Compression of Wi-Fi 802.11ax Data using Large Language Models

Matthew Russell Francis

500303593

A Thesis Submitted in Partial Requirement for Award of the Degree of Bachelor of Engineering Honours (Electrical Engineering) and Bachelor of Science.

School of Electrical and Computer Engineering
The University of Sydney

October 2024

Supervisor: Professor Philip H. W. Leong

Disclaimers

Statement of achievement

We contribute transformer based models which are able to compress MATLAB-generated WiFi 802.11ax `MCS0` signals at varying scales and noise-levels. We propose an algorithm for a scale-invariant compression scheme for use with an arithmetic coder. Further we train a custom 'head' appended to LLaMA 2 and compress with both the base model and the headed model.

Declaration of work

The work presented in this report is my own unless otherwise referenced or acknowledged. The ideas presented herein are the product of my own research in tandem with Professor Philip Leong. The code and concepts on which this work was based was created by DeepMind (Delétang et al. [20]) and can be found at https://github.com/google-deepmind/language_modeling_is_compression. A link to this works' codebase can be accessed at https://mfra.dev/thesis.

Declaration of AI

In line with the School of Electrical and Computer Engineering's policy, AI was not used for original content generation. AI was used to:

- improve the grammar and syntax, and make more concise originally written sentences;
- generate basic code-blocks for testing the proposed algorithms in this work; and
- help in the generation and formatting of `matplotlib` and Ti*k*Z figures, and the particulars of some LaTeX formatting.

Signed: _____

Matthew Francis

1/11/2024

Acknowledgements

I would first like to acknowledge the privilege I have had of being part of the Computer Engineering Laboratory as an undergraduate. The support and learning I have received from all here has been immense. I particularly acknowledge Mr Rich Rademacher who spent a considerable amount of time answering my seemingly never-ending barrage of questions in both engineering and life.

Further, I would like to extend a similar acknowledgment to Dr David Boland for trusting me to teach his course as a second year student and every year henceforth. It has been a fantastic experience to work with him professionally and academically.

I am very grateful for Dr. Teng-Hui Huang's assistance with all things information theory which underpins much of the theory of this work.

Finally, I would like to acknowledge my supervisor Professor Philip H.W. Leong for his support not only in the writing of this thesis but throughout my undergraduate degree. From visiting the radio club to teaching me how to solder, Professor Leong has been a stalwart throughout my degree and I am forever indebted to him.

Abstract

In recent years, large language models have captured public attention for their impressive inference capabilities and human-like reasoning. Less widely recognised, however, is their potential for data compression. Indeed, a model's ability to predict coincides with its ability to compress. The success of modern language models is attributed to the transformer architecture, which efficiently learns dependencies within data. In 2024, DeepMind demonstrated that transformer architectures could be repurposed as data compressors when trained on the data-type to compress. Furthermore, they demonstrated that large foundation models could serve as general-purpose compressors for data types on which they were not explicitly trained, using their in-context inductive abilities.

This work applies transformer architectures to compress WiFi 802.11ax data generated in MATLAB. While DeepMind was unable to achieve compression on time-series data, we showed with explicit training that WiFi time-series data can be compressed. We expand on their training regime by incorporating a cosine annealing learning schedule, as well as relative positional embeddings. When benchmarked against `gzip`, we demonstrate that our approach achieved superior compression rates in both noiseless and noisy scenarios (30 dB signal-to-noise-ratio), outperforming `gzip` by 7.2 % and 21.54 % respectively.

We provide evidence to support the literature which suggests that large foundation models have strong in-context inductive abilities by demonstrating that LLaMA could effectively compress WiFi data without explicit training. We achieved a noiseless compression rate of 71.6 %, falling short of `gzip` by 7.5 %. However, as noise increased, LLaMA surpassed `gzip` by 4.5 % at 40 dB signal-to-noise ratio. Attempts to enhance LLaMA's performance by training a linear head appended to the final transformer block proved ineffective, likely due to the linear architecture's limited capacity to encode additional information.

Recognising that real radio frequency data varies in amplitude, we proposed a scale-invariant training and compression algorithm. However, this approach did not outperform either `gzip` or the developed models, which were not trained on various scales. It was found that models not trained to be scale-invariant were still able to compress across scales they were not trained on.

The findings of this work suggest that transformers can achieve performant compression rates but, practical implementation remains limited due to their demanding time and space complexity, as well as their limitations as a universal model of compression.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

AC Arithmetic Coder

ASCII American Standard Code for Information Interchange

AWGN Additive White Gaussian Noise

BPSK Binary Phase-Shift Keying

CDF Cumulative Distribution Function

DNN Deep Neural Network

FFT Fast Fourier Transform

GPT Generative Pre-Trained Transformer

GPU Graphical Processing Unit

IID Independent and Identically Distributed

LLaMA refers specifically (unless specified) to LLaMA-2-7B-Chat

LLM Large Language Model

ML Machine Learning

PMF Probability Mass Function

QAM Quadrature Amplitude Modulation

RF Radio Frequency

RoPE Rotary Positional Embeddings

RU + CA Ramp Up and Cosine Annealing

RV Random Variable

SNR Signal-to-Noise Ratio

SPE Sinusoidal Positional Embedding

WiFi refers specifically (unless specified) to WiFi 802.11ax

Mathematics

$\log(\cdot)$ Logarithm in base-2.

$\mathbb{R}$ The set of real numbers.

$\mathbb{R}^n$ The set of n-dimensional real vectors.

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

$I(X; Y)$ Mutual information of X and Y

$p(x)$ $\mathbb{P}[X = x]$ under some distribution p.

Chapter 1

Introduction

1.1 Motivation and aims

The information age is defined by an insatiable demand for large amounts of data [93]. Everyday services such as YouTube exemplify this trend with an estimated 293 petabytes (293×10^{15} bytes) uploaded every year [12]. This pales in comparison to the raw data generated annually by the Large Hadron Collider, which reaches 40 zetabytes (40×10^{21} bytes) before any data reduction is applied [28]. While advancements in solid-state physics have enabled denser storage media [36], an additional degree of freedom may be improved: the format in which data is stored. This study examines the concept of compression; storing data with fewer bits while preserving the ability to recover the original, useful information. Compression exploits the structure of underlying data by identifying and eliminating statistical redundancy.

Statistical redundancy refers to the presence of order in non-random data and serves as a measure of this non-randomness. Data compression capitalises on this fact, and its use in data storage is well-established. For instance, `gzip` [31] is a ubiquitous [91] programme that compresses files by identifying and efficiently encoding repeated patterns in data [104]. Repeated patterns are not the only method in which models can extract statistical redundancy. Data may have underlying dependencies which make their output calculable, or at least predictable. Language is one such example; its redundancy stems from certain words being more likely to follow others rather than explicit repetition.

In recent years, the rise of the transformer architecture [92] has accelerated the development of large language models (LLMs) such as ChatGPT [67] and LLaMA [21, 90] which exhibit human-like conversational ability. The strength of this machine-learning architecture is borne of its ability to learn and identify dependencies within language, much like how humans discern meaning through contextual understanding. This thesis extends these principles to data compression, exploring how transformers can leverage their ability to capture these hidden structures to achieve more efficient compression than traditional tools like `gzip`.

Delétang et al. [20] demonstrated that untrained transformers could be trained to compress text, but did not achieve competitive compression on other data modalities such as audio and images. They did however show that large foundation models such as LLaMA were able to compress on these modalities without being explicitly trained on such data suggesting an impressive degree of in-context learning and induction.

This work aims to compress WiFi IEEE 802.11ax data. Collection and storage of radio frequency (RF) data has many uses in machine learning as training data for downstream tasks such as modulation classification [76] as well as radio frequency fingerprinting [32]. Radio frequency data such as WiFi provides unique challenges in learning a potential compression scheme due to variations in amplitudes and the presence of noise. This thesis therefore has two primary aims:

1. Train and use transformer architectures to compress better than `gzip` on WiFi data; and
2. Train such an architecture to be robust against different scales and noise levels.

1.2 Contributions

This work proposes a transformer-based model than can compress quantised WiFi data. With appropriate hyperparameter choice and training setup it was shown to out-perform `gzip` by 7 % in a noiseless environment and 21.5 % at a signal-to-noise ratio of 30 dB. Further, we provide evidence to support the notion that large foundation models have strong zero-shot inductive capabilities by demonstrating that LLaMA could compress WiFi data, outperforming `gzip` by 4.5 % at a 40 dB signal-to-noise ratio. We trained an additional 'head' to append at the output layer of LLaMA to improve its compression rate but found this only hindered LLaMA's performance. Recognising that WiFi data can have varying amplitude, we proposed an algorithm to train and test a transformer model to be invariant to this. Such a model was unable to outperform `gzip`, but we showed that models not explicitly trained various scales were able to maintain their compression rate when the data's amplitude was artificially adjusted.

1.3 Thesis structure

The work is divided into five sections; background, methodology, results, discussion, and conclusion. Each chapter includes an introduction and summary for quick reading. The background assumes no prior knowledge of information theory and established a foundation to understand data compression. Further, it introduces machine learning concepts and motivates the use of the transformer architecture as a potential model for compression. The methodology details the various transformer architectures and training schemes explored, outlining specific combinations used to achieve the final results. The results showcase the capabilities of our best compressor, benchmarking its performance against `gzip` across various noise and scale levels. We outline the performance of our scale-invariant transformer,

as well as the performance of LLaMA with and without an appended linear head. The discussion and conclusion explore why certain training schemes and architectures proved more effective than others, and provide commentary on the practical applications of the use of transformers for data compression.

Chapter 2

Background

2.1 Introduction

The background of this work explores three core areas: data compression, machine learning, and WiFi data. The chapter begins with a mathematical introduction to coding theory which serves a foundation for data compression. Various practical implementations of compression algorithms are explored and gesture to the idea that more sophisticated models lead to improved compression. Machine learning concepts are introduced through examples culminating with an introduction to the transformer architecture. Data compression and machine learning are united, establishing that a strong transformer model makes a strong compressor. A brief overview of the taxonomy of WiFi data and a literature review conclude the chapter.

2.2 Coding theory

Fundamental to how computers store, transmit, and process data is the concept of codes. Intuitively one can understand that data inside any machine is stored as a representation of some reality. For instance, the number 6 is stored in a computer as 110_2, a combination of 'ons' and 'offs'. We are limited to a binary alphabet due to digital computers' limitation in only being able to represent a 1 or a 0 as a given voltage. Coding theory therefore plays a crucial role in computer science as a means of formalising ways to represent values meaningfully in various contexts.

One practical example is cryptographic coding [64], where information is coded to prevent unwanted parties accessing data through obscurity. For a formal introduction to these concepts we refer to *Elements of Information Theory* by Cover and Thomas [17]. We will herein build on this framework to discuss coding in the context of compression, the primary focus of this work. We rely on [13] for the following introduction.

We consider data to be the realisation of a random variable (RV) X described by

$$X : \Omega \to \mathcal{X}, \tag{2.1}$$

where Ω is the sample space of all possible outcomes. The probability of $x \in \mathcal{X}$ appearing is $\mathbb{P}(X = x)$. We define a the function C as a code that takes some outcome x and transforms it into some other representation. We characterise this representation by its alphabet Σ which is a set containing the possible symbols present in the output. For example, in a computer system we may say that

$$\Sigma = \{0, 1\}.$$

We use the Kleene star operator to represent all possible concatenations of the elements of Σ, written Σ^*. Therefore for the example above,

$$\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000 \dots\},$$

where ε is the empty set. We can now define a code as a function C where

$$C : \mathcal{X} \to \Sigma^*. \tag{2.2}$$

We write $C(x)$ as the code associated with x and $l(C(x))$ as the length of the code word associated with x.

2.3 Compression

Data compression is a coding scheme where the coding function C aims to minimise the length of its output while still keeping the original data recoverable. Recovery can occur in two ways. First, data may be compressed so that even if parts of the original content are lost, it remains understandable to humans; this is known as lossy compression. Lossy compression is widely used in daily applications such as when sending photos on mobile phones, where the image quality is acceptable but reduced to save bandwidth. Algorithms such as JPEG [94] achieve this by removing visually redundant information. Similarly, lossy audio formats such as MP3 [6] take advantage of psycho-acoustic redundancies to remove sounds humans find difficult to perceive.

The second interpretation of recoverability is that the data can be fully restored to its original form without any loss of information. This is known as lossless compression and will be the focus of this work. Lossless compression takes advantage of the statistical redundancy of information found in non-random data. The goal of this work is to find some C that will compress WiFi data better than the scheme used by `gzip`.

2.3.1 Lossless compression

The goal of a compressor C to minimise the average codeword length can be written mathematically as:

$$\mathbb{E}[l(C(x)] = \sum_{x \in \mathcal{X}} l(C(x))\mathbb{P}(X = x). \tag{2.3}$$

Two properties are required of C to ensure data can be fully recovered:

1. C must be *injective*, that is, C uniquely maps $\mathcal{X}^*$ to Σ^*.
2. C must be *instantaneous*, that is, $C(x_j)$ is not a prefix of $C(x_j)$ for $\forall i, j$. Codes that obey this property are known as prefix codes[1].

In order for a code to be uniquely decodable, the Kraft-McMillan inequality (Cover and Thomas [17] p. 108-109) must hold. For each outcome $x \in \mathcal{X}$ with codeword $C(x)$ and length $l(C(w))$ and assuming binary coding, we must have:

$$\sum_{x \in \mathcal{X}} 2^{-l(C(x))} \leq 1. \tag{2.4}$$

Corollary 1 (Pigeonhole Principle)**.** Due to property (1), by the pigeonhole principle, there is no mapping C that compresses all data. The better we make our compressor for WiFi data the worse it will become for other data modalities. There is no such thing as a free lunch [27].

As codes of varying length are considered (as opposed to codes for transmission which work in specific size chunks) all output codes must not be a prefix of any other code. This would create ambiguity when decoding.

Example 1 (Prefix Codes)**.** $C_1 : \mathcal{X} \to \{1, 01, 00\}$ is a prefix code because 1 is *not* a prefix of 01 or 00 (and vice-versa). $C_2 : \mathcal{X} \to \{0, 1, 10\}$ is *not* a prefix code as the string 010 could be decoded as $\{0, 10\}$ or $\{0, 1, 0\}$. Equation (2.4) may be used to show this as well. For C_1 we calculate:

$$\sum_{x \in \mathcal{X}} 2^{-l(C_1(x))} = 2^{-1} + 2^{-2} + 2^{-2} = 1 \leq 1.$$

For C_2:

$$\sum_{x \in \mathcal{X}} 2^{-l(C_2(x))} = 2^{-1} + 2^{-1} + 2^{-2} = 1.25 \not\leq 1.$$

[1]The nomenclature is inverted - prefix codes do *not* have prefixes.

The discussion now turns to methods for finding such a C using information theory, with a focus on a type of compression called entropy coding, which takes advantage of the statistical properties of the data X to approach a theoretical optimum.

2.3.2 Entropy coding

In 1948 Shannon established the foundation of information theory with his seminal paper, *A Mathematical Theory of Communication* [80], which serves as a cornerstone for the theoretical framework underpinning this thesis. We are primarily concerned with what Shannon coined entropy. Many disciplines of science have co-opted the use of the word entropy however they all point towards the same underlying concepts of uncertainty and information. In this paper we adhere to the definition described formally by Shannon and, informally as a measure of uncertainty or 'surprise' in the realisation of a random variable. We refer to Carter [9] as an introductory guide for these concepts.

Everyday life demands an intuitive understanding of probability. A fair coin has a 50% chance of landing on heads. It happens (roughly) half as many times as one flips the coin. But what about the 'information' learned from a single flip? Shannon related the probability of an event to the amount of information it conveys, capturing this relationship in three ways [80, 22]. Denoting the information of an event with probability p as $I(p)$:

1. The more probable an event is; the less information we obtain in its observation. $I(p)$ is monotonically decreasing in p;
2. We gain no information from an event that always occurs. $I(p) = 0$ when $p = 1$, and;
3. The information we gain from two independent events must be additive, $I(p_1, p_2) = I(p_1) + I(p_2)$.

These constraints about the intuition of information lead to a mathematical definition of the information of an outcome x as:

$$I(x) := -\log[\mathbb{P}(x)], \tag{2.5}$$

as the logarithm function provides all of the above properties. We therefore define entropy (2.6) as the expected information content of some random variable $\mathcal{X}$ as the weighted average of the information content of the variable, measured in bits.

$$H(X) := -\sum_{x \in \mathcal{X}} \mathbb{P}(x) \log \left[\mathbb{P}(x)\right]. \tag{2.6}$$

Shannon then proved that for any coding scheme C the number of bits required to code N outcomes of an independent and identically distributed (i.i.d.) X would minimally take $NH(X)$ bits. Formally, with the outcomes of X induced under some probability distribution P:

$$\mathbb{E}_P[C(l(x))] \geq H_P(X), \tag{2.7}$$

known as Shannon's source coding theorem. A compression scheme is optimal if its expected length approaches H_P, or its total length approaches $NH_P(X)$ which we will refer to as the 'Shannon limit'. P is explicitly written as a subscript of the entropy function H to emphasise the point that a compressor is limited by the entropy in the context of the chosen probability distribution. This idea is discussed further in section 2.3.2. Further, this limit applies to the expectation of the code-word length. Examples can be provided which 'beat' the Shannon limit but on average a scheme cannot compress better than the Shannon limit as the number of realisations grows *ad infinitum*.

Example 2 (A coin toss)**.** A coin toss is provided as an example. The sample space is $\Omega = \{\text{heads}, \text{tails}\}$ and the probability distribution is $P = \{0.5, 0.5\}$. The entropy is calculated as:

$$\begin{aligned} H(X) &= -0.5 \cdot \log(0.5) - 0.5 \cdot \log(0.5) \\ H(X) &= 1. \end{aligned}$$

To encode N flips would take $H(X) \times N = N$ bits. We denote 0 for 'tails' and 1 for 'heads', that is $\Sigma = \{0, w\}$. Intuitively for an equally likely outcome, the best way to represent the outcomes is by two unique representations of equal length. We say that this random variable has no redundancy. Redundancy is defined as:

$$R = 1 - \frac{H}{H_{max}}, \tag{2.8}$$

where H_{max} denotes the entropy of a random variable such that all symbols are equally likely, as in, the random variable is uniformly distributed. Taking the above example we find:

$$\begin{aligned} R &= 1 - \frac{1}{1} \\ R &= 0. \end{aligned}$$

That is to say, there is no redundancy in this information source. Contrast this with an unfair coin which has probability 0.95 to land on heads. The entropy is calculated as:

$$\begin{aligned} H(X) &= -0.95 \cdot \log(0.95) - 0.05 \cdot \log(0.05) \\ H(X) &\approx 0.2864. \end{aligned}$$

The entropy (average information per outcome) has reduced. This is expected, we are highly likely to see 'heads' and there is less surprised when we do. The redundancy is approximately 0.7163, that is to say, if we encoded 10 flips with 1 for heads and 0 for tails as before, approximately 7 of those bits would be 'redundant' and could be compressed by an appropriate algorithm. It is therefore possible to encode 10 flips with only 2.864 bits. It is obscure to think of a non-integer number of bits. Recall that Shannon's source coding theorem considers the limit as the data's length approaches infinity. The principal questions to consider are:

i How is redundancy practically removed?

ii How is P determined for a given set of data?

Subsection 2.3.3 will discuss different methods to practically take advantage of these redundancies. On a basic level, these implementations work by representing likely outcomes with fewer bits, and unlikely outcomes with more bits. Determining the underlying distribution P relies on statistical methods, with the choice of these methods being central to this work. The implication of the pigeonhole principle is that our choice in modelled probability distribution is capital. section 2.3.2 discusses the implications of different choices of probability models and section 2.4 shows that the closer these are aligned with the real data probabilities, the better the compression scheme.

Beyond I.I.D. random variables

The above example illustrates the redundancy that can be found in i.i.d. random variables. Indeed efficient compression can be achieved by heuristically calculating the probability distribution of the data through the frequency of observed symbols. This is the underpinning of Huffman coding described later in section 2.3.3. This thesis aims to find a compressor C that does not just use the frequency of the data but rather learns statistical dependencies between individual observations. WiFi data is *not* independent. That is,

$$\mathbb{P}(x_i \mid x_{j<i}) \neq \mathbb{P}(x_i). \tag{2.9}$$

To consider each symbol individually disregards much of the information intrinsic to a source. For example, coding the letters of some English text, the letters q and u illustrate this limitation as their appearance in language is not independent of each other. The letter q is extremely likely to be followed by u, however coding these letters in a memoryless system does not allow such a dependency to be exploited. The redundancy found in frequency may be considered, but not the redundancy of order.

Higher order entropy addresses the issue of extracting redundancy out of non i.i.d random variables by consider the ordering of the data. The coin-toss example used zero-order entropy, where only individual frequencies are considered to extract redundancy. An example sentence is provided which is used to compare zero-order entropy with first-order entropy to show more redundancy may be extracted. Langmead [55] is used as inspiration for the following example.

Example 3 (Queens and quiche)**.** Consider the sentence

the queen requested exquisite quiche

A realisation of the random variable representing this sequence is the appearance of a letter. The zero-order entropy is calculated over the outcomes (letter appearances) ignoring whitespace. The following probabilities are observed:

letter	t	h	e	q	u	n	r	s	d	x	i	c
frequency	3	2	9	4	4	1	1	2	1	1	3	1
probability	0.094	0.063	0.28	0.13	0.13	0.031	0.031	0.063	0.031	0.031	0.094	0.031

The the average information per realisation (that is, the entropy) is calculated as $H_0 \approx 3.186$. The 0 subscript emphasises the zero-order entropy. Indeed this 32 long sequence can be theoretically compressed to a maximal limit defined in (2.7) as $32 \times 3.186 \approx 102$ bits. Compared to a standard 8-bit representation of letters a theoretical compression rate of $r = \frac{32 \times 3.168}{32 \times 8} = 0.396$ can be achieved.

We repeat the process with first-order entropy, which incorporates context into its calculation. Imagine a computer programme which reads the sequence letter-by-letter. The programme has a memory of the the current letter and calculates the probability of the following letter conditioned on the current letter. Each letter is considered a state in a Markov chain where the transition probabilities are the probabilities of the next letter appearing given the current letter. This is represented as the transition matrix $\mathbf{P}$ and is generated through observation of the example sentence.

$$\mathbf{P} = \begin{array}{c} \\ T \\ H \\ E \\ Q \\ U \\ N \\ R \\ S \\ D \\ X \\ I \\ C \end{array} \begin{array}{c} \begin{array}{ccccccccccc} T & H & E & Q & U & N & R & S & D & X & I & C \end{array} \\ \begin{pmatrix} 0 & 1/3 & 2/3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/8 & 3/8 & 0 & 1/8 & 0 & 1/8 & 1/8 & 1/8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1/2 & 0 & 0 & 0 & 1/2 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

The matrix provides a mathematical representation of the previously mentioned intuition, that q is likely to be followed by q. In fact in this case, q is always followed by a u indicated by $\mathbf{P}_{QU} = 1$. We invoke the formula for k-order entropy from Langmead [55] as:

$$H_k(S) = \sum_{t \in \Sigma^k} \frac{|S_t|}{n} H_0(S_t), \tag{2.10}$$

where n denotes the length of the sequence, Σ^k represents all concatenations of elements of the alphabet Σ up to length k, and S_t is the concatenation of all symbols with context t.

