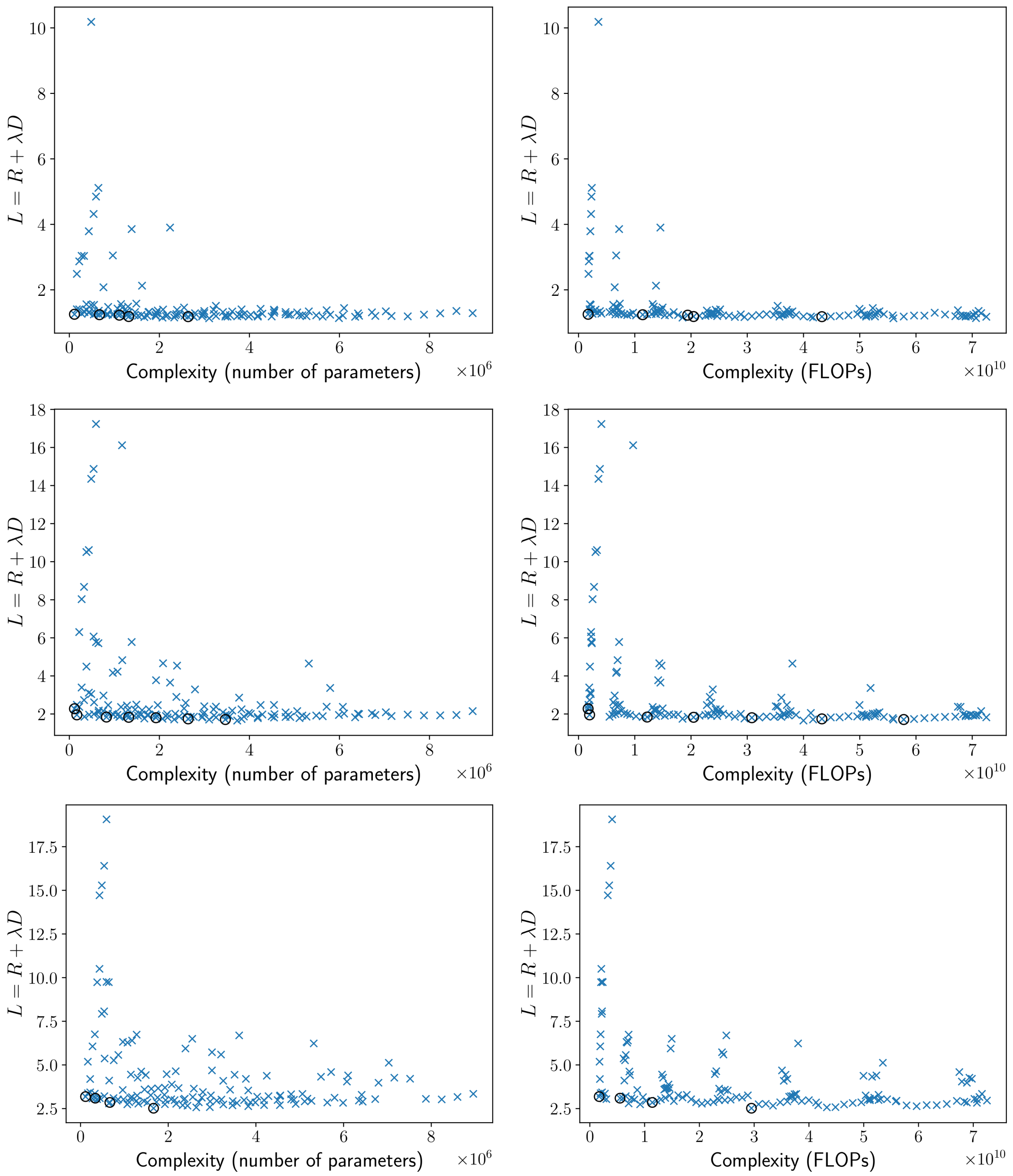


Figure III.5: Clouds of loss-complexity points, with emphasis on the parent nodes found by the GLCH algorithm, using the select function in Algorithm 6. The loss is equal to $R + \lambda D$, where λ is equal to $255^2 \times 0.005$ (top row), $255^2 \times 0.01$ (middle row) and $255^2 \times 0.02$ (bottom row), R is the rate in bits per pixel, and D is the distortion in mean squared error, with the pixel values normalized between 0 and 1. Complexity is given in number of parameters (left column), and in number of FLOPs (right column).

Table III.1: Points of the loss-complexity clouds that were parent nodes during the execution of the GLCH algorithm with the select function Alg. 2 for $\lambda' = 0.005$ (top row), 0.01 (middle row) and 0.02 (bottom row) and for complexity measured as number of parameters (left column), and number of FLOPs (right column). Each node is represented by its hyperparameters in the order: number of layers of the encoder and the decoder, number of filters of all layers except the last of the encoder, and number of filters of the last layer of the encoder.

(a)

node	10^6 parameters	loss
3,32,32	0.114	1.254
3,96,32	0.668	1.241
3,128,32	1.112	1.226
3,128,64	1.319	1.185
3,160,32	1.666	1.168
3,192,32	2.330	1.157
3,224,32	3.105	1.131

(b)

node	10^{10} FLOPs	loss
3,32,32	0.171	1.254
3,96,32	1.141	1.241
3,128,32	1.941	1.226
3,128,64	2.046	1.185
3,160,32	2.950	1.168
3,192,32	4.170	1.157
3,224,32	5.599	1.131

(c)

node	10^6 parameters	loss
3,32,32	0.114	2.274
3,32,64	0.167	1.953
3,64,32	0.336	1.855
3,96,32	0.668	1.842
3,96,64	0.824	1.838
3,128,64	1.319	1.825
3,160,64	1.924	1.803
3,192,64	2.639	1.742
3,224,64	3.466	1.709

(d)

node	10^{10} FLOPs	loss
3,32,32	0.171	2.274
3,32,64	0.197	1.953
3,64,32	0.551	1.855
3,96,32	1.141	1.842
3,96,64	1.220	1.838
3,128,64	2.046	1.825
3,160,64	3.082	1.803
3,192,64	4.327	1.742
3,224,64	5.782	1.709

(e)

node	10^6 parameters	loss
3,32,32	0.114	3.186
3,64,32	0.336	3.094
3,96,32	0.668	2.851
3,160,32	1.666	2.520

(f)

node	10^{10} FLOPs	loss
3,32,32	0.171	3.186
3,64,32	0.551	3.094
3,96,32	1.141	2.851
3,160,32	2.950	2.520

Table III.2: Points of the loss-complexity clouds that were parent nodes during the execution of the GLCH algorithm with the select function Alg. 3 for $\lambda' = 0.005$ (top row), 0.01 (middle row) and 0.02 (bottom row) and for complexity measured as number of parameters (left column), and number of FLOPs (right column). Each node is represented by its hyperparameters in the order: number of layers of the encoder and the decoder, number of filters of all layers except the last of the encoder, and number of filters of the last layer of the encoder.

(a)		
node	10^6 parameters	loss
3,32,32	0.114	1.254
3,96,32	0.668	1.241
3,128,32	1.112	1.226
3,128,64	1.319	1.185

(b)		
node	10^{10} FLOPs	loss
3,32,32	0.171	1.254
3,96,32	1.141	1.241
3,128,32	1.941	1.226
3,128,64	2.046	1.185

(c)		
node	10^6 parameters	loss
3,32,32	0.114	2.274
3,32,64	0.167	1.953
3,96,64	0.824	1.838
3,128,64	1.319	1.825
3,160,64	1.924	1.803
3,192,64	2.639	1.742
3,224,64	3.466	1.709

(d)		
node	10^{10} FLOPs	loss
3,32,32	0.171	2.274
3,32,64	0.197	1.953
3,96,64	1.220	1.838
3,128,64	2.046	1.825
3,160,64	3.082	1.803
3,192,64	4.327	1.742
3,224,64	5.782	1.709

(e)		
node	10^6 parameters	loss
3,32,32	0.114	3.186
3,64,32	0.336	3.094
3,96,32	0.668	2.851
3,160,32	1.666	2.520

(f)		
node	10^{10} FLOPs	loss
3,32,32	0.171	3.186
3,64,32	0.551	3.094
3,96,32	1.141	2.851
3,160,32	2.950	2.520

Table III.3: Points of the loss-complexity clouds that were parent nodes during the execution of the GLCH algorithm with the select function Alg. 4 for $\lambda' = 0.005$ (top row), 0.01 (middle row) and 0.02 (bottom row) and for complexity measured as number of parameters (left column), and number of FLOPs (right column). Each node is represented by its hyperparameters in the order: number of layers of the encoder and the decoder, number of filters of all layers except the last of the encoder, and number of filters of the last layer of the encoder (continued on the next pages).