Here, $k = 1$, so t enumerates over each individual letter. As an example, when $t = u$, $S_u = \{eeii\}$. Intuitively, this calculation derives the entropy conditioned on a known state. Conditional entropy [15] may be thought of as the entropy of a random variable given another random variable has already occurred; if there is a relationship, there is less surprise in our observation because we know some information about the state of the system. The conditional entropy of Y given $X = x$ is given below:

$$H(Y \mid X = x) = -\sum_{y \in \mathcal{Y}} \mathbb{P}(Y = y \mid X = x) \log \mathbb{P}(Y = y \mid X = x). \tag{2.11}$$

$H(Y \mid X)$ is then the expectation of the above over $\forall x \in \mathcal{X}$. Equation (2.10) is used as an empirical way of calculating conditional entropy with random variables $X = x_0$ and $Y = x_1$ where each x_i represents a letter with ordering $\{x_0 x_1\}$.

The first order entropy of the example sentence is calculated as:

$$H_1(S) = \sum_{t \in \{T,H,E,Q,U,N,R,S,D,X,I,C\}} \frac{|S_t|}{32} H_0(S_t) \approx 1.01.$$

The entropy has reduced, and this particular scheme can achieve a compression rate of $\frac{32 \times 1.01}{32 \times 8} = 0.126$. There are combinations in the sentence which are completely redundant; q is always followed by u, and r is always followed by e. The conditional entropy of u given q is 0, there is no surprise.

Mutual information is used to quantify how much information about one random variable is gain from the observation of another. For the discrete case:

$$I(X;Y) = \sum_{y \in Y} \sum_{x \in X} \mathbb{P}_{(X,Y)}(x,y) \log \left(\frac{\mathbb{P}_{(X,Y)}(x,y)}{\mathbb{P}_X(x) \mathbb{P}_Y(y)} \right) \tag{2.12}$$

It can be shown that

$$I(X;Y) \equiv H(X) - H(X \mid Y) \equiv H(Y) - H(Y \mid X). \tag{2.13}$$

Observe that $I(X;Y) \geq 0$ and the equality is obtained when X and Y are independent in which case no information with respect to the other by observing either outcome. Perfect information is gained when X and Y can fully describe one another in which case equation (2.13) degenerates to the entropy of either variable. In the case of q and u in the example sentence:

$$I(X = q; Y = u) = H(Y = u) - H(Y = u \mid X = q) = H(X = x) - H(X = q \mid Y = u)$$
$$= H(Y = u) = H(X = q).$$

The amount of information q reveals about u is the same as simply observing u. There is no additional uncertainty, and observing q is effectively the same as observing u (and vice-versa.)

The following section will describe different compression schemes which take advantage of the preceding theory and provide a practical approach to compressing data. Further, we introduce `gzip` as a practical model which is used as a benchmark in this study.

2.3.3 Compression schemes

Compression techniques in practice aim to approach Shannon's source coding limit with varying trade-offs. The following example demonstrates Huffman coding which creates a code through observed data frequencies similar to zero-order entropy. The example is followed by an explanation of the algorithm that underpins `gzip`. The final compression scheme and the focus of this work is arithmetic coding. As it underpins this thesis, it will be described in its own section.

Huffman coding

Huffman coding [40] is a compression scheme that works by reducing the number of bits required to represent more frequent data. Huffman coding is a prefix code and is uniquely decodable. It uses a frequency-sorted binary tree to find the optimal prefix code given a set of observed frequencies induced over an alphabet of symbols.

Example 4. Given the sequence $\{x_0x_1 \dots x_n \mid x \in \mathcal{X}, n \in \mathbb{N}\}$ for some X, a sequence with $n = 40$ is generated:

abaaabacaaaabcabbaaaabaaacaababababaabaaad,

with symbol probabilities in Table 2.1 and generate the corresponding binary tree shown in Figure 2.1 [42].

x	a	b	c	d
p	0.65	0.25	0.075	0.025

Table 2.1: Observed probabilities in Huffman coding example.

The coding scheme is generated by concatenating a 1 for a visit to the left child and a 0 for a visit to the right child until the symbol's leaf node has been reached. Generation of such a tree will result in a prefix code and may be confirmed using the Kraft inequality described bt (2.4):

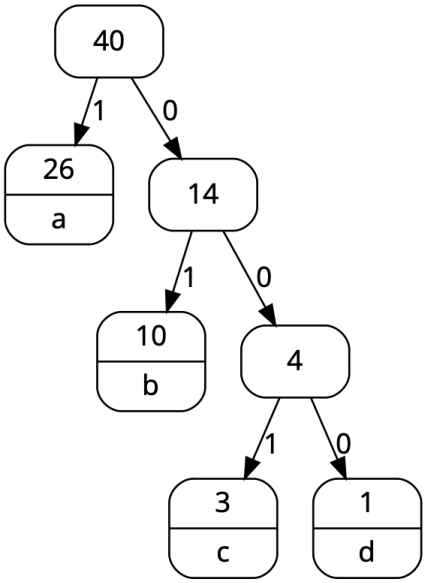


Figure 2.1: Huffman tree coding for letter sequence

$$2^{-1} + 2^{-2} + 2^{-3} + 2^{-3} = 1 \leq 1.$$

The alphabet $X = \{a, b, c, d\}$ is coded by $\Sigma = \{1, 01, 001, 000\}$. The string may be coded in $1 \times 26 + 2 \times 14 + 3 \times 3 + 3 \times 1 = 56$ bits. Contrast this with a naive scheme where each symbol is represented by two bits[2] requiring a total 80 bits. The entropy of X is calculated to be approximately 1.317; an optimal scheme could code the 40 symbols with 52.68 bits.

Recall the entropy of a source is equivalent to the average number of bits required per symbol to code the source. The entropy was found to be 1.317 however the Huffman scheme above had an average code length of 1.4 bits. This rises from the fact that we cannot code each individual symbol with a non-integer number of bits - a pitfall of Huffman coding. Huffman coding also only considers the individual letter frequency which as demonstrated, can be improved on.

In the special case that the probability distribution of the symbols is 2-adic, Huffman coding will reach the theoretical optimum [41], that is, the probabilities take the form:

$$f(x) = 2^{-n_x} \quad n \in \mathbb{N}.$$

Gzip

`Gzip` is a free and open source compression programme from the GNU suite of software. The algorithm uses a combination of the Lempel-Ziv 77 (LZ77) algorithm and Huffman coding.

[2]Most numerical representations in computer systems have a fixed length, such as `int8` which represents integers with up to 8 bits representing the range $[0, 255]$.

Ziv and Lempel's original IEEE paper [104] serves as an explanatory guide for the following section. In addition, we refer to supplementary lecture notes from Northwestern University [1]. LZ77 is a *universal* dictionary coding scheme. Universality indicates the algorithm can approach the entropy limit of the source[3] without previous knowledge of the data. This is an attractive feature for a compression programme as data does not need to be learned before and therefore can be used on any file (Cover and Thomas [17] sec. 12.10).

The premise of LZ77 is to represent repeating patterns in data with pointers to previous occurrences. The pointer consists of two numbers:

1. An offset from the current position to the repeating sequence; and
2. The amount of bytes to copy forward.

The dictionary of patterns to choose from consists of all the previously read data.

Example 5 (LZ77 Example)**.** Consider the sentence from Dickens' Tale of Two Cities:

it was the best of times, it was the worst of times.

There is redundancy in this string due to the repeated phrase. Observe an iteration of the `gzip` algorithm where we have read up to the start of the second clause. 'It was the' can be replaced by a pointer to the already-parsed phrase. Pointers are represented by $\mathrm{P}\langle .,. \rangle$ where the first argument is the reversed offset, and the second is the amount of characters to copy forward.

$$\overbrace{\text{it was the best of times, }}^{w}\underset{\uparrow}{\text{it}}\text{ was the worst of times.}$$

'it was the ' is replaced by a pointer referring to 26 characters earlier, 11 characters are copied forward.

$$\overbrace{\text{it was the best of times, }\mathrm{P}\langle 26{,}11\rangle}^{w}\underset{\uparrow}{\text{w}}\text{orst of times.}$$

Continuing, 'of times' is copied forward.

$$\overbrace{\text{it was the best of times, }\mathrm{P}\langle 26{,}11\rangle\text{worst }\mathrm{P}\langle 18{,}8\rangle}^{w}\underset{\uparrow}{.}$$

This process is repeated over the entire string. The maximum size of the context window w is 32 kB and the maximum amount of characters to copy forward is 258. The algorithm is a dictionary code where the dictionary values are inside the string itself. The string is then further compressed using Huffman coding, extracting any remaining redundancy in the data.

`Gzip` is used as a benchmark to compare to the transformer models presented in this work. The choice it use `gzip` stems from its ubiquitity [91] and its inclusion with most Linux distributions [30, 31].

[3] Requiring the source to be stationary and ergodic.

Research into the synergies between neural networks and existing compressors like `gzip` are becoming more researched topics. Jiang et al. [46] describes a computationally efficient combination of `gzip` and deep neural networks (DNNs) which use k-nearest-neighbours to improve the compression rate of `gzip`.

2.4 Arithmetic coding

Arithmetic coding [96, 3, 73, 20, 66] addresses the inefficiency of having to code each symbol with an integer number of bits seen in Huffman coding. Instead, the entire message is coded into an arbitrary-precision real number in $[0, 1)$.

Consider an alphabet $X = \{a, b, c\}$ with probability distribution $P = \{0.5, 0.3, 0.2\}$. Starting with the interval $I_0 = [0, 1)$, the goal is to code the sequence the sequence `abc`. The interval is subdivided proportionally to P where each sub-interval corresponds to an entry in X. To code a symbol in X, select the sub-interval associated with that symbol and repeat the process within that interval. Figure 2.2 demonstrates this process for the string `abc`.

The string `abc` is represented by the range $[0.37, 0.4)$. Unlike Huffman coding, which encodes individual symbols, this method encodes the entire sequence within a continuous interval and is not restricted to an integer number of bits per symbol. As the sequence length increases the compression rate approaches the source's theoretical entropy.

The final result must be a single binary number that unambiguously represents the interval $[0.37, 0.4)$. For example, $0.011_2 = 0.357_{10}$ lies within the interval, but 0.011_2 is a prefix of $0.0111111_2 = 0.4921875$ which is outside the required range. 0.011_2 is not a valid prefix code as it does not uniquely identify this interval. To find an unambiguous representation, a binary search over the interval $[0, 1)$ identifies a range completely within $[0.37, 0.4)$. For this example, the binary range $[0.011000_2, 0.011001_2) = [0.375, 0.390625)$ is identified, and any digits appended to 0.011000_2 will remain within the specified interval ensuring a prefix code.

This implementation of an arithmetic coder requires infinite computer precision which is impossible (Cover and Thomas [17] sec. 5.10). A real implementation will only require finite precision with one such implementation described in Nelson [66].

The probability distribution is given as $P = \{0.5, 0.3, 0.2\}$ with the entropy calculated as $-0.5\log(0.5) - 0.3\log(0.3) - 0.2\log(0.2) \approx 1.49$. The optimal coding of the string `abc` is therefore $\lceil (l(abc) \times 1.49) \rceil = 5$.

Recall however `abc` was coded as 0.011000 requiring 6 bits (arithmetic coders imply the integer bit). This discrepancy arises from the different in the modelled probability P and the observed probabilities $P_o = \{1/3, 1/3, 1/3\}$ (see Cover and Thomas [17] thm. 5.4.3). Cross entropy is used as a measure of this discrepancy, quantifying the average number of bits required to represent a string with true distribution p under an estimated distribution q. It is defined as:

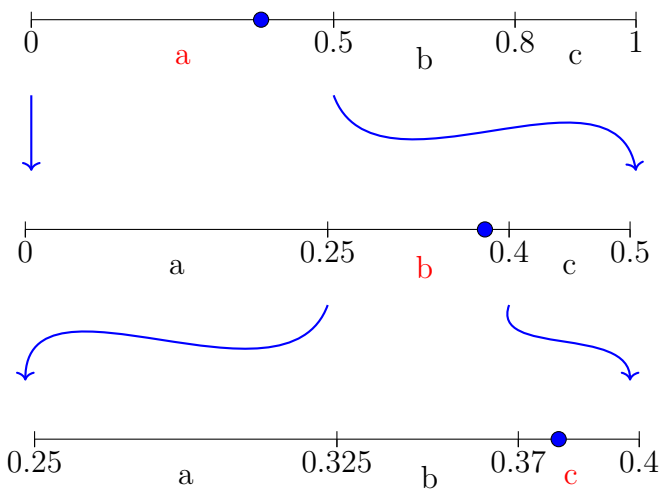


Figure 2.2: Arithmetic coding example for alphabet X and probability distribution P coding `abc`.

$$H(p, q) = -\sum_{x \in \mathcal{X}} p(x) \log q(x). \tag{2.14}$$

Note that p and q share the same support $\mathcal{X}$. The example above has true distribution $P_o = p = \{1/3, 1/3, 1/3\}$ and modelled distribution $\tilde{P} = q = \{1/2, 3/10, 1/5\}$. The cross-entropy is calculated as approximately 1.69 yielding an expected code length of $\lceil 3 \times 1.69 \rceil = 6$ bits, matching the above results. Using Gibbs' inequality [26] it can be shown that the number of coding bits is minimised when $q = p$ approaching the Shannon entropy of the source. This implies that the the closer the modelled distribution is to the observed distribution, the more efficiently we can compress, bounded by the entropy of the source.

An important feature in arithmetic coding is that the probability distribution can be adapted based on context. Instead of coding a sequence $\{x_i\}_{i \in \mathbb{N}}$ with unconditional probabilities $p(x_i)$, the model can instead use conditional probabilities that account for prior symbols. Specifically, each x_i can be coded based on the probability $\mathbb{P}(x_i \mid x_{i-1}, x_{i-2}, \ldots, x_0)$ taking into account the preceding sequence. As in section 2.3.2, a more sophisticated model lends itself to reduced entropy and improves compression efficiency.

WiFi data (and RF time-series data) are dependent on preceding signals, resulting in non-zero mutual information between an observation and prior values. This implies by Equation 2.13 that the conditional entropy is strictly less than the unconditional entropy of each symbol; there is less surprise observing x_i having observed $\mathbf{x}_{j<i}$.

This relationship suggests that if the underlying distribution of WiFi data can be learned and deterministically calculated given a prior sequence, efficient compression of the sequence is achievable using an arithmetic coder. For both compression and decompression the probability distribution must be causal, relying only on previously observed data. Since decompression cannot use future data, it calculates probability distributions based only on the

data already decompressed. Compression must therefore adhere to the constraint to ensure equal probability distributions in both processes.

Two essential points should be clear:

1. The more contextual information we can extract to model an estimated probability distribution q of the underlying data, the more efficient the resulting compression.
2. The closer our estimated distribution q is to the observed distribution p, the better the compressor.

The fundamental challenge now becomes *how can the distribution of future data be accurately predicted based on past data?* The hidden complexities of many time-series data are challenging to empirically model. Transformers, a machine learning architecture designed for handling sequential data, offer a powerful framework for learning approximating these underlying distributions. Their architecture allows them to model long-range dependencies and relationships within data. While commonly used to find relationships in natural language, this work will show they can be used to find these relationships in WiFi data.

2.5 Machine learning

Machine learning is the branch of science that uses algorithms to improve a machine's performance on specific tasks. Through repeated experiences humans are able to learn concepts and skills, and computer architectures can be designed to emulate such learning. Mitchell [62] is used as a foundation for the following sections. Examples presented drew inspiration from Fox [23]. As defined by Mitchell [62] on p. 2:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

In this work, our task T is to compress WiFi data, with performance measure P as the compression rate. Later it will be shown that experience E is repeated exposure to example data. The following example introduces key concepts and nomenclature foundational to these ideas. The following is an example of a support vector machine which was proposed by Cortes and Vapnik [16].

Example 6 (Flowers and weeds)**.** Imagine a garden bed divided into sections of flowers and weeds. The task is to identify the line that best separates these regions so that a new example can be 'classified' as a flower or a weed.

Given n training points $(\mathbf{x_1}, y_1), \ldots (\mathbf{x_2}, y_2)$ where each $\mathbf{x_i}$ is a vector of position of the plant in the flower-bed, and each y_i determines the examples 'class', (e.g. flower or weed). More generally, we consider each element of the vector $\mathbf{x_i}$ a 'feature' of the data. A more useful model might consider each data-point to be a patient in a hospital with features corresponding to health attributes (e.g. weight, sex, height) and classes assigned to different diagnoses.

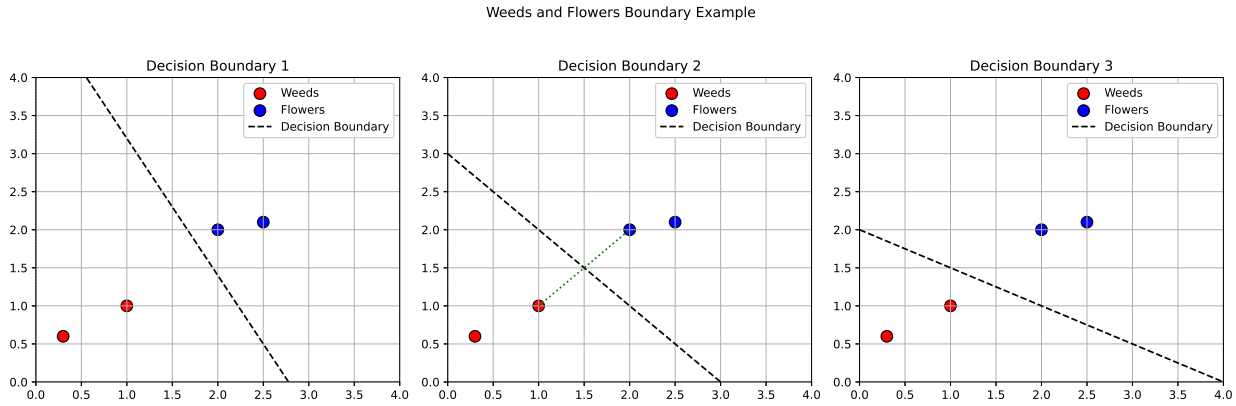


Figure 2.3: A plane of flowers and weeds, with different decision boundaries separating each class.

Returning to the garden-bed,the computer completes task T of classifying plants as flowers or weeds by finding a line that separates the two classes, subsequently classifying new examples based on their position relative to this line. How then should model performance (P) be evaluated and can improvement (E) be achieved? Finding a line that separates the training data is straightforward and three possible boundaries are shown in Figure 2.3. Occam's razor[4] (Mitchell [62] sec 3.6) suggests the simplest solution is the best, suggesting boundary 2 as it evenly and maximally separates the garden bed with all the information provided. Boundary 1 and 3 have inexplicable complexities.

This is an example of a support vector machine, and a more detailed description may be found in Suthaharan [87]. The optimal decision boundary described by the line $\mathbf{w}^T\mathbf{x} - b = 0$ with $\mathbf{w}$ and b constrained such that the two closest points in each class (the support vectors) lie on the line $\mathbf{w}^T\mathbf{x} - b = 1$ and $\mathbf{w}^T\mathbf{x} - b = -1$. In the example the support vectors are $(1, 1)$ and $(2, 2)$. Under these constraints the width of the maximally separating hyperplane is $\frac{b}{\|\mathbf{w}\|}$. Performance P is optimised by minimising $\|\mathbf{w}\|$ and b. With N training examples this is formulated as:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ \text{s.t.} \quad & y_i(\mathbf{w}^T\mathbf{x}_i - b) \geq 1 \quad \forall i \in \{1, \ldots, N\}. \end{aligned} \tag{2.15}$$

The above expression defines the loss function of this architecture where $\frac{1}{2}\|\mathbf{w}\|^2$ being the loss. Machine learning training aims to minimise this loss. In the example, $\mathbf{w} = [1, 1]$ and $b = 3$. This reduces the decision making capabilities to three values, which are known as the parameters of the model and fully define its function. Take a new plant (which is a weed) at coordinates $\mathbf{x}_j = [1, 0.5]$. Using decision boundary 2, the plant falls on the weed side of the decision boundary. The model correctly classifies it as a weed. Our performance P can be measured as being 100 % accurate. We have:

[4]Entia non sunt multiplicanda praeter necessitatem.

i Trained a machine learning model on a task T (to classify weeds and flowers);

ii Evaluated the performance metric P as its ability to correctly classify a new measurement;

iii Improved the model (E) through observations of existing plants in the garden bed.

Example 6 illustrates the canonical process for training and applying a machine learning model. This example is provides two conveniences:

i The features and classes of the training data are known; and

ii The loss function has a closed solution.

The following section discusses the situation where these conditions are not met.

2.5.1 Unsupervised learning

Predicting the next word in a sentence is a well-defined task for a machine learning model. With publicly available language models like LLaMA [90, 21] and ChatGPT [67], the concept is readily accessible, but it remains open to question; how is such a model trained? Manually identifying the features of natural language is an insurmountable task. One might begin with simple deterministic rules, such as 'the' being followed by a noun or that adverbs precede verbs. Clearly, the hidden complexities of language are not so easily defined. Moreover, as there is no clear rule for 'correct' language, there is no deterministic solution for any loss function. Instead, parameters $\mathbf{w}$ are found iteratively with:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \eta \nabla H(\mathbf{w_i}), \tag{2.16}$$

where $H(\mathbf{w})$ denotes the loss function with respect to parameters $\mathbf{w}$, and η as an adjustable parameter known as the learning rate. The next set of weights is found by moving in the direction that decreases the loss function relative to the current set of weights tending towards a local minimum of $H(\mathbf{w})$. A language model thus requires a computable and differentiable loss function H that when minimised provides parameters capable of accurately predicting the next word. Before language models are discussed in section 2.6 we briefly overview this gradient descent and how minimising this loss function allows us to extract hidden features embedded in language

2.5.2 Deep neural networks

Deep Neural Networks (DNNs) are multi-layered machine learning models capable of learning hidden 'features' within data without their explicit definition at the input (Mitchell [62] sec. 4.6.4). In Example 3 it was demonstrated that a model could be trained to classify a plant as a flower or a weed, using known features about the plant (its position in the garden bed.) DNNs are able to find rules through exposure to inputs (and an appropriate loss function) on their own. When training a model on English text, it is not required to tell the model

what a grammatical article is[5] and yet it may learn that 'the' and 'an' are typically followed by nouns. The model architecture used in this work is the transformer [92] and is elaborated on in subsection 2.6.1.

The correct choice in architecture, model size, and training will allow a model to extract useful features. The extraction of features in training DNNs is described in Mitchell [62] p. 106-107. The following example of a DNN relies on Fox [23] sec. 2.1.3.

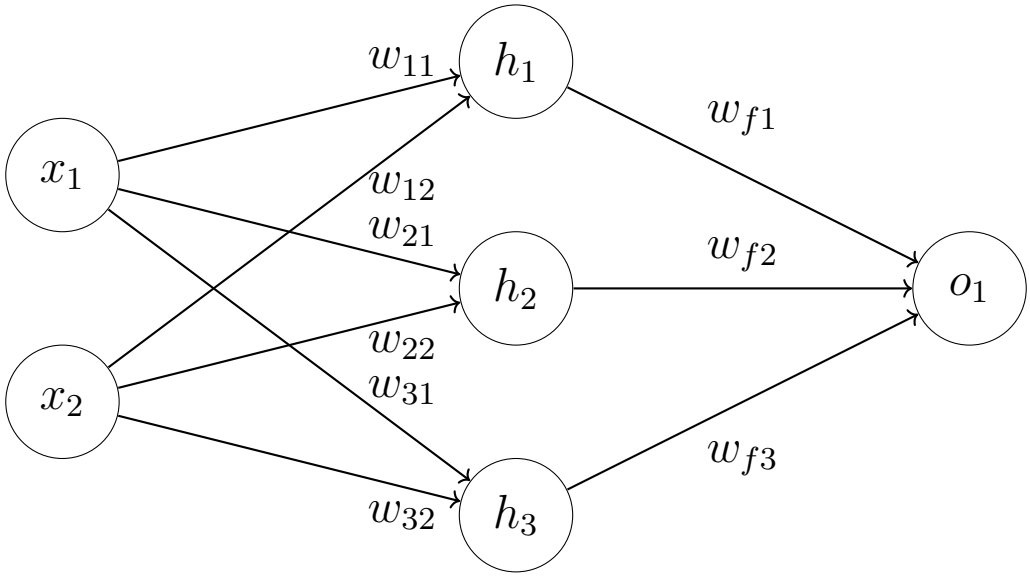


Figure 2.4: Example of a neural network with two input features, one hidden layer, and one output feature.

Figure 2.4 is used an example of a small deep neural network, with such a model commonly called a perceptron. Each node is a neuron which aims to mimic the neurons of the brain although this analogy has limitations (Mitchell [62] sec 4.1.1). Each neuron performs the operation:

$$o_i = g(\mathbf{w_i}^T \mathbf{x} + b) \tag{2.17}$$

where g is a non-linear function. In a network with a single hidden layer, such as this one which has three hidden neurons (a hidden layer is a layer than exists between the input layer and the output layer) and a non-linear function g it can be trained to approximate any Borel measurable function to some arbitrary error ε and is thus a universal approximator [37]. This is significant: any neural network can be trained to approximate any desired function with accuracy determined by number and configuration of neurons. This work aims to create a function with machine learning that can compress WiFi data.

To determine the appropriate weights, it is necessary to examine how these networks are trained. The network output is the result of combining the neuron equation (2.17) as a

[5] An article is a type of adjective that specifies how identifiable the noun is to the speaker and listener.

function considering all the neurons in a single layer. Each layer i represents an application of its function f^i:

$$o_1 = f^o(f^h(\mathbf{x}))$$

where input $\mathbf{x}$ is the input first processed by the hidden layer f^h, then the output layer f^o, yielding output o_1. Each layer is a matrix multiplication, with the hidden layer being equivalent to:

$$f^h\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = g\left(\begin{bmatrix} w_{11} & w_{w21} \\ w_{12} & w_{w22} \\ w_{13} & w_{w23} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}\right) \tag{2.18}$$

Written in vector form:

$$f^h(\mathbf{x}) = g(\mathbf{W}^T\mathbf{x} + b) \tag{2.19}$$

where $x \in \mathbb{R}^2$, $W \in \mathbb{R}^{2\times3}$, and $f^h(\mathbf{x}) \in \mathbb{R}^3$. f^o is then applied to the output of f^h. The objective is to find the weights and biases that approximate the target function. Given input training data $\mathbf{x_i}$ with real output y_i, an estimated output $\hat{y}_i$ can be calculated as $f^o(f^h(\mathbf{x_i}))$. The loss is computed as a distance between the real y_i and estimate $\hat{y}_i$. An example measure is the mean squared error taken as $\frac{1}{2}(y_i - \hat{y}_i)^2$ [59]. The output of this function H is an error E. Using the chain rule this can be decomposed into:

$$\frac{\partial H}{\partial w_{f1}} = \frac{\partial H}{\partial out_o}\frac{\partial out_o}{\partial net_o}\frac{\partial net_o}{\partial w_{f1}}. \tag{2.20}$$

where net_o is the input to neuron o and out_o is its output. Provided the activation function g is differentiable this has a closed solution. The loss with respect to the output weight is easily computable, contrasted with the loss with respects to w_{11}:

$$\begin{aligned} \frac{\partial H}{\partial w_{11}} =& \frac{\partial H}{\partial out_{h_1}}\frac{\partial out_{h_1}}{\partial net_{h_1}}\frac{\partial net_{h_1}}{\partial w_{11}} \\ & \downarrow \\ & \frac{\partial H}{\partial out_{h_1}} = \frac{\partial H}{\partial net_o}\frac{\partial net_o}{\partial out_{h_1}} \end{aligned}$$

The error in w_{11} is dependent on how h_1 influences the output o, creating a backwards dependency. While inference proceeds forward, error correction propagates backwards, a process known as backpropagation [75]. The weights are updated according to (2.16). Using (2.20) as an example:

$$w_{f1}^{i+1} := w_{f1}^{i} - \eta \frac{\partial H}{\partial w_{f1}}. \tag{2.21}$$

The discussion on training concludes with two points:

i The learning rate η can be dynamic, typically decreasing over training to avoid overshooting a local minimum; and

ii The above example trains on a single observation, but to expedite training data often trained in batches. This provides an approximation of the gradient $\nabla \hat{H}$. Calculating the true gradient ∇H across the entire training set would be computationally intractable.

2.6 Large language models

Large language models (LLMs) are advanced probabilistic models designed to interpret and generate natural language. They fall under a broader category of language models that use various statistical methods to mimic human language. The distinction between LLMs and standard language models is informal, primarily reflecting scale. In the 1990s, IBM pioneered research into LLMs with a focus on training models for translation. Brown et al. [7] describes a English to French translator trained on 1.7 million translations with 2.4 billion translation probabilities. Indeed IBM dominated the machine translation world for many decades henceforth. In 2016, Google released their *Google Neural Machine Translation* [98] which used a long short-term memory (LSTM) [33] architecture to address the issue of the vanishing gradient problem in recurrent neural-networks (RNNs) [75]. Whilst the long short-term memory architecture proved successful for Google, the recursive nature of both LSTM and RNN architectures meant that training parallelisation was impossible. These deficiencies motivate the requirement for a more sophisticated architecture. The section begins by showcasing the deficiencies of previous architectures, followed by a solution proposed by Vaswani et al. [92].

A recurrent neural network, a layer function $f(\cdot)$ is applied repeatedly over N steps. This repeated application can be viewed as a DNN with N layers. Figure 2.5 visualises the backpropagation process with the DNN from the previous section but note the equivalent process for an RNN.

Recall $\frac{\partial H}{\partial w_{11}}$ depends on $\frac{\partial H}{\partial w_{f1}}$. When N is large, as in deep RNNs, these dependencies make certain gradients increasingly unstable as they depend on every gradient that came before it. They are prone to exploding (becoming exceedingly large) or vanishing (approaching zero). While the LSTM [33] architecture addressed addressed the vanishing gradient problem it remained susceptible to exploding gradients [38]. Further, these models still required sequential training, training on each word in a sentence one at a time, with the model's state evolving with each successive word. This training scheme consequently weights more recent words as more important regardless of their semantic meaning in the sentence proving inept for natural language.

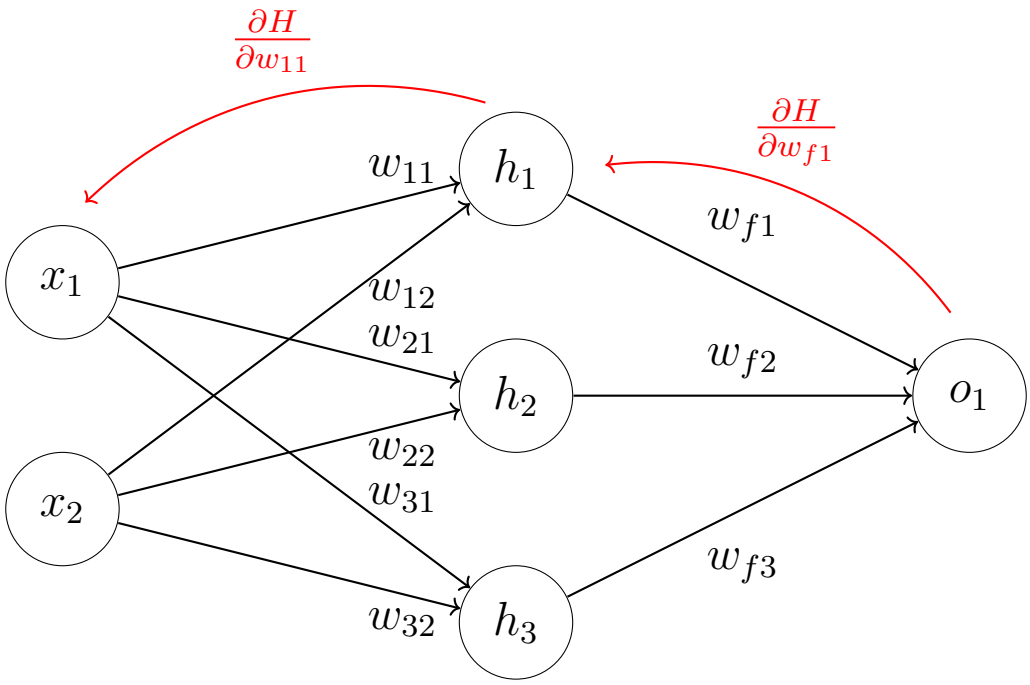


Figure 2.5: The backpropagation algorithm visualised.

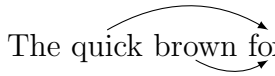
Vaswani et al. [92] introduced the transformer architecture, enabling the training of large models in parallel, eliminating the need for recursion. This architecture employs the concept of attention, an already-established mechanism surveyed by Soydaner [85], who highlights the efforts to model the psychological notion of attention in a machine learning architecture. As will be demonstrated, the transformer allows for significant improvements in both training speed and quality of prediction. This ability to 'attend' to a sentence's context makes transformers a promising candidate for use as a dynamic probabilistic model in arithmetic coding.

2.6.1 Transformers and attention

Transformers are a machine learning architecture that supports parallel training on text by leveraging the concept of 'attention'. In machine learning, attention enables individual units of data (tokens) within a context window to be influenced by the meanings of surrounding tokens [92].

Although attention as a concept in machine learning may seem opaque, its aim is to emulate humans' natural ability to understand sentences. The following example draws on the one provided by Sanderson [78], introducing attention using a common example sentence.

Example 7 (Foxes and dogs)**.**



The example phrase allows for easy identification of the characteristics of the fox as both 'quick' and 'brown,' and previous examples suggest 'dog' as the next word in the sentence. Such inference is straightforward, but training a computer model to do the same presents a challenge. In this example, each word is treated as an individual token, though the exact tokenisation of a sequence will depend on the model's design.

Each token is represented by an embedding vector $\mathbf{E} \in \mathbb{R}^E$, encoding its semantic meaning (and position in the sentence) as a point in $\mathbb{R}^E$. The aim is to move the embedding vector of 'fox' such that the idea of idea of 'brownness' and 'quickness' has been imbibed into the embedding of 'fox'. In a trained language model, the query vector $\mathbf{Q}$ of each token interacts with the key vectors $\mathbf{K}$ of preceding tokens via a dot product. Tokens with strong contextual relationships yield a higher dot product, thereby **attending** to one another. Ideally, this structure enables the model to recognize that 'quick' and 'brown' should attend to "fox."

Each token also has a trainable value vector $\mathbf{V} \in \mathbb{R}^E$, which represents the direction in $\mathbb{R}^E$ should shift to transfer its meaning to the token it is attending to, with its values scaled by the attention score of the interaction. For instance, if 'brown' strongly attends to 'fox' with dot product 0.9, then $\mathbf{E}_{fox}$ is moved by $0.9 \times \mathbf{V}_{brown}$. In practice, attention scores are normalised to form a probability distribution using the softmax function $\sigma : \mathbb{R}^K \to (0,1)^K$, applied element-wise to operation to the vector $\mathbf{x}$ [84]:

$$\sigma(\mathbf{x})_i = \frac{e^{x_i}}{\sum_j x_j} \tag{2.22}$$

The formula for attention [92] is written as:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \sigma\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_q}}\right)\mathbf{V} \tag{2.23}$$

In Equation 2.23 there is no concept of a tokens' position within a sentence. As a result, the example sentence is mathematically indistinguishable from 'brown The quick the lazy over jumps'. Vaswani et al. [92] addressed this by proposing the addition of a positional vector unique to each tokens position. This approach enables the model to capture not only meaning about the word but also the contextual influence of its position within a sentence. For a token's embedding in E-dimensional space, the following positional encoding vector is added to E, as proposed in the original paper:

$$\begin{aligned} PE_{(pos,2i)} &= \sin(pos/10000^{2i/E}) \\ PE_{(pos,2i+1)} &= \cos(pos/10000^{2i/E}) \end{aligned} \tag{2.24}$$

Here, *pos* denotes the token's absolute position within the sentence, and i represents the index of the embedding vector, with even i using the sin function and odd i using the cos function. This approach guarantees each position has a unique embedding for contexts as long as 10 000.

Figure 2.6 provides a mock illustration of token embeddings in $\mathbb{R}^2$. The real embedding dimension E is usually much higher and the precise semantic meaning of each dimension are often not fully interpretable. The figure serves to visually represent how attention may learn and apply meaning.

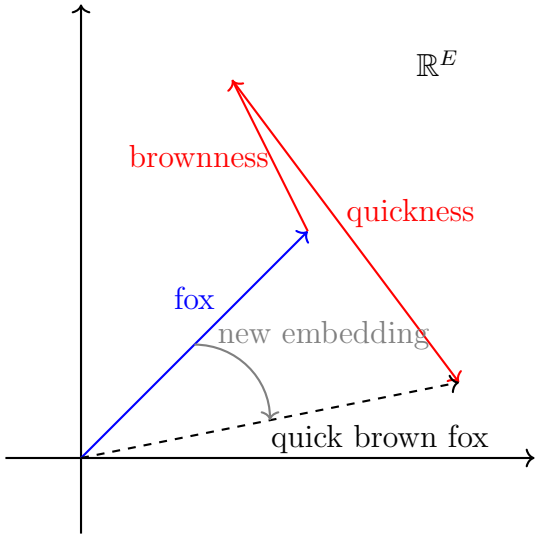


Figure 2.6: A toy example of the embedding space projected into $\mathbb{R}^2$.

The interaction of 'quick' and 'brown' with 'fox' enriches 'fox' with these attributes, which in turn, passes its more enriched embedded meaning to the final token, 'lazy.' The application of attention allows meaning to carry through the sentence. To predict the next token, the final token's embedding (containing the sentence's full context) is transformed into the token space via an unembedding matrix $\mathbb{R}^{t\times E}$ [70] yielding a vector $\mathbf{x} \in \mathbb{R}^t$. Here, t is the dimension of all model tokens, with each x_i giving the probability of token i being the next.

Example 7 illustrates our intuitive understanding that adjectives like 'quick' and 'brown' attend to nouns like 'fox'. In reality when training, the model defines $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ without prior linguistic assumptions and aims solely to minimise its loss function. Indeed, these vectors represent a single attention head. In complex models, multiple heads operate in parallel, each modifying the embedding vector in a unique way. Furthermore, these models have multiple layers, with each attention layer usually followed by a 'feed-forward' layer and normalisation [92], as shown in Figure 2.4.

The task of defining a loss function involves aligning the model's predicted word distribution with real sentence structures rather than predicting a single correct word. For instance, in Example 7, the model ideally predicts 'dog' as the next word, but another plausible option such as 'cat' isn't strictly incorrect. A commonly used loss function for this purpose is the

cross-entropy function [45], which compares the actual distribution p from training data[6] with the model's predicted distribution q. As noted in section 2.4, cross entropy was defined as Equation 2.14 and was shown to minimise the required bits for representation as its value was minimised. If then the cross-entropy is chosen as models loss function, a model trained for inference necessarily trains it to be an adept compressor.

A model's ability to predict is in sympathy with its ability to compress.

A transformer can be trained on WiFi data to produce an output probability distribution which can then be used as input for an arithmetic coder as a compression scheme. Decompression is achieved by reversing this process, auto-regressively feeding the most recent decompressed token back into the model. The data which is provided to the transformer is known as the 'context window' (which was given as w in the `gzip` example.) The time complexity inference is quadratic in the size of this context window [92] proving to be a constraint in practical compression applications. Additionally, the constants such as the embedding dimensions and indeed the number 'heads' to use within a model are known as hyperparameters, to distinguish them from the parameters which hold the information a model has learned.

2.7 IEEE 802.11ax

The IEEE 802.11ax standard [43, 95, 25] defines the physical layer implementation of WiFi 6, specifying how data sequences are physically generated and transmitted through a medium [81], in this case, free space, where signals propagate as perturbations in the electromagnetic field, commonly termed radio frequency (RF) data. This thesis addresses the compression of IEEE 802.11ax (WiFi) data in its RF form, limiting implementation specifics to those that affect signal waveforms. Whilst WiFi RF signals vary in exact modulation scheme, this study focuses on signals generated within a single modulation and coding scheme, detailing the relevant taxonomy below

Devices connected to a WiFi node must synchronise to accurately tune their local oscillators before data transmission can occur. Additionally they require preamble data to interpret the payload correctly [43]. The 802.11ax standard specifies twelve modulation and coding schemes (MCS), from `MCS0` to `MCS11`, which offer varied data rates to adapt to noise levels in the environment. For instance, `MCS0` uses binary phase shift keying (BPSK) with a 1/2 coding rate suited for high-noise environments limiting the data rate to 72 Mbit/s, whereas `MCS11` uses 1024-QAM (Quadrature Amplitude Modulation) with a 5/6 coding rate, yielding data rates up to 1201 Mbit/s.

This work uses `MCS0` which uses BPSK. BPSK modulates the phase of a carrier signal based on binary input data, with each bit dictating the carrier's phase (Proakis and Salehi [71] p.

[6] This training data could be found by considering extremely large corpa of text such as the entire internet.

102). For instance, a binary `0` leaves the carrier phase unchanged, while a binary `1` induces a 180-degree phase shift encoding information with the phase of the carrier.

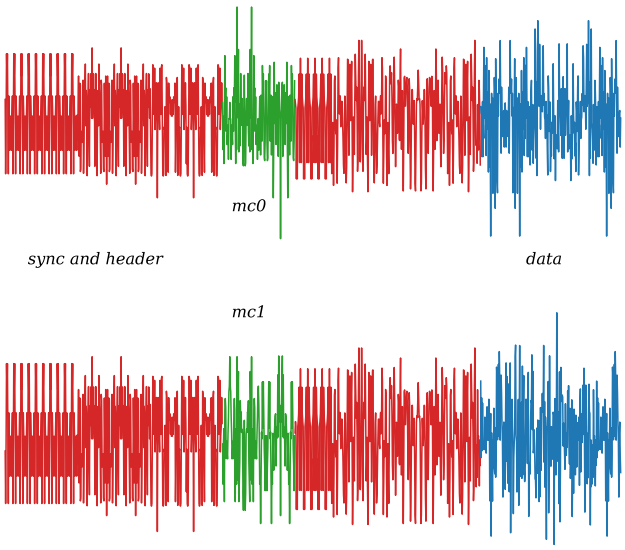


Figure 2.7: Taxonomy of the real part of a WiFi 802.11ax signal with MCS0 and MCS1

In Figure 2.7, the generation of two WiFi signals with a one-byte payload is shown, one configured as `MCS0` and the other as `MCS1`. The preamble adapts to reflect the change in modulation scheme. The payload (in blue) is random in both cases.

2.8 Relevant work

2.8.1 Language Modelling is Compression

This work is based on Delétang et al. [20], a paper by Google's DeepMind, and uses a modified version of their code to implement our the compression schemes described in this work. The work demonstrated impressive compression rates by using existing models such as LLaMA 2 [21, 90] to achieve state-of-the-art compression in multiple domains such as image, text, and audio. Although Delétang et al. [20] did not explicitly compress WiFi data or any RF, the compressors were tested against a LibriSpeech corpus (a collection of many hundreds of hours of English speech) [68], which while not RF data, shares characteristics such as being homoscedastic, zero-mean, and time-series in nature.

Results from their study show poor compression rates audio data for small transformers models not explicitly trained on such a modality ('base transformers'), however impressive compression rates were achieved with large pre-trained models such as LLaMA 2 7B and

Chinchilla [35] achieving a compression rate of 23.1% for the LibriSpeech data, compared to a rate of 36.4% using `gzip`. This work will use LLaMA 2 7B Chat as it is open source and is small enough for use on available hardware.

2.8.2 LLaMA as a pattern recogniser

Recall that a model's predictive accuracy directly influences its compression efficacy. Here, the focus is on foundation models' (specifically LLaMA 2) capabilities as general-purpose pattern recognisers [60]. The LLaMA 2 family comprises open-source transformer models developed by Meta, available in various sizes (7B, 13B, 34B, and 70B), where the number denotes trainable parameters in billions. For this study, LLaMA will refer to the LLaMA-2 7B Chat model optimised for conversational tasks.

LLaMA, trained on publicly available internet text data [90], was not explicitly designed for time-series recognition. Delétang et al. [20] however observed that due to the extensive scope of its training data, some time-series data might have appeared in the training set. Still, LLaMA's strong performance in compressing the LibriSpeech data suggests capabilities beyond incidental exposure in training.

Instead, we view large foundation models as advanced pattern recognisers. Gruver et al. [29] proposes that language models function as zero-shot time-series forecasters, presenting a framework for time-series prediction by tokenising sequences of data as a string of numerical digits and predicting the next strings of numbers. The paper invokes Goldblum et al. [27] which offers a novel perspective on the *no free lunch* theorem, initially proposed by D. H. Wolpert [18], which claims that across random tasks, no model is universally optimal. They argue that since real-world data often exhibits autoregressive characteristics, models can be tailored to perform well on naturally structured data rather than completely random data. This notion is further discussed in Shalev-Shwartz and Ben-David [79] sec. 5.1.

Rasul et al. [72] considers the fine-tuning of pre-trained LLaMA weights for time-series to improve predictive accuracy by exposing the model to time-series data. Due to computational constraints, this work does not fine-tune LLaMA but instead utilises the pre-trained LLaMA 2 model with an appended feed-forward 'head', trained on WiFi data using the `transformer-heads` package [10].

2.8.3 Tokenisation choice

ASCII [24, 44] (American Standard Code for Information Interchange) is a standard character encoding scheme where the Latin alphabet and other common characters are represented in a 7-bit range.

A key difference with the base transformer models and LLaMA is the choice of tokenisation strategy. In the base transformer model each ASCII byte (letter) functions as an individual token, representing an individual measurement of RF. As LLaMA was designed for and trained on natural language, it treats chunks of text as tokens not dissimilar to Example 7, grouping meaning within words and sub-words.

Delétang et al. [20] recognised this and remarked that the grouping of multiple bytes into discrete tokens is itself compression. This approach, they noted, increases the total number of tokens which makes it more difficult to reduce the entropy of the output probability mass function (PMF) to be used with the arithmetic coder. For instance, using each ASCII character as a token requires 128 tokens, as all ASCII characters are represented by 7 bits. Contrast this to a tokenisation scheme where one token is two characters resulting in $128^2 = 16384$ tokens. Although each token may carry more meaning, the entropy of the PMF is substantially harder to reduce.

2.8.4 Scaling and positional embedding

Radio frequency data can vary by scale as transmitters and receivers can be placed arbitrarily. Because RF at different scales ultimately represents the same data we aim to produce a compressor that can work invariant to the scale of the incoming data. Previous papers have outlined the difficulty with time-series prediction with different scales, namely Ansari et al. [2] and Salinas et al. [77]. Ansari et al. [2] addresses different scales and offsets by proposing the affine transformation on the time series $\{x_i\}$: $x_i = (x_i - m)/s$ varying s and m to restrict data to a set range. This is inappropriate for compression which cannot distort the original data.

When initially proposing the attention mechanism, Vaswani et al. [92] provided a method to encode positional information within a sentence using sinusoidal absolute embeddings, creating a unique encoding based on the token's position. Following this, Zhang et al. [102] conducted an empirical study comparing various positional embeddings, including absolute, relative, and even none, finding that relative positional encoding offers improved performance by embedding information on the position of tokens in relation to one another.

Su et al. [86] introduced rotary positional embeddings, a method that encodes token distance directly within the attention mechanism itself. This is particularly relevant for RF data, where dependencies are sequential rather than absolute. Relative positional encoding cannot be embedded within a single token alone, as a token's position only holds meaning in relation to its distance to others. The approach augments attention scores by modifying them based on token distances.