(a)		
node	10^6 parameters	loss
3,32,32	0.114	1.254
3,64,32	0.336	1.318
3,96,32	0.668	1.241
3,128,32	1.112	1.226
3,160,32	1.666	1.168
3,192,32	2.330	1.157
3,224,32	3.105	1.131
3,224,64	3.466	1.184
3,224,96	3.826	1.215
3,224,128	4.186	1.213
4,224,128	6.796	1.204
4,224,160	7.157	1.202
4,224,192	7.517	1.190
4,224,224	7.878	1.242
4,224,256	8.238	1.283
4,224,288	8.598	1.355
4,224,320	8.959	1.288

(b)		
node	10^{10} FLOPs	loss
3,32,32	0.171	1.254
3,64,32	0.551	1.318
3,96,32	1.141	1.241
3,128,32	1.941	1.226
3,128,64	2.046	1.185
3,128,96	2.151	1.208
3,128,128	2.256	1.206
4,128,128	2.360	1.247
4,160,128	3.606	1.241
4,160,160	3.639	1.241
4,192,160	5.153	1.221
4,192,192	5.192	1.240
4,224,192	6.975	1.190
4,224,224	7.021	1.242
4,224,256	7.067	1.283
4,224,288	7.113	1.355
4,224,320	7.159	1.288

(c)		
node	10^6 parameters	loss
3,32,32	0.114	2.274
3,64,32	0.336	1.855
3,96,32	0.668	1.842
3,96,64	0.824	1.838
3,128,64	1.319	1.825
3,160,64	1.924	1.803
3,192,64	2.639	1.742
3,224,64	3.466	1.709
3,224,96	3.826	1.735
3,224,128	4.186	1.780
3,224,160	4.547	1.812
3,224,192	4.907	1.847
3,224,224	5.268	1.860
3,224,256	5.628	1.881
3,224,288	5.988	1.947
3,224,320	6.349	1.825
4,224,320	8.959	2.150

(d)		
node	10^{10} FLOPs	loss
3,32,32	0.171	2.274
3,64,32	0.551	1.855
3,96,32	1.141	1.842
3,96,64	1.220	1.838
3,128,64	2.046	1.825
3,160,64	3.082	1.803
3,192,64	4.327	1.742
3,224,64	5.782	1.709
3,224,96	5.966	1.735
3,224,128	6.149	1.780
3,224,160	6.333	1.812
3,224,192	6.516	1.847
3,224,224	6.700	1.860
3,224,256	6.883	1.881
4,224,256	7.067	1.927
4,224,288	7.113	1.946
4,224,320	7.159	2.150

(e)		
node	10^6 parameters	loss
3,32,32	0.114	3.186
3,64,32	0.336	3.094
3,96,32	0.668	2.851
3,128,32	1.112	2.862
3,160,32	1.666	2.520
3,192,32	2.330	2.691
3,192,64	2.639	2.568
3,192,96	2.949	2.582
3,224,96	3.826	2.636
3,224,128	4.186	2.699
3,224,160	4.547	2.696
3,224,192	4.907	2.764
3,224,224	5.268	2.941
3,224,256	5.628	2.847
3,224,288	5.988	2.829
3,224,320	6.349	2.960
4,224,320	8.959	3.343

(f)		
node	10^{10} FLOPs	loss
3,32,32	0.171	3.186
3,64,32	0.551	3.094
3,96,32	1.141	2.851
3,96,64	1.220	2.951
3,128,64	2.046	2.780
3,160,64	3.082	2.749
3,192,64	4.327	2.568
3,192,96	4.484	2.582
3,192,128	4.642	2.736
3,192,160	4.799	2.821
3,192,192	4.956	2.859
3,224,192	6.516	2.764
3,224,224	6.700	2.941
3,224,256	6.883	2.847
3,224,288	7.067	2.829
3,224,320	7.250	2.960
4,224,320	7.159	3.343

Table III.4: Points of the loss-complexity clouds that were parent nodes during the execution of the GLCH algorithm with the select function Alg. 5 for $\lambda' = 0.005$ (top row), 0.01 (middle row) and 0.02 (bottom row) and for complexity measured as number of parameters (left column), and number of FLOPs (right column). Each node is represented by its hyperparameters in the order: number of layers of the encoder and the decoder, number of filters of all layers except the last of the encoder, and number of filters of the last layer of the encoder.