Writing f_q and f_k as the functions that retrieve the query and key vectors of some embedding vector $\mathbf{x}_m$ at position m, the attention mechanism must be of the following form:

$$\langle f_q(\mathbf{x}_m, m), f_k(\mathbf{x}_n, n)\rangle. \tag{2.25}$$

They propose a solution to this equation as:

$$g(\mathbf{x}_m, \mathbf{x}_n, m-n) = \Re[(\mathbf{W}_q\mathbf{x}_m)(\mathbf{W}_k\mathbf{x}_n)^* e^{i(m-n)\theta}], \tag{2.26}$$

taking $*$ to mean the complex conjugate. Here, $\mathbf{W}_q\mathbf{x}_m = \mathbf{Q}$ and $\mathbf{W}_k\mathbf{x}_n w = \mathbf{K}$. The intuition is that the absolute position is encoded as an $m\theta$ rotation and that, regardless of absolute

offset, the inner product of the same tokens equally far apart will be the same regardless of their absolute offset, i.e.

$$e^{i((m+k)-(n+k))} = e^{i(m-n)}.$$

The specific implementation from the paper computes:

$$f_{\{q,k\}}(\mathbf{x}_m, m) = \mathbf{R}^d_{\Theta,m}\mathbf{W}_{\{q,k\}}\mathbf{x}_m \tag{2.27}$$

where

$$\mathbf{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix} \tag{2.28}$$

Where a predefined Θ controls the schedule for θ_i which each rotate some two-dimensional segment of the initial embedding. Here d refers to the embedding dimension.

2.9 Summary

Data representation is formally grounded in coding theory, a branch of information theory, which allows different codes to serve various functions. For compression, coding aims to minimise the expected bit-length of data by quantifying its compressibility through entropy, a theoretical upper bound established by Shannon. Practical compression can improve by discovering more sophisticated relationships using the mutual information found in non-random data. A mismatch in model and reality will increase the entropy and thus the required bits for coding. This mismatch may be quantified by cross-entropy. Transformers, which learn data dependencies by minimising cross-entropy during training, are therefore capable compressors. Foundation models like LLaMA achieve competitive compression without domain-specific training. We hypothesise however that training a model specifically on WiFi data will improve its compression in this domain, though likely at the cost of performance in other contexts.

Chapter 3

Methodology

3.1 Introduction

This methodology builds on the work of Delétang et al. [20] and utilises their codebase. We outline the training and testing of custom transformers ('base transformers') as well as the development and evaluation of a modified LLaMA with an appended 'head' ('headed LLaMA') as well as the evaluation of LLaMA itself. The methodology is structured in three parts:

1. Section 3.2 to section 3.4 describes the process of generating and preparing data, followed by a detailed explanation of how transformer models are integrated with an arithmetic coder for compression and decompression, along with the specifics of the training and evaluation workflow.
2. Section 3.5 to section 3.6 details the distinct training strategies and hyperparameter selections for our models. The concepts described here should be treated as distinct, to be unified at the close of the chapter.
3. Section 3.8 and section 3.9 close the chapter with an explanation to the measurement of the compression rate and how the regimes described in the above sections are combined to create our compression models.

3.2 Data synthesis and preparation

Owing to the scarcity of publicly available WiFi 802.11ax data, training data was synthesised in MATLAB [89] using the WLAN Waveform Generator module. The chosen parameters for generation is described in Table 3.1. Recall that `MCO` prescribes a BPSK encoding. A packet length of 512 bytes was used to ensure the model trained primarily on payload BPSK data rather then overfitting to the deterministic pilot and preamble. The data generated for testing is described in Table 3.1.

Parameter	Value
MCS	0
Number of Packets	16
Packet Length	512

Table 3.1: Training and testing parameters.

The IQ data was scaled to $[-32768, 32767]$ to utilise the maximum range of the `int16` to provide maximal variation in scale. Only the real component of the signal is considered with the assumption (though untested) that similar results would be achieved with the imaginary part. Each file contained 24 640 real data points. 1000 files were generated for training and 10 for testing.

The data was then downsampled to `int8` by removing the least significant eight bits. 128 was added and the data was considered as unsigned `uint8`. Henceforth references to the data will refer to this `uint8` data. A discussion of this choice is found in section 5.3. To establish a compression baseline, the probability mass function (PMF) of the training data distribution was used as an unconditional distribution in an arithmetic coder for use over the test files ('naive arithmetic coding').

As LLaMA was trained on text [90] the input data required ASCII formatting. Given ASCII's limitation to $[0, 127]$, only the first seven bits were compressed using the model, leaving the eighth bit uncompressed. This last bit was reintroduced when calculating the size of the compressed output, as illustrated in Figure 3.1. This methodology is specifically required for the tokenisation process in LLaMA; however, it was also applied to the base transformers for sake of comparability.

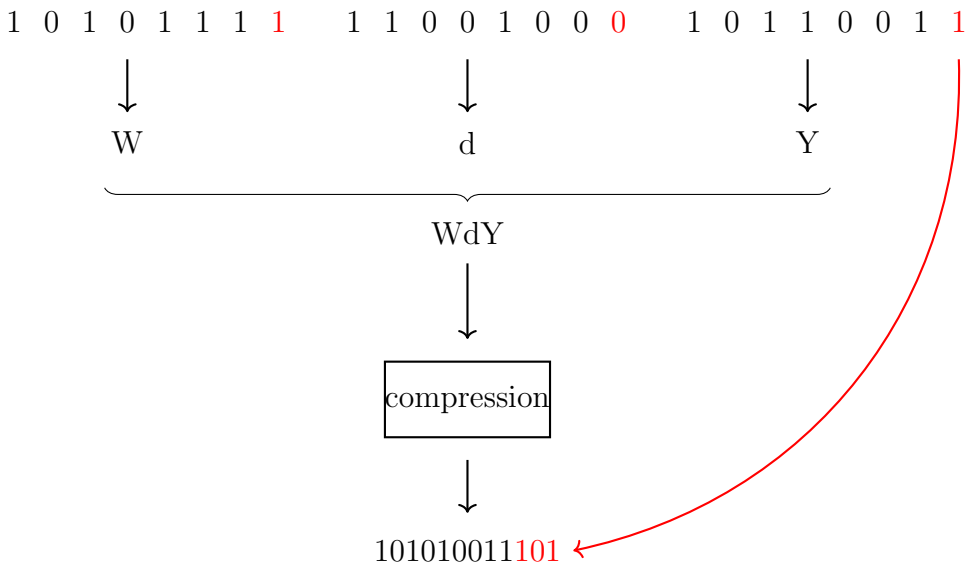


Figure 3.1: Extracting ASCII from `uint8`, compressing, and appending the lost bits back. The compression rate is $\frac{9+3}{3\times 8} = 0.5$

3.3 Compression and decompression

Compression To compress WiFi data using a transformer, a segment of the data was taken as the context window, with the final symbol set as the target for compression. The data was tokenised into ASCII which were used as input to the model. After processing, the final token's unembedding was normalised to form a probability distribution. This distribution determined the token probabilities, where the target was the ASCII character immediately following the context window. This distribution is provided to the arithmetic coder for use in compression. After compressing the token, the context window shifted one character to the right, maintaining a fixed length. Figure 3.2 illustrates this process.

Decompression Decompression required a causal approach, using only previously decompressed data to generate probability distributions for use with the arithmetic coder. Once this PMF was produced, the decompressed code-word was applied to this distribution in an arithmetic coder until a unique interval was found corresponding to the token to be decompressed. This symbol was extracted and appended to the output, and the process was repeated. This process is the direct mirroring of the compression process, and the PMFs generated must be identical for lossless decompression. In initiate the process, a `0` token was used as the starting input. Padding tokens were employed to ensure the input to the model matched the input width that the model used for compression.

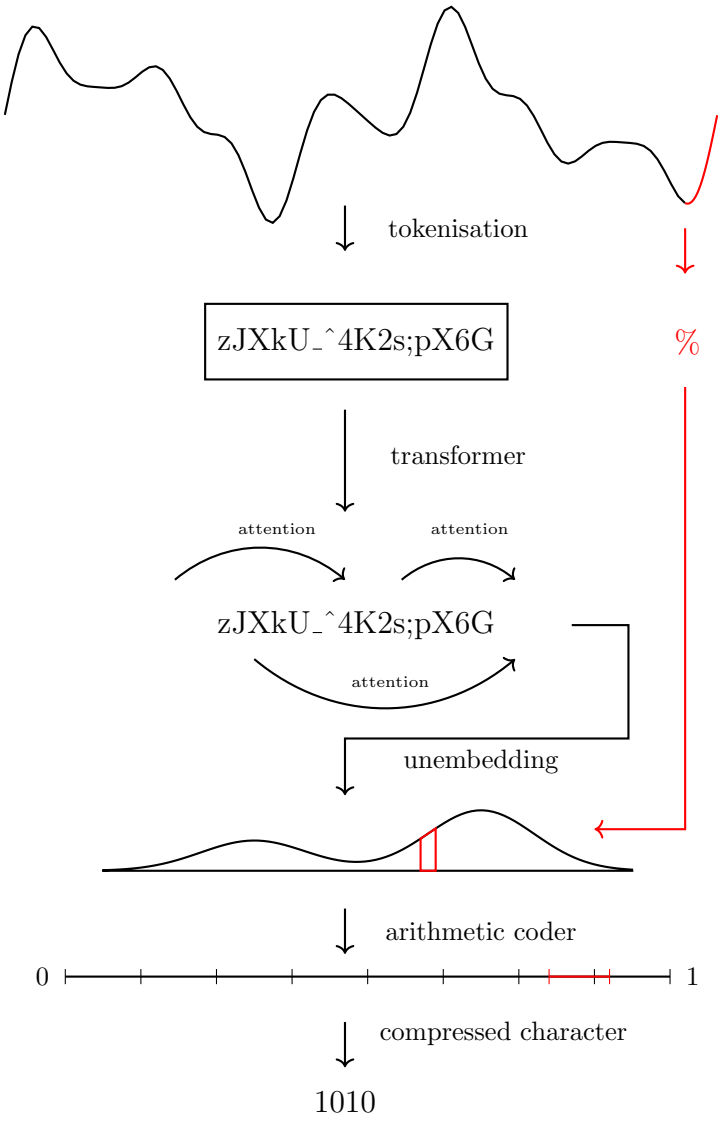


Figure 3.2: The compression process. The black waveform represents the RF context, while the red segment denotes the future data to be compressed. The context is tokenized and processed by the transformer using the attention mechanism. The final token is unembedded to obtain an estimated probability distribution for the next token, which the arithmetic coder uses for compression.

3.4 Implementation

A command-line interface was developed in Python to automate the training and testing process over various schemes. Model configurations were predefined in a YAML file which was read in the Python environment, allowing for training with specific hyperparameters, learning schedules, batch sizes, iterations, and other adjustable parameters. Four Nvidia RTX 3090 GPUs were used to accelerate both training and inference. Model evaluation was conducted on generated WiFi data that was outside the training set. A block diagram of the training setup is provided in Figure 3.3

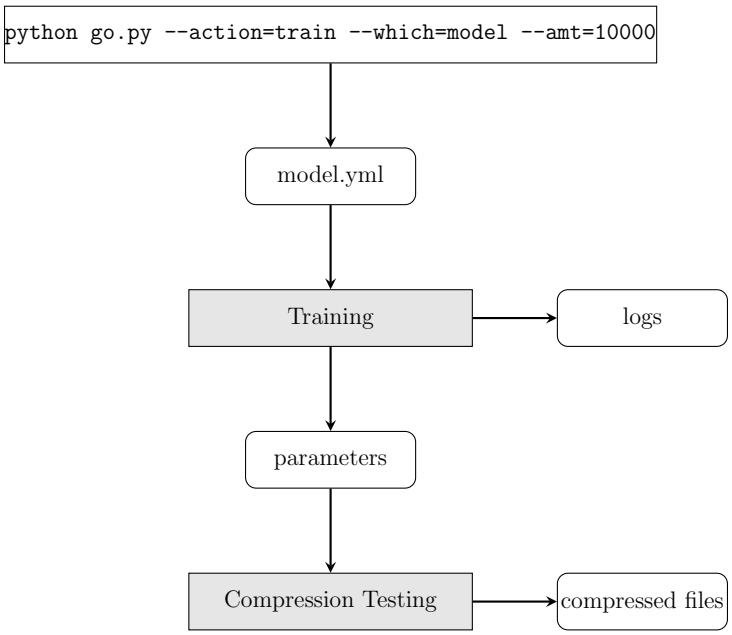


Figure 3.3: Training and testing process. Processes in grey required the graphical processing unit (GPU).

3.5 Hyperparameter choices

Training large language models is inherently complex, with optimal hyperparameter choices often lacking clear rationale. Documenting key training decisions, as in the DALL-E Mega journal [19], helps capture these choices. This work used hyperparameters and training schedules used in Delétang et al. [20] as a starting point which was iteratively improved upon. The transformer comprised of layers with a self-attention block, linear transformations, a Gaussian error linear unit (GELU), and final layer normalisation. These tunable parameters along with training settings are detailed in Table 3.2 which outline this paper's hypotheses and ranges tested. This section discusses the embedding dimension, context lengths, and concludes with a summary of other selected hyperparameters.

Hyperparameters and Training Choices	Typical Range	Hypotheses
Training steps	1e6 - 5e6	More training will result in a higher accuracy on the training data but may result in overfitting. Some sources say the opposite and recommend more training for sake of generalisation [34].
Epochs	1- 3	More epochs might allow for gradient descent to escape sub-optimal minima through a ramp-up stage. More runs over the data means better convergence [82].
Learning rate	1e−3 - 1e−7	A higher learning rate may result overshooting optimal minima but also allows for escaping suboptimal ones. We hypothesise the schedule matters more than the learning rate [82].
Batch size	16 - 128	We begin with small batch sizes to somewhat suboptimal gradients to increase the randomness at the start which may lead to better convergence later [48].
Embedding Dimension	32 - 128	A larger embedding dimension may learn more nuanced relationships between tokens but may result in overfitting [92].
Number of layers	4 - 16	More layers may be able to learn more complicated dependencies but may take longer to converge.
Number of heads	4 - 16	More heads may allow the model to learn more hidden relationships but may take longer to converge [92].
Widening factor	2 - 8	A larger widening factor means more hidden neurons in the feed-forward layer of the attention head. We hypothesise the model may be able to learn more hidden relationships but may take longer to converge [47].

Table 3.2: An overview of model hyperparameters and training choices. The 'typical range' column is an indication of the choices explored in this study.

3.5.1 Embedding vector

Transformers are effective as they transfer ideas between tokens which guide a tokens embedding in $\mathbb{R}^E$ to a more contextually rich meaning. In language words are semantically similar if they share a similar meaning. We propose the following definition for time-series.

Definition 1 (Semantic Similarity)**.** In the context of time-series data, semantic similarity is proportional to the absolute numerical difference between the two values.

The hypothesis is that if the model learns a regime of semantic similarity through training, it will show increased resilience to noise and scaling. For instance, if noise shifts a value k to $k+1$, this should minimally affect the predicted PMF as these tokens should share similar meanings. To test the impact of embedding dimension on a model's ability to learn this regime, the models described in Table 3.3 were tested over 1 000 000 training runs with training data fixed at unity scale, and their compression rate was evaluated. Hyperparameters not mentioned in the table were fixed. Figure 3.4 visualises the compression rate of these experiments relative to their compression rate at unity-scaled testing data (the scale on which they were trained).

Parameter	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5
Embedding Dimension	128	64	32	16	8
Widening Factor	4	8	16	16	16

Table 3.3: Embedding size experiment run.

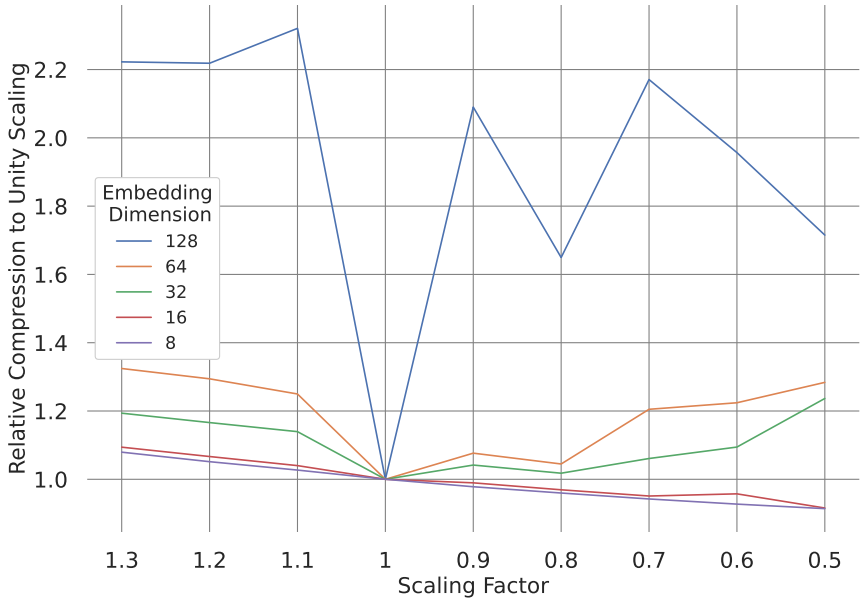


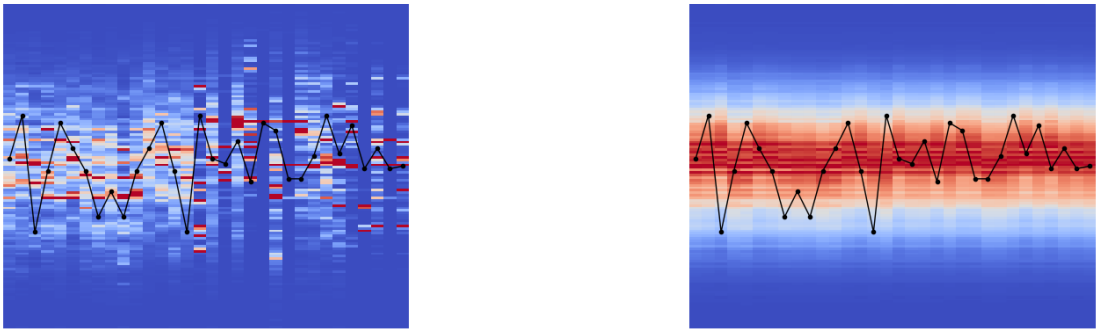
Figure 3.4: Relative Compression Rates with Different Scales

Two insights were taken from Figure 3.4 about how model training converges:

i At too high an embedding dimension the model does not generalise well to different scales and performs poorly.

ii At too low an embedding dimension the scale was inconsequential. In fact, the compression rate was linear in the scaling factor.

Figure 3.5 visualises a random segment of the RF payload being compressed, superimposed over a heatmap representing the PMF generated for each symbol. Each column corresponds to a specific symbol's PMF under which it was compressed. Compression improves when data points align with the 'hotter' (red) cells, while a 'colder' (blue) cell indicates a lower value in the PMF. The closer the real data (black) aligns with the hot cells of the PMF, the more efficient the compression for that datum. In addition, the more confident the the model is about the true distribution the more condensed the hot areas of the heatmap will be, corresponding to a lower entropy.



(a) Embedding dimension 128.

(b) Embedding dimension 8.

Figure 3.5: Compression heatmap for embedding dimensions 128 and 8. The data to be compressed is represented by the solid black line and the PMF produced by hte language model is represented as a heatmap where red (hot) indicates a stronger prediction and blue (cold) represents a weaker prediction.

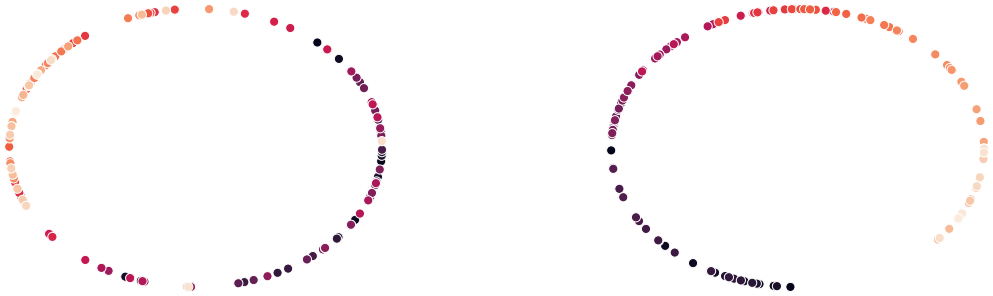
Figure 3.5 demonstrates the embedding vector dimension affecting the convergence of the model by either:

i being high enough that the model will try to learn dependencies to approximate $\mathbb{P}(x_i \mid \mathbf{x}_{j<i})$ or;

ii will try to approximate the unconditional distribution of the underlying training data.

It was found that embedding dimension was not the only hyperparameter choice that affected model convergence. This would will refer to the two convergence regimes described above as regime 1 and regime 2.

The embedding dimension was selected to capture semantic meaning (Definition 1). To visualise the positional relationships within the embedding space, we applied principal component analysis (PCA) to plot the embedding space in two dimensions (Figure 3.6). Colours were assigned as a gradient spanning the alphabet range, $[0, 127]$. This method reveals the geometry of the embedding values, revealing how well they encode semantic relationships based on their relative positions. We would like semantically similar tokens to be geometrically closer.



(a) $d = 8$ (b) $d = 128$

Figure 3.6: Principal component analysis in two dimensions of two different choices of embedding vector size. The gradient from dark to light represents the range $[0, 127]$

Figure 3.6 shows that a higher embedding dimension $d = 128$ achieves a smooth gradient where tokens with similar values are positioned closely, with a noticeable gap between the minimum and maximum tokens. In contrast, $d = 8$ captures only a general separation of far-apart tokens but fails to exhibit the same smooth relationship.

The experimental trade-offs indicated that a low embedding dimension ($d = 8$) prevented the model from learning contextual dependencies, instead producing results close to the unconditional training distribution[1]. Additionally, the model failed to develop meaningful token embeddings. Conversely, an embedding dimension of $d = 128$ enabled the model to learn meaningful embeddings and utilised context in generating its PMFs, yet led to poor generalisation beyond the training scale. Thus, an embedding dimension of $d = 64$ was selected for all experiments unless otherwise specified, striking a balance between these extremes.

3.5.2 Context length

To assess the effect of context length on compression rate, we conducted the experiment detailed in Table 3.4 over 2 000 000 training runs. All other hyperparameters were held

[1]This was verified quantitatively using the Kullback-Leibler divergence relative to the training data's PMF.

Parameter	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6
Context Window	16	32	64	128	256	512

Table 3.4: Context window experiment runs.

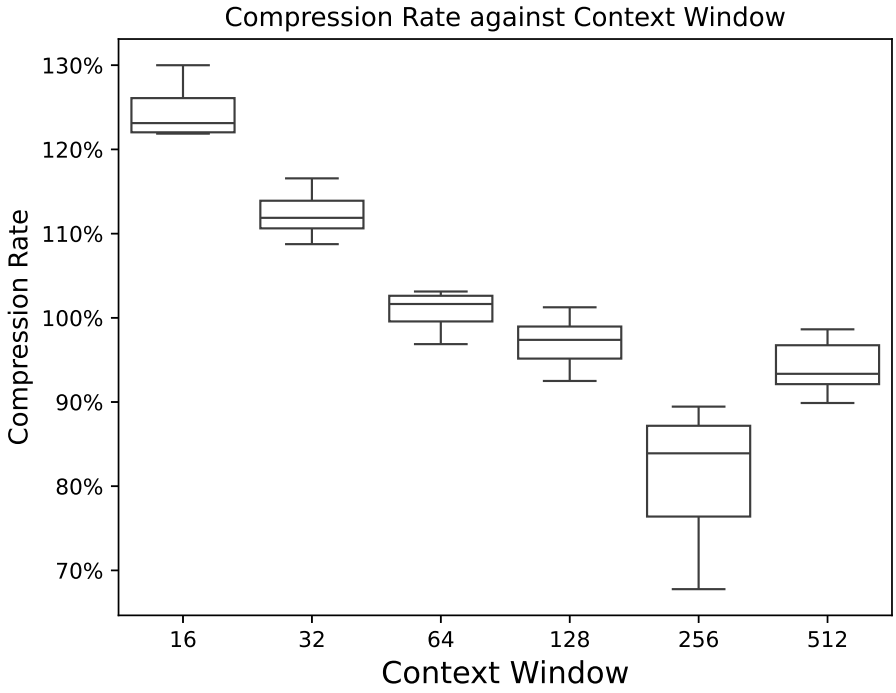


Figure 3.7: Compression rate over various context windows. We observe a larger context window yields an improved compression rate however this relationship breaks down at too large a context.

constant. The compression rate was evaluated using 1000 random selections of context windows across 10 plotted in Figure 3.7.

Figure 3.7 demonstrates that larger context windows improved compression by capturing more mutual information from preceding tokens, thereby reducing the entropy of the next token's PMF (refer to equations (2.10) and (2.13)). This relationship was observed to break at a context window of 512. We hypothesise that larger context windows require more training to properly take advantage of the dependencies within data. We also hypothesise that the dependencies in WiFi are diminishing at too large a context window owing to the random nature of the payload. Since inference is quadratic in the size of the context window [92], 256 was selected as a trade-off between inference time and the amount of mutual information which could be extracted from the data. This limitation did not apply to LLaMA-based models, where their inductive ability depends on what they can learn from the context window provided. As this is their only exposure to the data, the context window was maximised given hardware constraints, using 1024 tokens.

3.5.3 Other hyperparameters

The number of heads within a transformer block were varied between 4 and 8 matching Delétang et al. [20], primarily chosen based on small experiments that demonstrated their efficacy. A similar treatment was given to layer count which varied between 4 and 16. The widening factor which determined the size of the linear layers inside the transformer block was fixed at 4. The batch size was chosen based on the learning schedule described later in this work, varying from 16 to 64. The hyperparameter choices for every model are found in Appendix A.

The following section describes specific training regimes chosen to improve generalisation ability across varying noise and scales. Their exact implementation in combination with hyperparameter choices is given in section 3.9.

3.6 Transformer training models

3.6.1 Naive training

Naive training involved training the model on a single scale without added noise. This approach was hypothesised to lead to overfitting, with the model likely to memorise specific sequences rather than the underlying patterns in BPSK.

3.6.2 Noisy training

Chi et al. [11] proposed an *RF-Diffusion* model using an attention-based diffusion[2] block to generate synthetic RF data. Diffusion models on increasingly noisy data, learning to iteratively reverse this process to recover the original signal [58]. While Chi et al. [11] aimed to generate synthetic data, this work adopts a similar forward destruction process to introduce noise in training in both time and frequency domain.

Noise in training has been empirically shown to improve generalisation [5, 63], and is critical to our work given the noiseless nature of synthetic data and the inherent randomness in real WiFi signals. White Gaussian noise (AWGN) was added to the signal and frequency blurring was applied to better adapt the models to learn the underlying BPSK characteristics rather than overfitting on the training data. We hypothesise by providing randomness in the training data the transformer will generalise better to new data, and be more robust against different noise levels (but not necessarily different scales).

Frequency blurring To introduce frequency domain noise, a Gaussian kernel was created with standard deviation chosen uniformly at random by $\mathcal{U}[0, 0.1]$. If this standard deviation is below 0.015, frequency blurring is skipped. Otherwise, the kernel was applied by convolving it with the FFT of the training data, followed by calculating the inverse FFT and rescaling to the original range.

[2]The same diffusion as in stable diffusion.

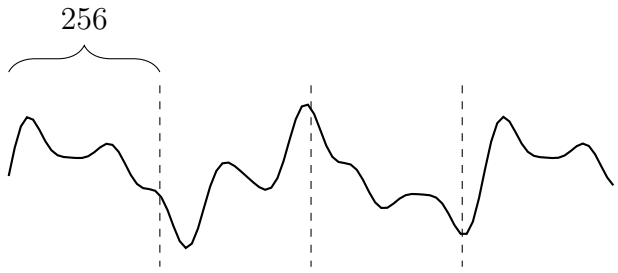
Temporal Noise AWGN was added to the batch with a standard deviation chosen uniformly at random between $[0, 3]$. The training is described in Algorithm 1

Algorithm 1 Noisy Training

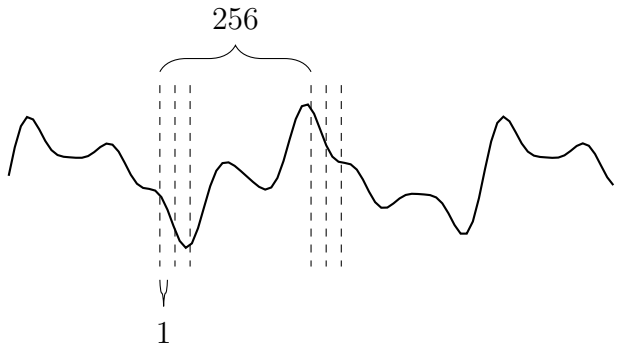
$\mathbf{d} \leftarrow$ batch
for $i \leftarrow 1$ to size($\mathbf{d}$) **do**
  $f \leftarrow \mathcal{U}[0, 0.1]$
  if $f \geq 0.015$ **then**
    $d_i \leftarrow \mathfrak{F}^{-1}(\mathcal{G}_f * \mathfrak{F}(\mathbf{d_i}))$ ▷ $\mathcal{G}_f$ is the Gaussian kernel with SD f
  end if
end for
$\mathbf{w} \leftarrow \mathcal{U}[0, 3]_{\text{size}(\mathbf{d})}$
$\mathbf{n} \sim \mathcal{N}(0, \mathbf{w})$
$\mathbf{d} \leftarrow \mathbf{d} + \mathbf{n}$

3.6.3 Training stride

DeepMind's training methodology divided the LibriSpeech data into chunks with strides matching the context window length, an approach we refer to as block-strided data (see Figure 3.8a). To explore the impact of stride length, we also used a unit-strided data method (Figure 3.8b) which considered training data over a sliding window.



(a) Block-strided Data (256-stride)



(b) Unit-strided Data (1-stride)

Figure 3.8: Single-sided versus block-strided data.

3.6.4 Single-scale training

RF data appears in varying scales, and achieving scale invariance in compression is a core goal of this thesis. We hypothesised that learning compression across multiple scales introduces additional complexity, given the increased information the model must internalise. We propose an algorithm that trains and compresses data to be scale invariant without augmenting the original data. To motivate why a recoverable scaling function is necessary we gesture to a naive configuration where trained data are normalised to a fixed range so that the maximum and minimum value of every chunk is 0 and 255 respectively. To compress a new chunk, it must be scaled to the same range so that the training and compressing scale are aligned.

We find however that there is no reversible scaling functions over the integers $\mathbb{Z}$. This prevents the solution fo storing a scaling factor and recovering the original data by taking its inverse. Figure 3.9 illustrates the irreversibility of scaling in `uint8`. The scheme is lossy.

To overcome this we propose the following algorithms:

Training Training data was normalised such that the maximum value was 255 and the minimum 0. The models were then trained on these chunks. This process is described in Algorithm 2.

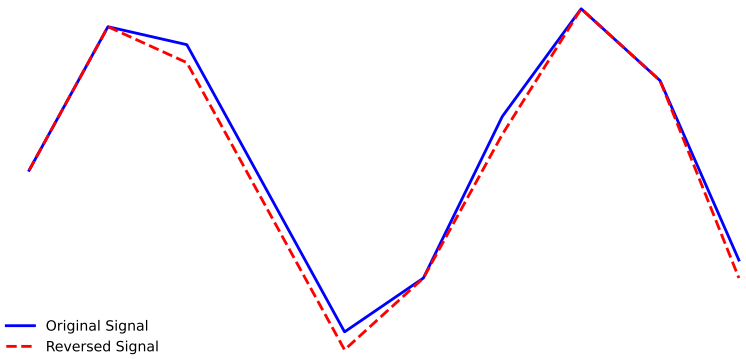


Figure 3.9: Demonstration of the irreversibility of scaling in `int8` by taking $1.731 \times \sin(x)//1.731$.

Algorithm 2 Single-scale Training

```
d ← batch
for i ← 1 to size(d) do
    m ← max(d_i)
    n ← min(d_i)
    for j ← 1 to size(d_i) do
        d_ij ← [(d_ij − n) × 255] // (m − n)
    end for
end for
Train(d)
```

Compression To align the data with the model's training scale, it was necessary to scale the data accordingly. As shown in Figure 3.9, this scaling process is irreversible; however, by extracting the probability distribution produced by the transformer with scaled input, and scaling the distribution back down to the size of the original data, we may losslessly compress. This process is illustrated in Figure 3.10 and the algorithm is defined in Algorithm 3.

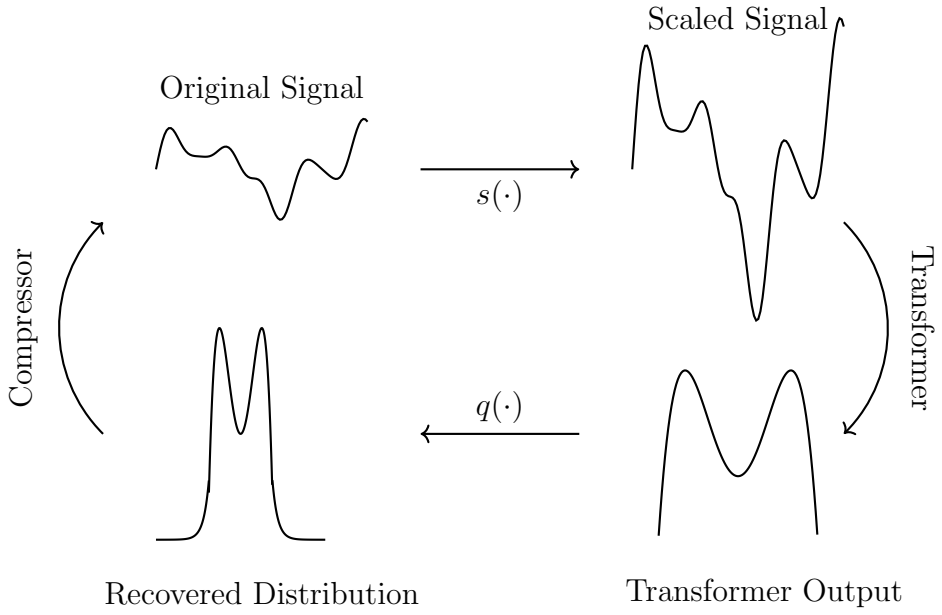


Figure 3.10: Recovering the distribution from the scaled transformer input. Scaling done with some function s and squishing done with q.

The approach relies on the hypothesis that training a model on a single scale to learn consistent signal relationships is easier than having to generalise across multiple scales, and that the relationships learned at the transformed scale hold when squished to the real scale. The original signal is scaled according to s and is used as input to the transformer, which outputs a distribution which was learned at this augmented scale. The transformer output is then squished by q to fit within the bounds of the current context window. The PMF values outside this range are set to an exponentially decaying tail. Should the next value fall out of this context window the new context window will update accordingly. Algorithm 3 describes the process for scaling and squishing at compression time.

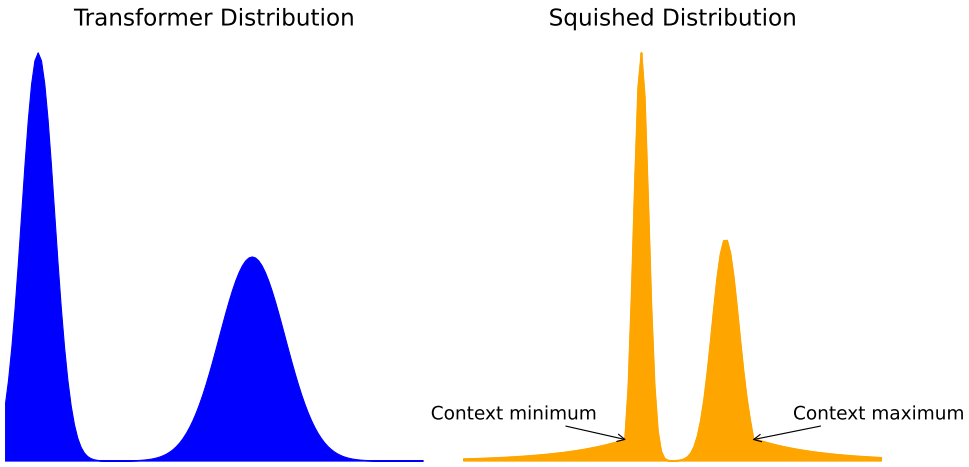


Figure 3.11: Example of a distribution being squished according to the minimum and maximum in the context window of the data being compressed. Note the exponentially decaying tail at the bounds of the range of the context window.

Algorithm 3 Single-scale Compression

$n \leftarrow |\mathcal{A}|$ ▷ n is the vocabulary / alphabet size the model was trained on, usually 128
$t \leftarrow$ current byte to compress
$\mathbf{c} \leftarrow$ previous `cw` bytes ▷ `cw` is the context window the model was trained on, usually 256
$m \leftarrow \max(\mathbf{c})$
$n \leftarrow \min(\mathbf{c})$
for $i \leftarrow 1$ to size($\mathbf{c}$) **do**
    $c_i \leftarrow [(c_i - n) \times 255] \ // \ (m - n)$
end for
$\mathbf{p}_u \leftarrow$ Transformer($\mathbf{c}$) ▷ $\mathbf{p}_u$ is the unadjusted PMF
$\mathbf{p}_a \leftarrow$ vector$_n$ ▷ $\mathbf{p}_u$ is an empty vector that will be populated by the adjusted PMF values
for $i \leftarrow 1$ to $n - 1$ **do**
    $y \leftarrow (i/n) \times (m - n) + n$
    $\mathbf{G} \leftarrow \mathcal{N}(y, \sigma^2)$ ▷ σ was chosen empirically
    $\mathbf{p}_a \leftarrow \mathbf{p}_a + \mathbf{p}_u[i] \times \mathbf{G}$
end for
$\mathbf{p}_a[: n] \leftarrow$ exponentially decaying tail
$\mathbf{p}_a[m :] \leftarrow$ exponentially decaying tail
$\mathbf{p}_a \leftarrow \mathbf{p}_a \ / \ \sum \mathbf{p}_a$ ▷ Normalising PMF
ArithmeticCoder($t, \mathbf{p}_a$)

3.6.5 Learning rate scheduling

A critical parameter is the learning rate (η), which governs the rate of change of the model parameters. [20] uses the `Adam` scheduler [49]. Research emphasises the learning rate's role in convergence [103, 74], with transformers benefiting from a 'warm-up' phase where η initially increases, then gradually decreases over each epoch as backpropagation converges on minima [56, 99].

Two learning schedulers were used, the `Adam` scheduler at a constant rate of 1e−5 and a `CosineOneCycle` schedule [83, 69]. The `CosineOneCycle` involves two phases:

i A linear warm-up where η is increased for a percentage of the epoch from an initial value until a maximum; and

ii a cosine annealing period where the learning rate is decreased to minimum.

The schedule is described in Algorithm 4

Algorithm 4 Linear ramp-up with cosine annealing (RU + CA)

for it ← 0 to ramp_up_its **do**
  $H[it] \leftarrow \eta_0 + (\eta_{max} - \eta_0) \times \frac{it}{ramp\_up\_it}$ ▷ $H[it]$ is the learning schedule for iteration it
end for
for it ← ramp_up_its to iterations **do**
  $d \leftarrow 0.5 \times (1 + \cos(\pi \times \frac{iterations - ramp\_up\_its}{iterations - ramp\_up\_its}))$
  $H[it] \leftarrow \eta min + (\eta_{max} - \eta_{min}) \times d$
end for
return H

This algorithm was applies over multiple epochs with variations in η_0, η_{max}, and η_{min}. As the learning rate was halved the batch size was doubled.

3.6.6 Positional embedding choices

Two different embedding approaches were evaluated to test the efficacy of absolute versus relative embeddings in time-series compression. Absolute positioning was implemented following the original attention mechanism outlined by Vaswani et al. [92]. It was hypothesised that relative embeddings may yield more effective compression, as mutual information in RF data is derived from the relative positioning of symbols, while absolute positions hold little meaning. Rotary positional embeddings (RoPE) [86] were used for relative positioning, with the angular schedule set to:

$$\mathbf{\Theta} = \{\theta_i = 256^{-2(i-1)/d},\ i \in [1, 2, \cdots, d/2]\}. \tag{3.1}$$

Where 256 matches the chosen context window, and d is the embedding dimension, usually 64. Setting the base to 256 ensures maximal variation across the chosen context window.

3.7 Llama models

3.7.1 LLaMA

The LLaMA model and its tokeniser were provided by the HuggingFace `Transformers` library [97]. LLaMA was not trained on a fixex-sized context windows and utilises RoPE embeddings which allow it to have varying context windows. As we are leveraging its inductive qualities rather than knowledge from training, we set its context window to 1024. This was chosen as a trade-off between predictive power and hardware constraints. The measurement of its compression is described in section 3.8

3.7.2 Headed LLaMA

For two parties to utilise LLMs for data compression and decompression, both must use the same model weights. To make LLaMA more accessible as a compressor, we avoided directly fine-tuning the main LLaMA model. Instead, we trained an appended a feed-forward 'head' layer, which maps the 4096-dimension LLaMA embedding space to a vocabulary of 128. This approach allows users to download only the head's parameters 524 416 in total $(4096 \times 128 + 128)$ reducing the required download size compared to the full 7B model. Users would still need the 7B weights, but given their growing ubiquity, we anticipate many systems will already have these models downloaded.

A high level illustration is given in Figure 3.12 and its exact training implementation is give in section 3.9.

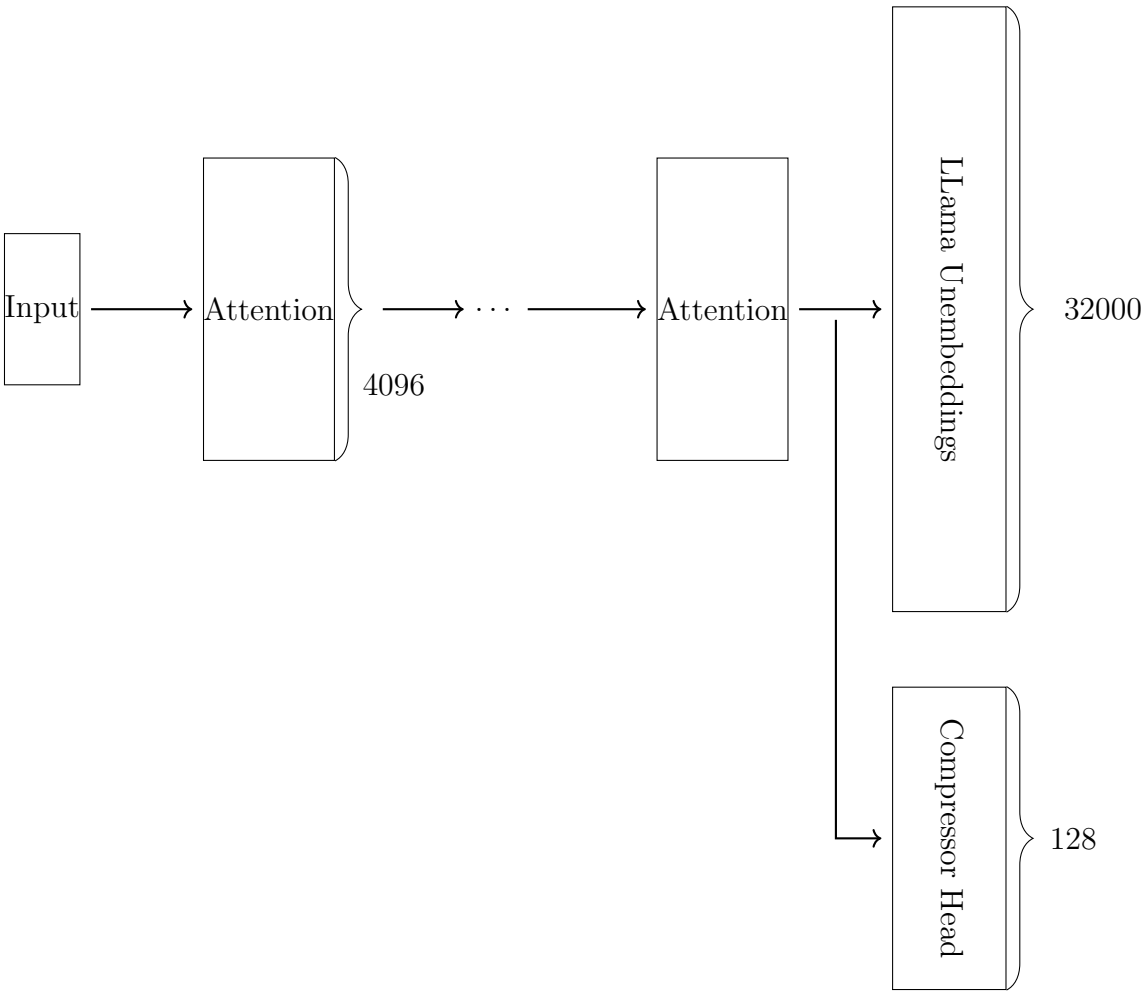


Figure 3.12: Compressor head attached and trained to augment the output of the original Llama 2 7B weights.

3.8 Measurement

Delétang et al. [20] recognised LLMs have significantly small context windows compared to `gzip` which has a 32 kB context window. They therefore measure the compression rate of a restricted `gzip` with a context window matching their transformer models, as well as an unrestricted version. This work aims to create models which are competitive against `gzip`. As such, `gzip` was not restricted. A discussion on the compression rate that considers the model size is given in section 5.3.

They also recognised the burgeoning time complexity of compressing large amounts of data with a transformer. As such, the base transformers were tested by calculating the attention scores and unembeddings for each token in the context window rather than just the final token, compressing the entire context in parallel. This prohibits accurate decompression, but we found the reported rate to be acceptably close (and marginally worse) than byte-by-byte compression but note a speed-up factor of 256.