(a)

node	10^6 parameters	loss
3,32,32	0.114	1.254
3,96,32	0.668	1.241
3,128,32	1.112	1.226
3,128,64	1.319	1.185
3,160,32	1.666	1.168
3,192,32	2.330	1.157
3,224,32	3.105	1.131

(b)

node	10^{10} FLOPs	loss
3,32,32	0.171	1.254
3,96,32	1.141	1.241
3,128,32	1.941	1.226
3,128,64	2.046	1.185
3,160,32	2.950	1.168
3,192,32	4.170	1.157
3,224,32	5.599	1.131

(c)

node	10^6 parameters	loss
3,32,32	0.114	2.274
3,32,64	0.167	1.953
3,64,32	0.336	1.855
3,96,32	0.668	1.842
3,96,64	0.824	1.838
3,128,64	1.319	1.825
3,160,64	1.924	1.803
3,192,64	2.639	1.742
3,224,64	3.466	1.709

(d)

node	10^{10} FLOPs	loss
3,32,32	0.171	2.274
3,32,64	0.197	1.953
3,64,32	0.551	1.855
3,96,32	1.141	1.842
3,96,64	1.220	1.838
3,128,64	2.046	1.825
3,160,64	3.082	1.803
3,192,64	4.327	1.742
3,224,64	5.782	1.709

(e)

node	10^6 parameters	loss
3,32,32	0.114	3.186
3,64,32	0.336	3.094
3,96,32	0.668	2.851
3,160,32	1.666	2.520

(f)

node	10^{10} FLOPs	loss
3,32,32	0.171	3.186
3,64,32	0.551	3.094
3,96,32	1.141	2.851
3,160,32	2.950	2.520

Table III.5: Points of the loss-complexity clouds that were parent nodes during the execution of the GLCH algorithm with the select function Alg. 6 for $\lambda' = 0.005$ (top row), 0.01 (middle row) and 0.02 (bottom row) and for complexity measured as number of parameters (left column), and number of FLOPs (right column). Each node is represented by its hyperparameters in the order: number of layers of the encoder and the decoder, number of filters of all layers except the last of the encoder, and number of filters of the last layer of the encoder.

(a)		
node	10^6 parameters	loss
3,32,32	0.114	1.254
3,96,32	0.668	1.241
3,128,32	1.112	1.226
3,128,64	1.319	1.185
3,192,64	2.639	1.181

(b)		
node	10^{10} FLOPs	loss
3,32,32	0.171	1.254
3,96,32	1.141	1.241
3,128,32	1.941	1.226
3,128,64	2.046	1.185
3,192,64	4.327	1.181

(c)		
node	10^6 parameters	loss
3,32,32	0.114	2.274
3,32,64	0.167	1.953
3,96,64	0.824	1.838
3,128,64	1.319	1.825
3,160,64	1.924	1.803
3,192,64	2.639	1.742
3,224,64	3.466	1.709

(d)		
node	10^{10} FLOPs	loss
3,32,32	0.171	2.274
3,32,64	0.197	1.953
3,96,64	1.220	1.838
3,128,64	2.046	1.825
3,160,64	3.082	1.803
3,192,64	4.327	1.742
3,224,64	5.782	1.709

(e)		
node	10^6 parameters	loss
3,32,32	0.114	3.186
3,64,32	0.336	3.094
3,96,32	0.668	2.851
3,160,32	1.666	2.520

(f)		
node	10^{10} FLOPs	loss
3,32,32	0.171	3.186
3,64,32	0.551	3.094
3,96,32	1.141	2.851
3,160,32	2.950	2.520

Annex IV

Publications

1.	L. S. Lopes, P. A. Chou and R. L. de Queiroz, “Adaptive Context Modeling for Arithmetic Coding Using Perceptrons,” in *IEEE Signal Processing Letters*, vol. 29, pp. 2382-2386, 2022.
2.	V. F. Figueiredo, R. L. de Queiroz, P. A. Chou, L. S. Lopes “Embedded Coding of Point Cloud Attributes” in *IEEE Signal Processing Letters*, vol 31, pp. 890-893, 2024.
3.	L. S. Lopes, R. L. de Queiroz and P. A. Chou, “Rate-Complexity Optimization in Lossless Neural-based Image Compression,” accepted for publication at ICIP 2024.