Base transformers The base transformer models were measured by compressing $C = 256$ sized chunks randomly selected from the body of testing data. The rate was computed by comparing the number of compressed bits to the original size of the data. The first 7 bits are compressed and the final bit is appended to the output as in Figure 3.1.

$$r = \frac{\text{size(compressed tokens)} + 256}{256 \times 8}. \tag{3.2}$$

Testing was conducted on 10 random files from the test set, with 1000 random contexts selected for each combination of SNR and scaling factor, resulting in compression over 6 400 000 data points.[3] The compression rate was measured per context window, with distributions and averages provided in chapter 4

LLaMA-based models LLaMA was evaluated by selecting random windows of size $C_l = 1024$ from the test data. These selections were tokenised, producing n tokens, with the final token designated as the target for compression. A PMF was generated for the target token using the first $n - 1$ tokens. Token n was then compressed, and performance was measured against the size of its detokenised ASCII. The size of the original data was calculated as the size of the detokenised $n - 1$ tokens as ASCII. The tokenisation process of ASCII s is denoted as $\rho(s) = t$ and the reverse as $\rho^{-1}(t) = s$. We are careful to add the non-compressed bit back to the numerator $(\rho^{-1}(s)/8)$ resulting in the compression rate:

$$r_{LLaMA} = \frac{\text{size(compressed } \rho(s)) + \rho^{-1}(s)/8}{\rho^{-1}(s)} \tag{3.3}$$

Headed LLaMA acts in ASCII-space, where context windows of size $C_l = 1024$ were randomly chosen from the test data and the 1024^{th} character before tokenisation is used as target. The first 1023 tokens were tokenised and the 128-wide PMF generated by the head was used to

[3] 25 combinations of SNR and scaling, each with a 256 context window, and 1000 random selections.

compress the target character. The compression rate of a single iteration was measured as:

LLaMA and headed LLaMA were tested against 1000 random chunks of 1024 for each of the 10 test files. The compression rate is measured per file and the average compression rate is the average of the per-file compression.

$$r_{Headed} = \frac{\text{size(compressed tokens)} + 1}{8}. \tag{3.4}$$

Single-scale transformers Single scale transformers required continuous context and were compressed byte-by-byte on the first 10 000 bytes of a chosen test file over the scaled mentioned above. They were not evaluated against varying noise. Their compression rate was calculated as:

$$r_{sst} = \frac{\text{size(compressed tokens)} + 10000}{10000 \times 8}. \tag{3.5}$$

Scale and noise Notwithstanding the single-scale setup, models were tested against various signal-to-noise (SNR) levels which were calculated by taking the signal power of the noiseless training data to calculate the σ^2 required for SNRs 10 dB, 20 dB, 30 dB, and 40 dB. The base case of no added noise was also tested. All data was then evaluated by scaling at factors 0.7, 0.8, 0.9, 1, and 1.1. The data was scaled while it was in its `int16` form to maintain an average centring around 0. The σ^2 values required for the specific SNR rates were adjusted according to the scale.

3.9 Model setup

As a reference, all transformer models can be found in Appendix A.

A base transformer model was initially trained using sinusoidal positional encoding, with the `Adam` scheduler set at a constant learning rate of 1e−5. This setup used 8 heads, 16 layers, a widening factor of 4, no noise in the data, and block-strided data with a stride of 256 to match the context window. The batch size was 32. To evaluate the effect of training with noise, the setup was repeated, introducing a random frequency blur and additive white Gaussian noise (AWGN) with setup described in Algorithm 1. The hypothesis was that noise would aid generalisation on test data.

To compare learning schedulers, the same setup was used, replacing `Adam` with a linear warm-up over the first 10% of training, ramping η from 1e−5 to 1e−3 and then annealing to 1e−6. We hypothesize that this will aid convergence to a better minimum. Based on the reported efficacy of RoPE, the experiment is repeated with 4 heads and 8 layers, replacing sinusoidal positional embeddings with rotary embeddings, as defined in Equation 3.1.

The same model as above was used (with 4 layers to speed up training) but trained on unit-stride data. Additionally, we trained over 3 epochs with a linear warm-up and cosine

annealing schedule described in Table 3.5. Observing promising results of such a training regime, a large version of the above model was trained with layer count increased to 8 and embedding dimension increased to 128.

Epoch	η_0	$\eta_{\max}$	$\eta_{\min}$	**Iterations**	**Batch Size**
Epoch 1	1e−4	1e−3	1e−5	2e6	16
Epoch 2	1e−5	1e−4	1e−6	1e6	32
Epoch 3	1e−6	1e−5	1e−7	500e3	64

Table 3.5: Training Parameters over 3 Epochs

The set-up of the large version ('final model', 'big model') was considered as the best model to train the scale-invariant version of our model. Accordingly, the same conditions of this model were used to train the single-scale model but using Algorithm 2.

The LLaMA weights were not trained. A linear-layer head was appended to the final transformer block of LLaMA and trained it on 30 000 iterations with cross-entropy loss freezing the original LLaMA weights in place. AWGN or frequency blur was not applied. The data was trained according to the `AdamW` scheduler with a learning rate of 5e−5. These were the default settings of the `HuggingFace` library. To accelerate training, the LLaMA weights were quantised to `int8`.

3.10 Summary

MATLAB was used to generate 1000 training and 10 testing files of IEEE 802.11ax data yielding 24 640 000 and 246 400 tokens respectively. To allow compression using LLaMA we quantise the data to `int8` and take the ASCII representation of the data by considering the first 7 bits. This method was utilised for the non-LLaMA models to ensure comparability. We explored variations in training methodology, including positional embeddings, learning schedules, data scaling, and hyperparameter tuning, and provide a detailed description of the 'headed LLaMA' model and its training process. Finally, we outlined the methodology for measuring compression rates and specified the exact combinations of training approaches used.

Chapter 4

Results

4.1 Introduction

The results are divided into four sections. First, we consider the aim of creating a noise-resistant compressor and demonstrate the robustness of the training regimes described in section 3.9. We do not consider the single-scale training algorithm or LLaMA. Secondly, we determine how LLaMA and the headed LLaMA model compress against varying levels of noise and scales. Thirdly, we analyse the ability for the single-scale training to be robust against various scales and compare its ability against the other training models. Finally, we provide a brief summary of the results leaving detailed for chapter 5.

4.2 Noise-resistant models

Excluding LLaMA and the single-scale model, the models described in section 3.9 are tested against the noise and scales described in section 3.8. Figure 4.1 plots the compression rate distribution for various models, iterating from the initial naive model over the proposed techniques in section 3.9. Compression rate is given at 30 dB to emphasise a model's ability to withstand noisy data, particularly demonstrating the subpar compression rates for the model which was not trained with noise. Table 4.1 records the average compression rate for these distributions. The large model had the best average compression rate at 63.92 %. We analyse this model further by showcasing its compression rate across multiple SNR (Figure 4.2) and scales at 30 dB (Figure 4.3). We provide the compression rate for this model over all combinations of noise and scale in Table 4.2.

In Figure 4.2 we observe a decline in compression rate as noise levels increase. Initially, the transformer demonstrates a clear advantage over `gzip` but as the signal degrades into pure noise its predictive power converges toward that of the unconditional distribution used by the naive arithmetic coder. Figure 4.3 demonstrates that despite the transformer being trained at unity scale, the compressors performance does not degrade when scaled. This is further discussed in subsection 5.2.5.

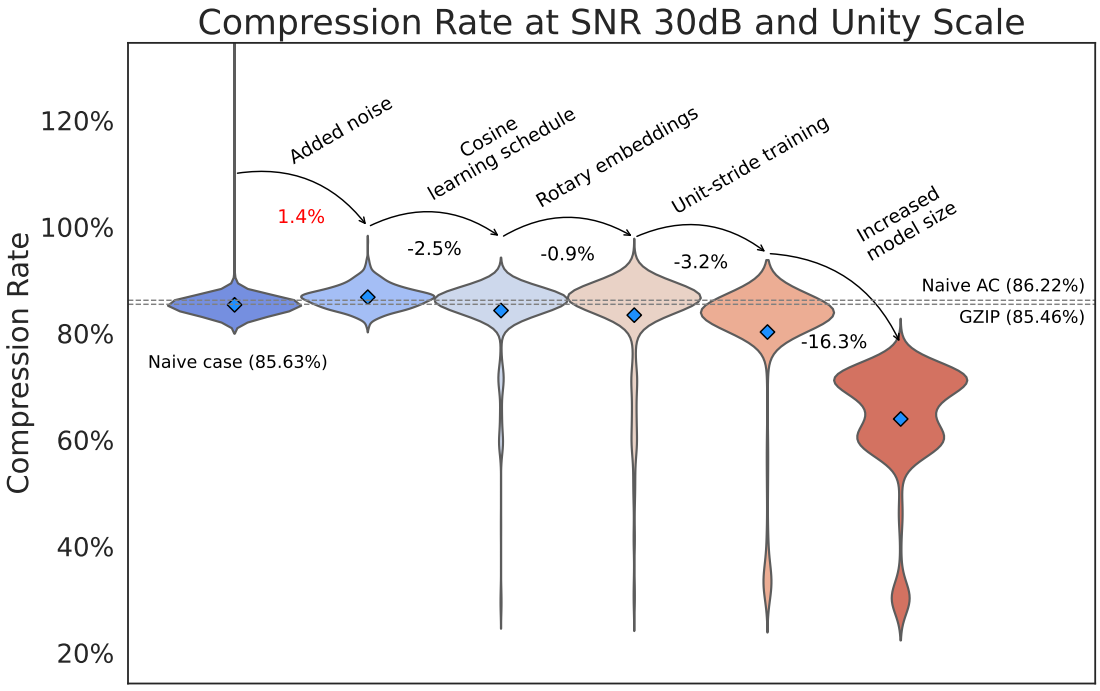


Figure 4.1: Various models ability to compress at 30 dB of added noise. The blue diamond indicates the mean compression of each model. The 'GZIP' and 'Naive AC' dotted lines are the measured compression rate of `gzip` and arithmetic coding using an unconditional distribution of the training data. The arrows are indicative of the changes made to each model and demonstrate how such changes impact the compression rate with the delta change indicated by the percentage below the arrow. Red indicates a worse compression rate. Note that the tail of the base case extends upwards of 140 % but was cut off for sake of compact illustration.

Model	**Naive**	**+ Noise**	**+ RU&CA**	**+ RoPE**	**+ Unit-stride**	**+ Model size**
r	85.63 %	86.79 %	84.29 %	83.43 %	80.25 %	63.92 %

Table 4.1: Compression rate across compressor models at 30 dB.

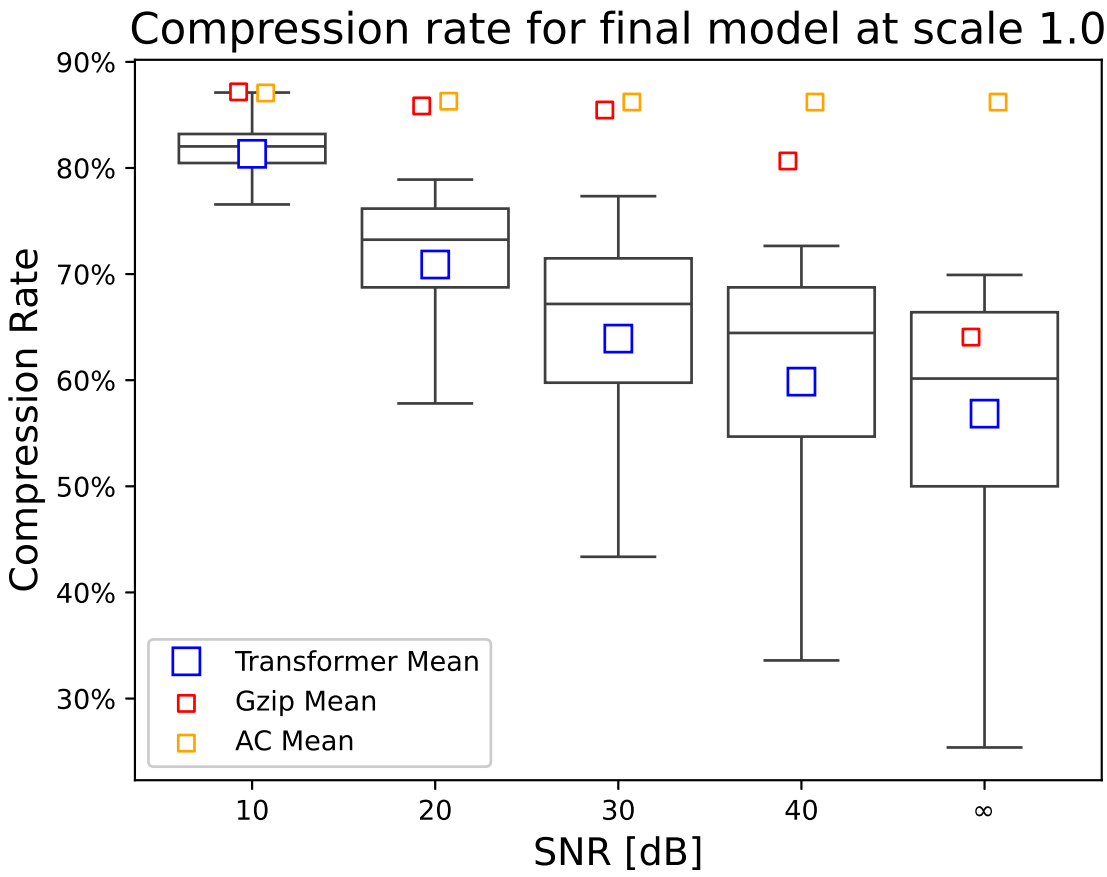


Figure 4.2: Boxplot of compression rate over varying SNR for the final compressor. We include the compression rates for `gzip` and naive arithmetic coding for comparison.

SNR / Scale	**1.1**	**1.0**	**0.9**	**0.8**	**0.7**
10	83.67 %	81.33 %	78.98 %	76.86 %	74.92 %
20	72.99 %	70.90 %	69.13 %	67.46 %	65.75 %
30	65.78 %	63.92 %	62.42 %	61.36 %	59.69 %
40	61.44 %	59.87 %	58.56 %	57.78 %	56.77 %
∞	57.59 %	56.85 %	56.13 %	53.99 %	53.99 %

Table 4.2: Compression rates for the final model over varying scale and noise levels.

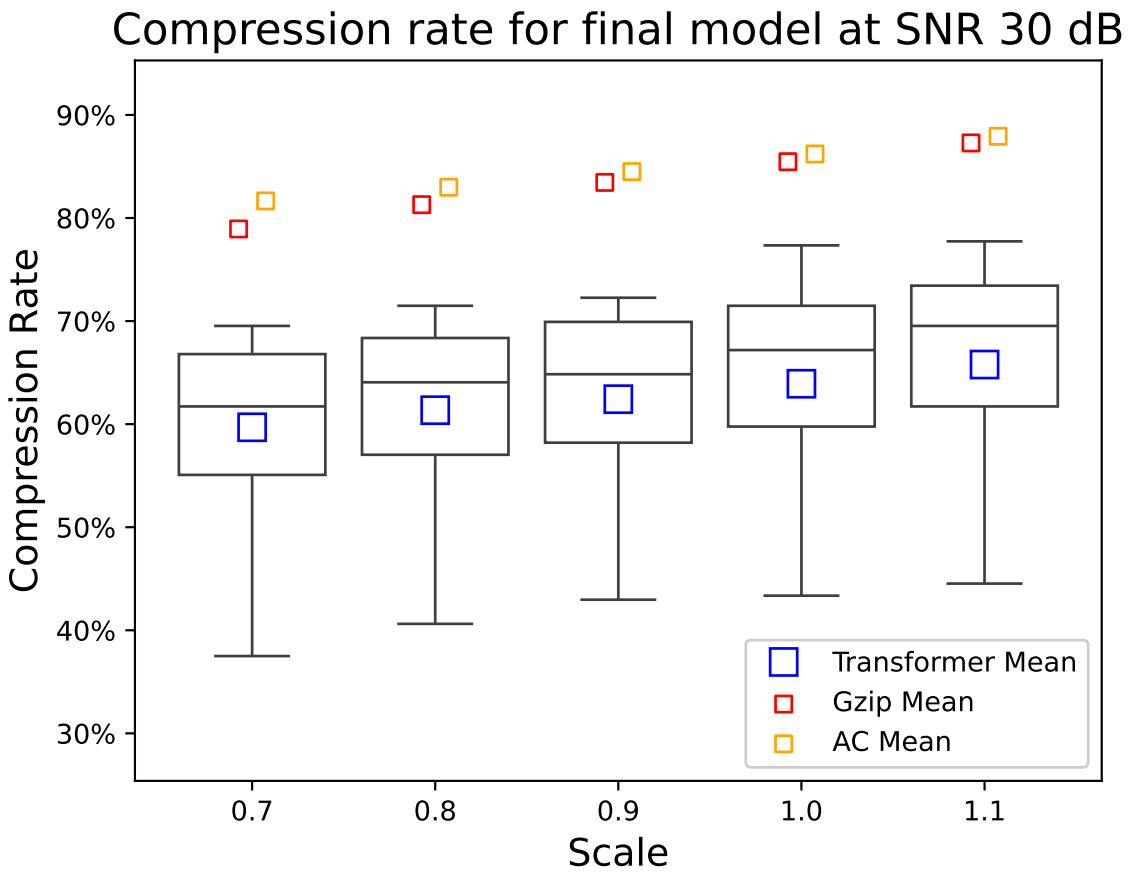


Figure 4.3: Compression results for the final compressor over varying scales at 30dB of noise.

4.3 LLaMA models

LLaMA and headed LLaMA were evaluated over varying scales and SNR levels. Figure 4.4 illustrates both models' performance at unity scale against varying noise levels. Headed LLaMA performs consistently worse in all cases. LLaMA out-performs `gzip` on average in all SNRs except with no added noise. The exact compression rate of LLaMA is found in Figure 4.5 over combinations of noise and scaling factors. The number below the compression rate is the compression rate relative to `gzip`, with red percentages indicating where `gzip` out-performed LLaMA. The values reported are slightly different to the means seen in Figure 4.4 as they were generated over separate testing runs. As the data's scale is reduced, LLaMA's compression rate proves less competitive than `gzip`. Indeed there seems to be a 'goldilocks' condition of SNR where LLaMA outperforms `gzip` consistently, between 30 dB and 40 dB SNR. This is further discussed in subsection 5.2.4.

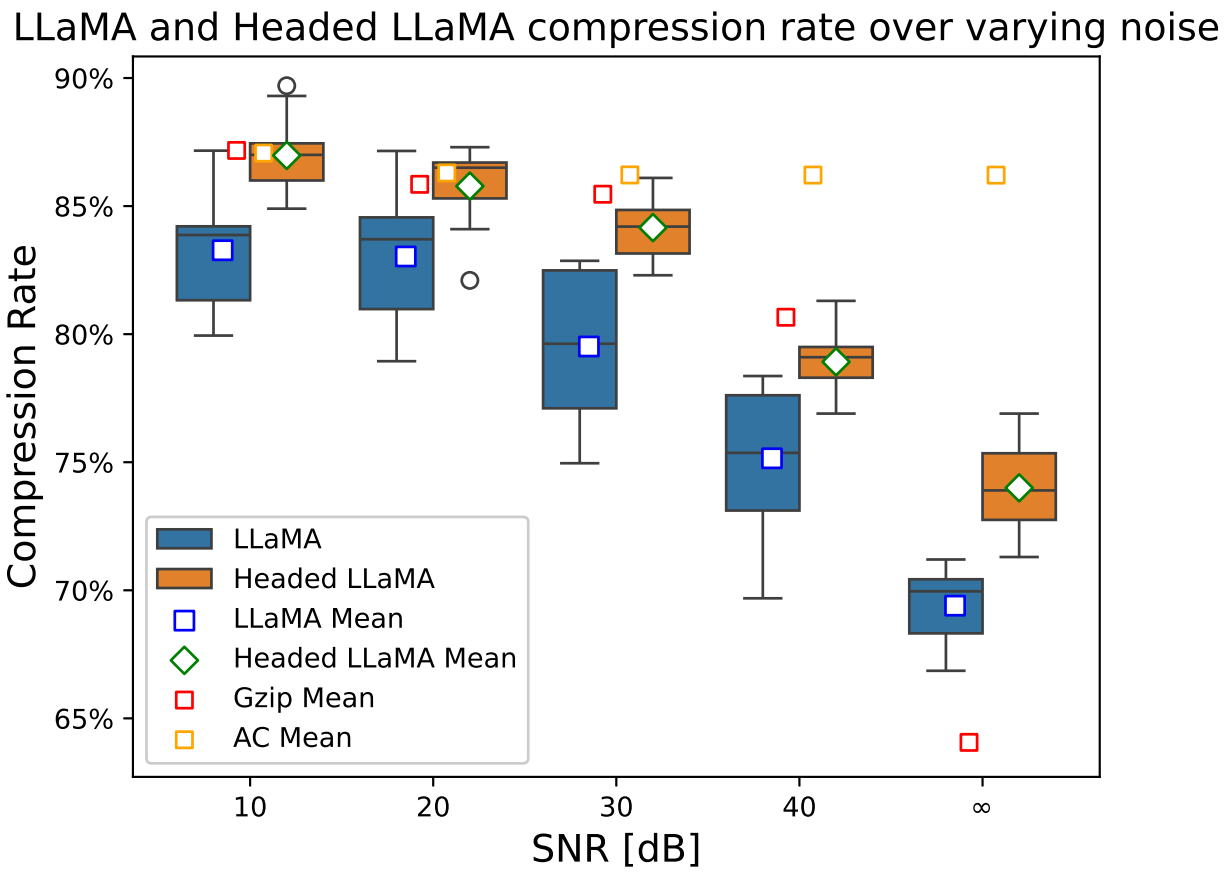


Figure 4.4: LLaMA and headed LLaMA performance over varying SNR.

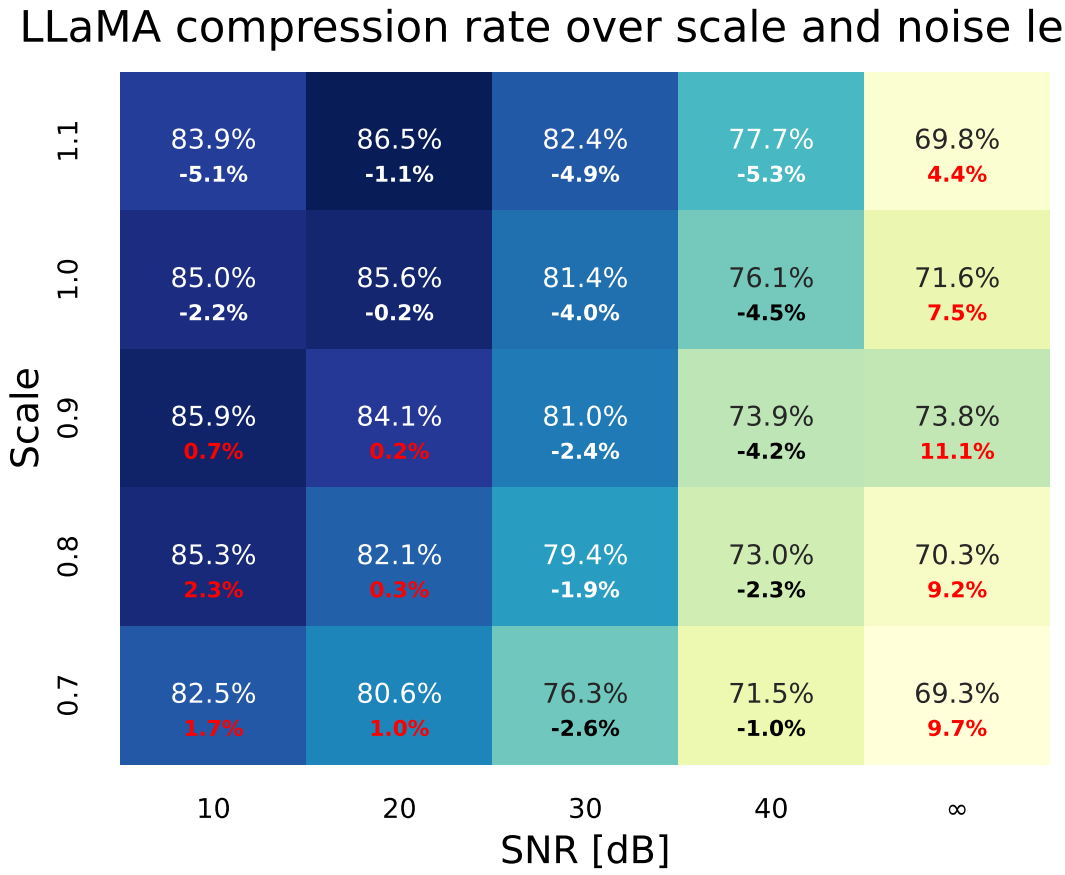


Figure 4.5: Compression rate of LLaMA over varying scales and noise levels. The number below the compression rate is the difference in rate with `gzip`, where red indicates worse performance.

4.4 Scale-resistant models

The single-scale transformer was tested over the first 10 000 data of a test-file over various scales with no added noise. Since the rate is not calculated in windowed batches a distribution is not provided. The σ in Table B.12 is that of the Gaussian kernel in the PMF squishing process described in Algorithm 2. Unlike the previous results we considered scaling factors down to 0.6 to illustrate performance against more pronounced scaling factors.

Scale / σ	**1.1**	**1.0**	**0.9**	**0.8**	**0.7**	**0.6**
3.0	73.8 %	79.94 %	76.44 %	79.82 %	79.72 %	78.82 %
2.0	71.64 %	78.58 %	75.4 %	79.08 %	79.64 %	78.14 %
1.5	70.16 %	77.64 %	74.64 %	78.56 %	79.54 %	78.08 %
1.0	68.22 %	76.4 %	73.58 %	77.86 %	79.26 %	77.98 %

Table 4.3: Single-scale transformer results over various data scales and σ with no added noise.

Figure 4.6 compares the single-scale compressor's (with $\sigma = 2$) resistance to changes in scale against LLaMA, the best compressor from above, and `gzip`. We normalise each model's compression rate to its performance at unity scale. All models followed a similar trend; the compression rate improves with decreasing scale. The single-scale compressor's normalised performance is marginally better in all scales except 0.8, but Table B.12 reveals it is less performant than the best transformer from the previous section. This result is further discussed in subsection 5.2.5.

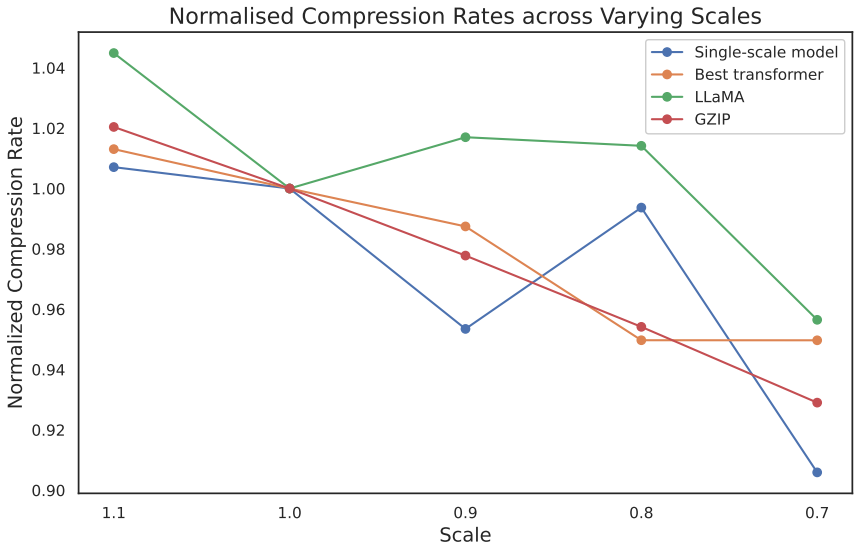


Figure 4.6: Graph of compression rate across various models normalised to unity-scale performance. We vary across scales to measure the resistance to changing scales of each model. `Gzip` is included as reference.

4.5 Summary of results

Transformer models were tested over various scales and SNR to examine their performance comparatively with `gzip` and naive arithmetic coding. The models were first tested for their noise resilience. It was found that with a combination of unit-stride training, adding noise, rotational embeddings, and a cosine annealing learning schedule, we were able to consistently out-perform both `gzip` and naive arithmetic coding, achieving an improved compression rate of 7.21 % in the noiseless case and up to 20.79 % in the 40 dB case.

The single-scale model proved less efficient than the model described above (76.4 % versus 56.9 % in the unity scale and noiseless case.) While it did maintain its compression rate across scaling factors from 1.1 to 0.7, the non-scaled transformers, including LLaMA, maintained their compression rate across this range, typically improving with smaller scales. This model was unable to out-perform `gzip` but proved better than naive arithmetic coding.

The headed LLaMA model worsened the compression rate of LLaMA. It only marginally outperformed `gzip` at unity scale and at 40 dB (78.92 % versus 80.66 %). LLaMA outperformed `gzip` at all SNR with unity scale except when no noise was added. LLaMA's edge over `gzip` diminished as the scale of the data was reduced.

Chapter 5

Discussion

5.1 Introduction

We herein analyse the above the results found in chapter 4, offering commentary and hypotheses on specific outcomes. A summary of key results is available above in section 4.5. The chapter begins with section 5.2 where we interpret such findings and provide critical analysis on DeepMind's original work, followed by section 5.3, which discusses implications and evaluates our models as potential alternatives to traditional methods like `gzip`. We conclude with a discussion on the limitations of this work.

5.2 Analysis of results

In chapter 4 we analysed multiple different models by iterating on a 'naive' base-case with additional techniques to try improve the compression rate. The choices and their impact were graphed in Figure 4.1. We will first analyse a selection of these choices and their measured impact on performance. We follow with discussion on LLaMA and headed LLaMA's performance and hypothesise why the headed LLaMA only saw to diminish performance. Finally, we discuss the performance of the single-scale performer and give reasoning for its inefficacy.

5.2.1 Adding noise

The first modification to the naive model was to add noise during training, detailed in Algorithm 1. This negatively impacts the mean compression rate illustrated in orange text in Figure 5.1. The figure demonstrates that with added noise, the model is less prone to extreme over-compressions shown by the strong tails of the naive distribution. This tail suggests the naive model has a tendency to be confidently wrong. This was also seen in the summary graph Figure 4.1.

We hypothesise that training with noise makes pattern learning inherently more challenging

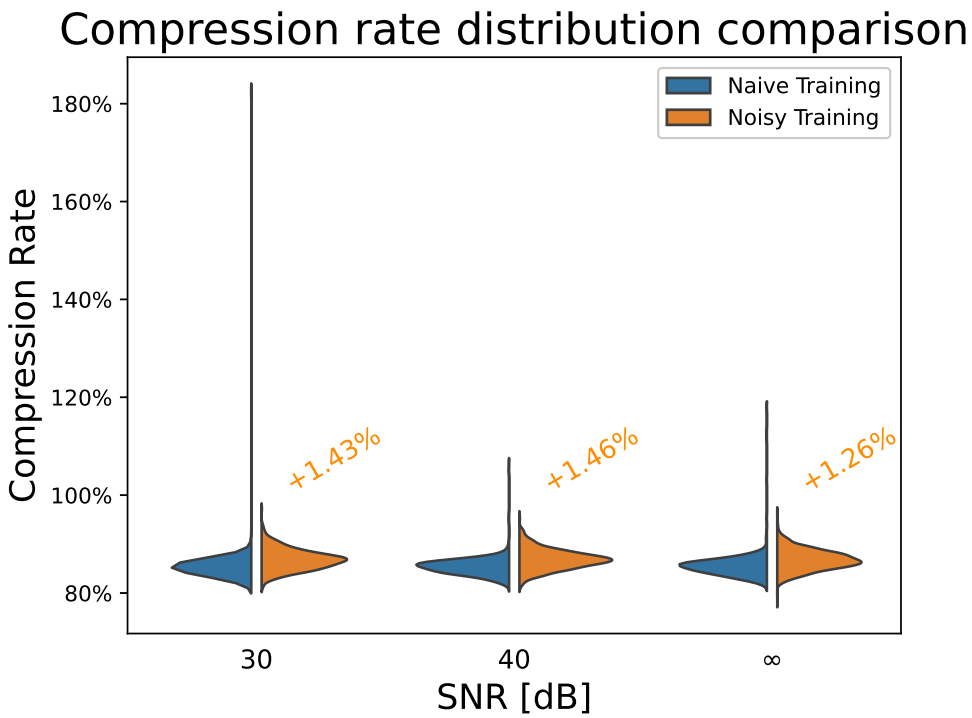


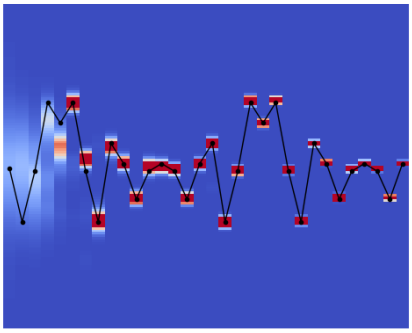
Figure 5.1: Distribution of measured compression rate for naive and noisy training at 30, 40, and ∞ SNR.

for the transformer. Given the small model size and limited training we suggest that this accounts for the reduction in the average compression rate. We next add a linear ramp-up and cosine annealing (RU + CA) learning schedule which helps the model converge with the addition of noise. This was seen as the second and third distribution in Figure 4.1, improving the compression rate by 3.36 % at 30 dB. A similar improvement was seen over all SNR levels except 10 dB.

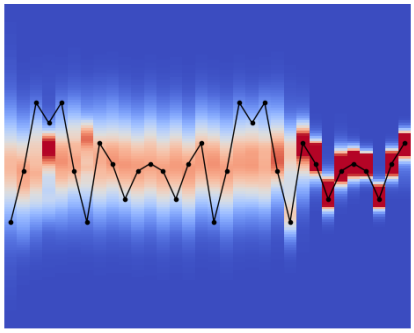
5.2.2 Training stride

The setup in DeepMind's code uses block-strided training and testing data as visualised in Figure 3.8a. We find this is insufficient for RF compression and does not generalise well to data that is not strided with the same initial offset, unsurprisingly, given that the absolute positional embeddings are unaligned. Absolute embeddings does not properly reflect the nature of RF; the absolute position of an RF observation has no meaning. In order to compress and decompress we have to take a sliding window over the data, moving byte-by-byte over each symbol. Therefore it does not make sense to train on mutually exclusive block-strided chunks.

Figure 5.2 demonstrates a compressor which was trained on block-strided data. In this example, The first 32 bytes of a training file are compressed. The start of each file is a pilot signal which is a simple repeating pattern. In Figure 5.2a we do not offset and the compressor is able to recognise the sequence. Notice in Figure 5.2b the high-entropy blur at the start of the sequence. The sequence, when offset by one byte, proves difficult for the model to compress. It takes many more observations before the transformer recognises it is seeing the pilot signal. The compression rate is reduced in the offset sequence by 37.5 %.



(a) No offset, compression rate is 40.63 %

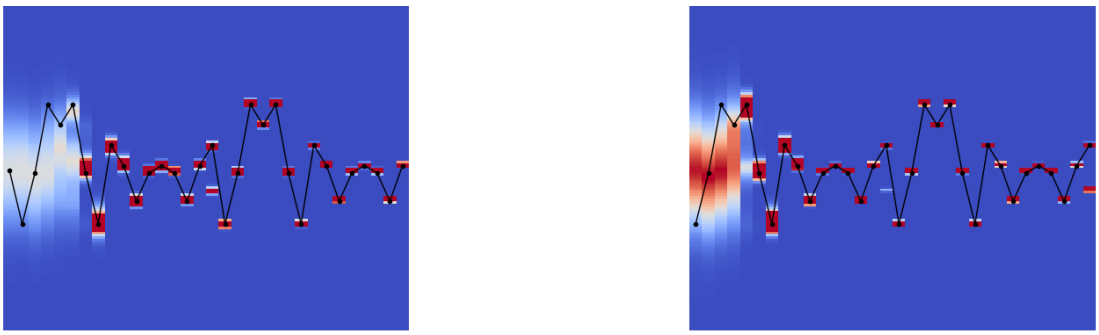


(b) Offset by 1, compression rate is 78.13 %

Figure 5.2: The above model was trained on chunks of 256 bytes. The left figure shows the compression heatmap when we are aligned to the training chunks (in-phase), and the second heatmap shows the heatmap when we are offset by just one byte.

Compare this with Figure 5.3 where we repeat the same experiment with a model that was trained with unit-strides. Such a model did not suffer any such degradation in performance

and was able to determine it was compressing the pilot sequence quickly.



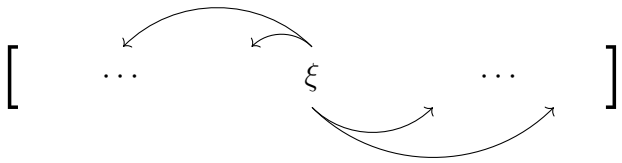
(a) No offset, compression rate is 37.5 %

(b) Offset by 1, compression rate is 31.25 %

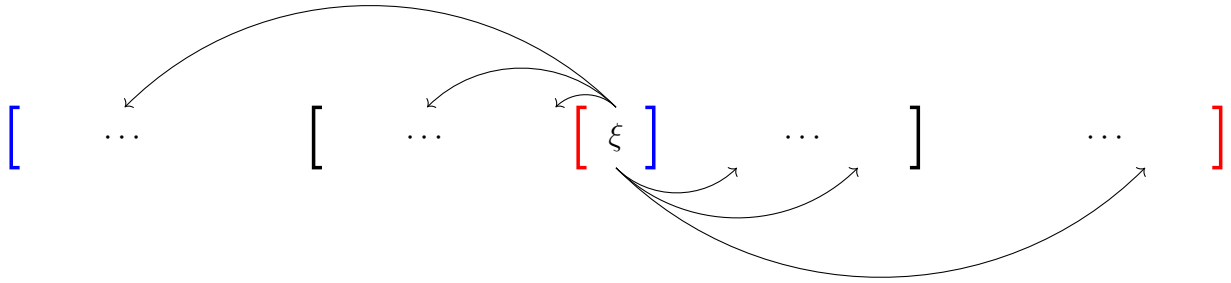
Figure 5.3: Here, the model is trained on all offsets and is able to recognise the pilot sequence regardless of offset. Compression rate is comparable for both scenarios.

Figure 4.1 shows an improvement of 3.2 % when training on unit-strided data. Contrast this to the improvement in the compression rate when training with RoPE instead of SPE, which was just 0.9 %. Despite the rotary embeddings learning the relative positional meanings of observations, we suggest there was not enough data to fully train the transformer to learn these relative meanings. By only training on block-strided data, we are learning on an arbitrarily restricted amount of data.

Consider a rare token ξ in a context window with C symbols. With a block-strided approach we may train over the relationship between ξ and $C-1$ other symbols with the same sequences each time potentially overfitting the embedding of ξ. Contrast this with unit-strided training which will consider $2 \times (C-1)$ tokens, for a total of C different sequences. These training paradigms are visualised in Figure 5.4. Consider the way these can be combined in batches to give a more varying estimate of the true gradient ∇H. We see considerably stronger compression when training on 256 times more data.



(a) ξ learning its relationship to $C-1$ tokens in block-strided training.



(b) ξ learning its relationship to $2 \times (C-1)$ tokens in unit-strided training.

Figure 5.4: Comparing chunked learning versus stream learning. The colour of the bracket indicates an independent context.

5.2.3 More is more

The penultimate model in Figure 4.1 combined techniques which were shown to improve compression rates such as training with noise, training streamed data, ramp-up and annealing, and using rotational positional embeddings. We postulated with such an efficient combination of techniques we might have been able to reduce the size and time complexity of the model by trading size for more efficient training techniques.

Compared with the ante-penultimate model, this smaller model used half as many layers (8 down to 4) yet improved on the compression rate by 3.2 %. (83.43 % to 80.25 %). Finding the compression rate still wanting, we created a larger version of the model which doubled the embedding dimension (64 to 128) and doubled the layers (4 to 8). This change achieved the maximal improvement of 16.3 % (80.25 % to 63.92 %). We approximate the the smaller model to have 205 000 parameters and the larger model to have 1 600 000 parameters.

There is an intrinsic tension in machine learning between model complexity and generalisation ability. Classical statistics dictates that too complex a model will lead to over-fitting of the training data. Chapter 5 of Shalev-Shwartz and Ben-David [79] discusses such a notion formally as the bias-complexity trade-off which decomposes the error of a model into its approximation error (the error due to model limitations such as a restricted parameter space) and the estimation error (error arising in inference, often due to an overly complex model trained on a limited dataset). In mitigate overfitting, training schedules typically incorporate regularisation [54] - techniques which help prevent the model memorising the training data at cost of generalisation. A common approach is early stopping, where training ceases once the validation set error begins to rise, even as training loss continues to decrease. In

this work we do not use any regularisation techniques.

Since neural networks can act as universal approximators [37] models may be trained to some arbitrarily small approximation error ε by increasing the parameter count at the cost of an increased estimation error. The results, however, reveal a different trend. The larger model was able to generalise better and the smaller model failed to converge to a competitive compression rate. Recall Figure 3.5 from subsection 3.5.1 where an appropriate embedding dimension was selected for this study. It was demonstrated that an embedding dimension of 128 overfit the training data, yet our large model with the same embedding dimension performed better than the smaller model with dimension 64. We illustrate the training loss over the 3 training epochs in Figure 5.5 and observe the loss stagnation in the smaller model

Figure 5.5: Loss over three epochs for the small and big models. Notice the ramp-up period at the start of epoch 2 helps the big model converge to a better loss.

Zhang et al. [100] argue that that conventional statistical concepts such as the bias-complexity trade-off are insufficient for explaining generalisation in large machine-learning tasks. Their findings indicate that regularisation techniques are neither necessary nor sufficient for training models to generalisers effectively. Rather, the choice in parameters and architecture play a more important role generalisation ability. Nakkiran et al. [65] demonstrate that beyond a certain model complexity, the bias-complexity trade-off fails to explain the observed improvements in test performance, leading to the contemporary approach to statistical modelling, namely 'more is more'.

5.2.4 LLaMA-based models

LLaMA

LLaMA's performance (Figure 4.4, Figure 4.5) at unity scale does not out-perform `gzip` (71.6 % versus 64.1 %) however at increasing levels of noise we achieve an improved compression rate. 40 dB saw the best improved rate at unity scale (4.5 %) however the edge over `gzip` diminishes as the noise increases. Noise negatively affects the performance of `gzip` by reducing the amount of repeated patterns to extract; and in LLaMA by increasing the difficulty of the model to deduce the underlying pattern.

The transformer which were trained on WiFi data outperformed both LLaMA and headed LLaMA (in unity scale and 30 dB SNR 63.92 % versus 81.4 %, and 84.16 % respectively) but recall that LLaMA has likely never been trained on a WiFi signal. This result is in-line with DeepMind's results, that foundation models can act as general-purpose compressors and echoes the literature; large foundation models are strong inductive reasoners [51, 29, 60, 90].

Figure 5.6 illustrates the first 32 bytes of a test file, with its equivalent in ASCII given below.

Figure 5.6: Pilot signal

```
I.AYQYA.JC7ACA7CJ.AYQYA.JC7ACA7C
```

Using basic inductive reasoning, it is possible to observe the pattern and predict the next few bytes as `J.AYQYA`. However, more complex data segments, such as those containing the RF payload present a greater challenge for LLaMA. The LlaMA models were given a context of 1024 bytes which allowed them greater opportunity to learn the patterns intrinsic in the BPSK signal.

Contrast the tokenisation of LLaMA with the models we trained explicitly on WiFi LLaMA. In Figure 3.6 we illustrated numerical values which were close were embedded to a similar

part of the embedding space, 10 was semantically similar to 11 (Definition 1). LLaMA however was trained on English text and the same semantic relationships do not hold. Tokens which are close numerically in respect to their ASCII code are not embedded to be geometrically close, yet we are still able to achieve a compression rate better than `gzip` in certain scenarios.

We hypothesise that the models work to infer differently:

1. The purpose-trained transformers can recognise specific sequences characteristic of a BPSK payload. They inherently understand that numerically close values have similar meanings, exhibit smooth distributions, and maintain resilience to noise.

2. LLaMA has no knowledge of BPSK. It does not recognise specific sequences nor understand that the input represents an RF wave. Further, it is not explicitly trained to continue such sequences. In the absence of English text or explicit instructions it infers a need to extend the observed pattern through its inductive reasoning. It learns the relationships between characters based on the context which is provided to it.

The results reported in Delétang et al. [20] showcase impressive compression of the LibreSpeech dataset using LLaMA as a compressor (23.1 % versus 36.4 % for `gzip`). Instead of using the 32 000-wide PMF generated by the LLaMA unembedding layer, only the top 100 log probabilities are taken as the PMF for the arithmetic coder. This approach was effective their experiment as all the tokens to be compressed were within the top 100 probabilities. The smaller distribution necessarily has less entropy than a 32 000 wide distribution. We argue that this adjustment is unsuitable for practical compression as the programme cannot know whether the token to compress lies in the top 100 until after inference. Consequently, decompression would then require non-causal information making decompression impossible.

Headed LLaMA

Headed LLaMA consistently underperforms compared to standard LLaMA. Two hypotheses are proposed to explain this result.

Insufficient training Headed LLaMA was trained on 30 000 iterations as training speed was hindered by LLaMA's inference time in the forward-pass of the backpropagation algorithm. The head also only had 524 416 parameters, a comparatively small network.

Insufficient architecture; The head attached to the last transformer block of LLaMA was a linear layer with a linear activation function. According to the *Universal Approximation Theorem* proposed by Hornik, Stinchcombe, and White [37], multi-layer networks can serve as universal approximators if they use a non-polynomial activation function. Specifically, the theorem requires that the activation function (i) Map to a finite range; (ii) be non-linear; and (iii) be monotonic.

The activation function used by headed LLaMA was $\phi(x) = x$, which violates the first two requirements. The entire head f is therefore a linear map:

$$f : \mathbb{R}^{4096} \to \mathbb{R}^{128}. \tag{5.1}$$

Two questions are posed; does the linear activation function make it unable to learn extra information due to violation of the universal approximation theorem, and, if the head f could learn something useful, what would such a linear mapping learn?

Recall LLaMA's 32 000 tokens which comprise of a combination of ASCII characters. Instead of mapping to each of these 32 000 tokens we hypothesise it would map the embeddings to $\mathbb{R}^{128}$ such that the probability of some token $x \in [0, 127]$ would be the sum of all the probabilities in the original embedding space of tokens that began with the character x. To clarify, consider mapping $\mathbf{x} \in \mathbb{R}^{4096}$ with both the the standard LLaMA head $g : \mathbb{R}^{4096} \to \mathbb{R}^{32000}$ and the appended head f of the character 'h'.

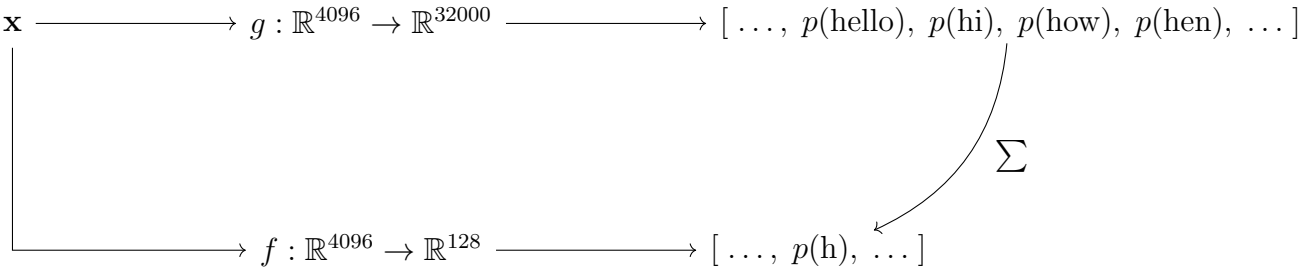


Figure 5.7: A proposed model for how f might learn. The probability in the $\mathbb{R}^{128}$ is the sum of probabilities in $\mathbb{R}^{32000}$ sharing the same first letter.

While it is difficult to predict what f might learn without training and empirical observation, Figure 5.7 raises a question for future research: can a for-purpose 'head' be trained in a way to improve compression? Our hypothesis is that no such linear mapping would outperform LLaMA's current compression rate but propose a more sophisticated head such as an extra attention head may improve on LLaMA's result.

5.2.5 Single-scale results

Unexpectedly, models trained on a single scale were able to maintain their edge over `gzip` and naive arithmetic coding when compressing data at different scales. With no added noise the compression rate for the final transformer outperformed `gzip` by 7.78 %, 7.21 %, 6.51 %, 7.14 %, and 5.53 % at scales 1.1, 1.0, 0.9, 0.8, and 0.7 respectively. Figure 4.3 illustrates this result with the final transformer model.

Data scaled by factors below unity have a smaller range, improving performance of both `gzip` and naive arithmetic coding. Variations in the RF are be quantised away by the smaller scale increasing the frequency of repeated patterns (sequences that were once different may become the same as extra detail is scaled away). Likewise, the PMF used by the arithmetic coder will have less entropy as the range of the distribution of the data decreases. This can be shown mathematically by estimating the data as Gaussian:

$$\mathcal{X}_1 \sim \mathcal{N}(\mu, \sigma^2) \tag{5.2}$$

Scaling every observation by 0.7 yields the distribution:

$$\mathcal{X}_{0.7} \sim \mathcal{N}(\mu, 0.7^2 \times \sigma^2) \tag{5.3}$$

And the entropy of a Gaussian is:

$$\begin{aligned} H(\mathcal{X}_1) &= -\int p(x) \log p(x) dx \\ &= -\mathbb{E}[\log \mathcal{N}(\mu, \sigma^2)] \\ &= \frac{1}{2} \ln(2\pi e \sigma^2) \end{aligned} \tag{5.4}$$

Equation 5.4 illustrates that entropy is a function of the standard deviation of the distribution, which is shown by Equation 5.3 varies with the scaling factor. The reason why a compressor trained on a single scale is able to maintain its performance is unclear. Three hypotheses are proposed to explain this result:

1. When the transformer is less certain in its prediction, the PMF it generates tends to centre around the midpoint of the data. Scaled down values are inherently closer to this midpoint;
2. Scaling has a minimal effect on data near the midpoint. This allows the transformer to continue to recognise sequences it was trained on close to the midpoint; and
3. The scaling factor is within noise margin the model was trained on, the model perceives the scaled data as simply a noisy signal.

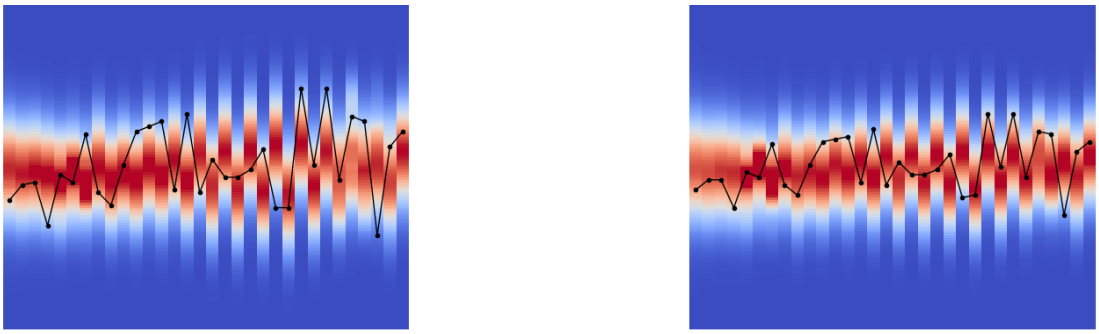
A sequence of data which illustrates these points is compressed using the final transformer model at both unity scale and a at a factor of 0.7.

Figure 5.8 demonstrates point (1); the individual observations are more closely centred around 0 and align more closely to the generated PMFs, which are similar for both sequences.

Figure 5.9 demonstrates point (2); the closer an observation is to the midpoint the less it is scaled. The transformer can still recognise these observations.

The sequence is graphed with a 95 % confidence interval of the added noise from training. Recall that the noise added to the training data is from $\mathcal{N}(0, \sigma^2)$ where $\sigma \sim \mathcal{U}(0, 5)$.

Figure 5.10 demonstrates point (3); signals which have been scaled by the smallest scaling factor 0.7 are either in or are very close to the 95% confidence interval of the training noise.



(a) Unity scale. $r = 90.63\,\%$ (b) 0.7 scale. $r = 81.25\,\%$

Figure 5.8: Compression heatmap for unity and 0.7 scaling factor.

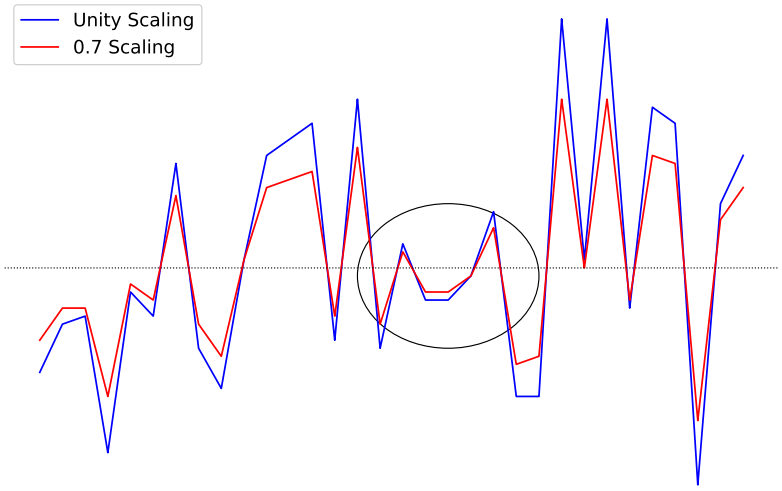


Figure 5.9: The sequence in Figure 5.8 with unity and 0.7 scaling overlaid. We see around the zero that values are very similar.

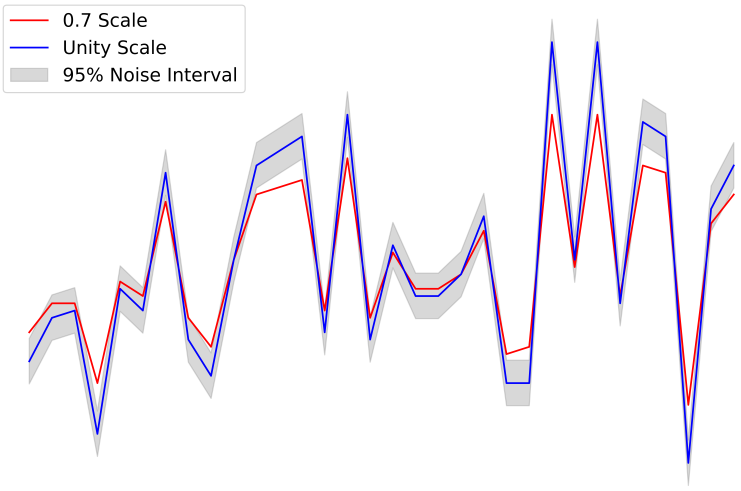


Figure 5.10: 95 % confidence interval of noise in the same sequence with unity and 0.7 scaling overlaid.

The performance of the single-scale transformer was worse on all scales with zero added noise compared to both the best transformer model and `gzip`, shown in Table B.12. The relative compression rate graphed in Figure 4.6 indicates the the single-scale model might be better at maintaining its performance with different scales but we suspect that this is not statistically significant. We propose that the scaling technique used in training between tokens and is a suboptimal regime.

Algorithm 2 scales data to a 0–255 range before removing the least significant bit, making it sensitive to the context window's minimum and maximum value. As illustrated in Figure 5.11, sequences offset by one byte exhibit considerable differences in scaled form due to variations in original sequence's extrema. This scaling introduces distortions in previously similar data, potentially hindering model convergence.

Finally, it is hypothesised that because the distribution is squished within the estimated bounds of the signal, any datum to compress falling outside these bounds requires significantly more bits for coding. This is due to the distribution in these regions being populated with exponentially decaying probabilities. Figure 5.12 demonstrates this penalty with the datum marked with a circle falling out of the bounds of the previous 256 bytes. Because of this it is compressed with 13 bits (nearly 100 % worse than no compression). The estimated bounds are subsequently adjusted.

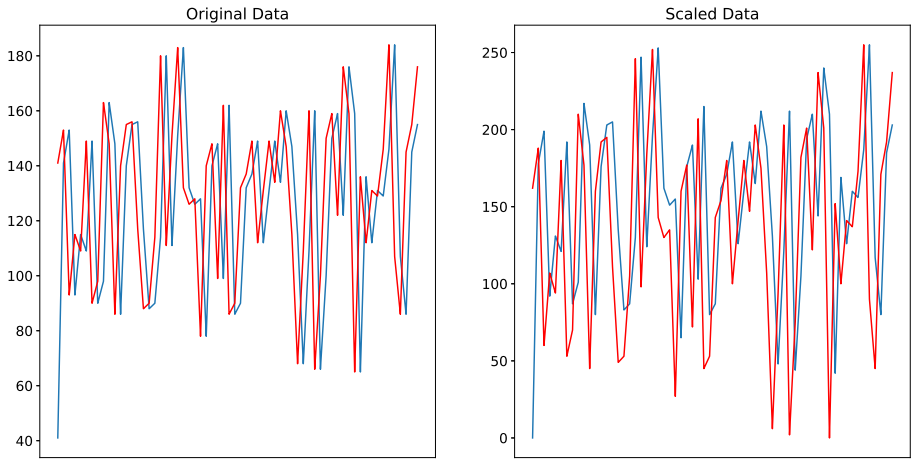


Figure 5.11: Distortion effect of Algorithm 2. Each line is a sequence one byte apart. On the left the original data is clearly shifted by one, whereas the scaled data becomes distorted as the minimum value changed dramatically in the original data (notice the minimum at the start of the original data.)

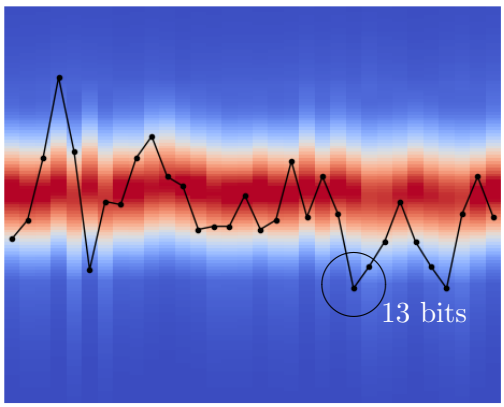


Figure 5.12: The circled measurement is compressed with 13 bits as it falls out of the estimated bounds of the signal.

5.3 Implications

This paper has explored the principle that there is no such thing as a free lunch. Data compression seems to follow such an axiom. While we have achieved compression rates that out-perform `gzip` under certain noise and SNR conditions, this success came at a significant cost in computing power and time. Space complexity was improved through a more performant compression rate at the expense of the increased compute time due to the transformer's inference.

Figure 5.13 illustrates these trade-offs. Although our best model achieved an average improvement of 7 % over `gzip`, this gain is overshadowed by a 3.5×10^6-fold increase in time complexity and a 68-fold increase in space complexity. Beyond this computational penalty, trained transformers can only compress a specific type of RF data, whereas `gzip` remains universal, compressing all data types without prior knowledge Delétang et al. [20] demonstrated LLaMA's performance on multiple data types but it remains uncertain if its inductive capabilities can match the versatility of `gzip` across all data.

It was argued in section 3.7 that the increasing ubiquity of LLMs suggests that most systems will eventually support transformer models like LLaMA, which could serve as compressors. If time constraints were negligible and high-efficiency compression was essential, large transformers might find a practical use. This however does not take into account a third constraint, power. The electrical cost of inference for an LLM far exceed a compression using `gzip`.

`Gzip` is a popular compression software due to its speed, however alternatives such as `bzip2` [8] and `xz` [88] are generally slower but achieve better compression rates [14]. These formats were not tested in this study however may be appropriate for scenarios where time complexity is less of a consideration.

The most notable limitation of both the trained transformer and the LLaMA models was that compression rates were measured against a reduced-precision version of the data, where 8 of the least significant bits were removed. A method for compressing the non-quantised `int16` data on a text-trained foundation model has yet to be demonstrated. Furthermore, while a model could theoretically be trained to handle `int16` data, this would increase the possible tokens to $2^{16} = 65536$ making it challenging to reduce the entropy of the PMF.

While we were able to out-perform `gzip` in specific contexts it remains that:

1. The compression rate was measured against an already quantised version of the original data.
2. The time and space complexity of transformer models, as well as electricity costs are significantly higher than `gzip`;
3. The trained transformer models are only capable of compressing the modality they were trained on;
4. The true compression rate should take into account the size of the model.

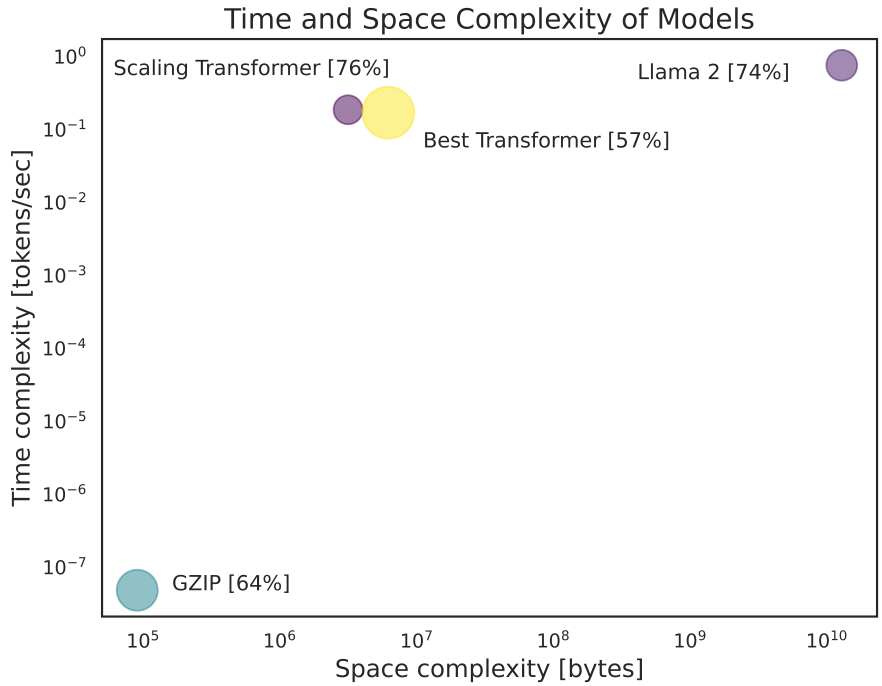


Figure 5.13: Time and space complexity of various models. Node size is proportional to the average compression rate at no added noise and unity scale.

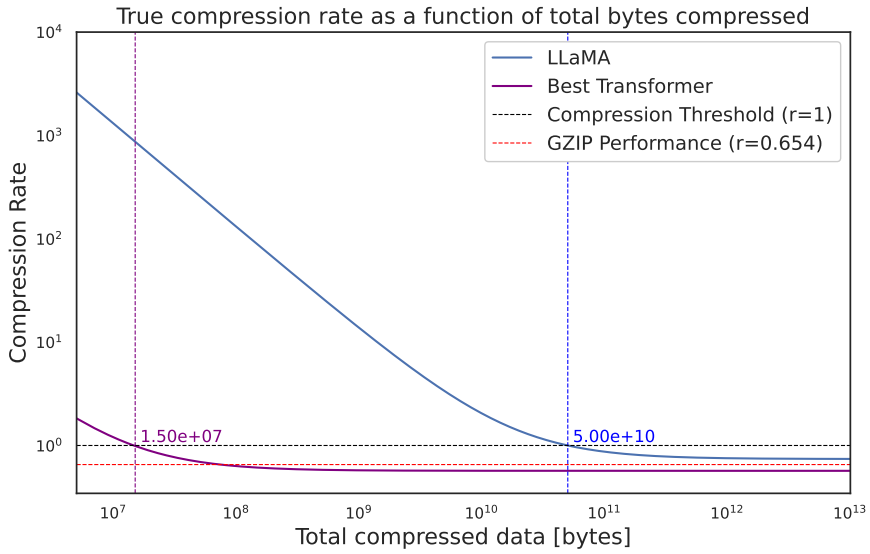


Figure 5.14: True compression rate of LLaMA and the best performing transformer.

Taking in account the size of the model, the equation for the true compression rate is:

$$r = \frac{\text{size(compressed data)} + \text{model size}}{\text{size(original data)}} \tag{5.5}$$

Figure 5.14 illustrates (5.5), plotting the true-compression curves for LLaMA and the best performing transformer. For the transformer, after approximately 15 MB of data has been compressed, and for LLaMA, after 50 GB, the compression threshold is crossed, resulting in a net reduction in the total data size. This figure underscores the aforementioned limitations; the model only becomes useful once a large amount of data is compressed.

5.4 Study limitations

Generated Data The training and testing data were generated in MATLAB using a deterministic process, resulting in data without noise, interference, or multi-path distortion. Real data exhibits such aforementioned properties making it inherently more challenging for transformer training. However, it is worth noting that noisier data would also present greater difficulty for compression methods such as `gzip` as seen in the results of this work.

Aside Data was generated by the WLAN waveform generator module in MATLAB. This data will always generate the same waveform given the same data to encode. The module itself could itself be considered a compressor as we could represent a given waveform by its payload, and the module can regenerate the original data. This was recognised by Kolmogorov who introduced the notion of *Kolmogorov Complexity* [52, 53]. While the

exact Kolmogorov complexity of a string $K(s)$ is uncomfortable it is described as the smallest programme that can produce such a string (Cover and Thomas [17] ch. 7), namely:

$$K(s) := \min\{|p| : U(p) = s\}, \tag{5.6}$$

where p is a programme and $|p|$ is its size. Deviating from the traditional entropic view, this provides another empirical measurement of compression. The representation of such an RF signal can be fully described by the programme which generates it. Here MATLAB can be regarded as both the generator and compressor.

Choice of hyperparameters While certain hyperparameters were careful consideration, many were chosen based on independent experiments that demonstrated their efficacy without accounting for potential interaction with other hyperparameters. Given that the combination of hyperparameters can impact a model's inference and generalisation ability [4], a hyperparameter sweep may have beenwappropriate to perform to ensure we were working with an apt combination. Additionally, the choice of parameters for training with noise in both time and frequency domains were arbitrary. Huepe and Aldana-González [39] suggests a noise threshold in training beyond which a model fails to learn. This threshold was not examined in this study, leaving open the question whether more or less noise in training may have improved generalisation.

Model size and training The size of the training set and transformers were chosen arbitrarily due to constraints time constraints. Larger models demonstrated more efficient compression; however, rapid testing requirements limited the feasibility of multi-epoch trainin. Whether further training on larger datasets would have improved compression remains to be tested. Data scarcity is a constraint, but data augmentation techniques like Mixup [101] and TSMixup for time-series data [2] have been proposed as methods for generating extra data from combinations of existing data when original data is scarce.

5.5 Summary

This chapter analysed results from chapter 4, highlighting how each applied technique contributed to an improved compression rate for the transformer model, as we well as its ability to generalise to noise. It was found that the base transformer could outperform `gzip` at all tested combinations of scale and noise. LLaMA performed well as a compressor in medium-noise environments despite no previous training on WiFi, while headed LLaMA proved ineffective, likely due to structural limitations. The scaling algorithm Algorithm 2 created an uncompetitive compressor as it neither achieved competitive compression nor exhibited resilience to scaling variations. Furthermore, it appears that the scaling factors applied may have been too small to significantly impact models not trained on various scales.

Chapter 6

Conclusion and future research

6.1 Conclusion

This thesis aimed to demonstrate the effectiveness of transformer-based architectures in compressing WiFi 802.11ax data across varying noise levels and scales. The transformer architecture proved well-suited to capturing dependencies within RF data and, with an appropriate training regime, outperformed `gzip` by 7 % in the noiseless case and 21 % at 40 dB SNR. Since the model's compression capability relies on its probabilistic modelling of the data, the proposed models demonstrated greater resilience to noise than `gzip`, which is made less effective by noise disrupting repeated sequences.

Although not explicitly trained on WiFi data, LLaMA 2 showed an ability to recognise the binary phase shift keying patterns in `MCS0` WiFi, leveraging its strong in-context learning to achieve competitive compression (improved by 4.5 %) under medium noise levels (40 dB). Appending a linear head to the final transformer block was explored as a way to reduce overhead from downloading fine-tuned LLaMA weights; however, this approach proved ineffective due to insufficient training and the limitations of a linear architecture.

To address the varying scales in real RF data, a scale-invariant algorithm was proposed. However, this training regime hindered the transformer model's learning and failed to outperform `gzip`. The limited dynamic range of `int8` data meant that the tested scaling factors fell within the noise bounds used in training, leading models not explicitly trained for scale-invariance to exhibit resilience to scale changes.

Transformers offer a powerful model for compression due to their ability to learn conditional dependencies that create low-entropy distributions for arithmetic coding. Their complexity, however, limits practical implementation. The more powerful and generalised a transformer model becomes, the better it may compress across various data-types, at the cost of increased storage and compute resources.

6.2 Future work

Removing alphabet limitation Training the base transformers with an alphabet size of 128 was to maintain a fair comparison with LLaMA which required an ASCII input. For a practical implementation this choice is not strictly necessary, any alphabet size may be used. While a smaller alphabet size was hypothesised to be easier to learn, the quantitative value of this trade-off remains to be determined; this could inform how many bits of the original `int16` may be learned before the PMF grows excessively large.

Compression using numerical inference Ansari et al. [2] demonstrated time-series prediction with LLMs, differing from Delétang et al. [20] by leveraging foundation models' exposure to strings of numerical data therefore combining inductive capabilities with prior training. This contrasts with our method which uses tokens without specific meanings. Using strings of numbers in inference enables the model to attribute semantic value to individual RF measurements. Using LLaMA for this remains challenging, as numbers are tokenised separately (e.g., '123' becomes $[1, 2, 3]$). Extracting a single probability distribution for the arithmetic coder from these independent tokens is yet to be demonstrated.

Smaller models Transformer-based compressors face practical limitations in time and space complexity compared to traditional compression programmes. This work aimed to create a model that outperforms `gzip`, but for practical use a smaller, faster, and less power hungry model is required. Models such as TerDiT [57] which use ternary weights show promising results as lightweight transformer models. Using the full power of LLaMA as a WiFi compressor is unnecessary, we do not require its conversational ability. Further research might work to extract its pure inductive abilities for use as a lighter general-purpose compressor.

Convolution on scale Scale invariance is a challenge across various machine learning domains. One solution is 1D convolution, as surveyed by Kiranyaz et al. [50]. Given the inefficacy of the scale-invariant algorithm proposed, we recommend exploring convolutional neural networks to achieve scale invariance in compression however recognise the added computational cost of such a layer.

Bibliography

[1] Anon. *Supplementary Lempel Ziv Notes*. Northwestern University. Jan. 1, 2004. URL: http://users.eecs.northwestern.edu/~rberry/ECE428/Lectures/lec8_notes.pdf (visited on 09/14/2024).

[2] Abdul Fatir Ansari et al. *Chronos: Learning the Language of Time Series*. May 2, 2024. DOI: 10.48550/arXiv.2403.07815. arXiv: 2403.07815[cs]. URL: http://arxiv.org/abs/2403.07815 (visited on 05/17/2024).

[3] *Arithmetic coding*. In: *Wikipedia*. Page Version ID: 1220110681. Apr. 21, 2024. URL: https://en.wikipedia.org/w/index.php?title=Arithmetic_coding&oldid=1220110681 (visited on 04/26/2024).

[4] Christian Arnold et al. "The role of hyperparameters in machine learning models and how to tune them". In: *Political Science Research and Methods* 12.4 (Oct. 2024), pp. 841–848. ISSN: 2049-8470, 2049-8489. DOI: 10.1017/psrm.2023.61. URL: https://www.cambridge.org/core/journals/political-science-research-and-methods/article/role-of-hyperparameters-in-machine-learning-models-and-how-to-tune-them/27296C04CF5935C55327F11BF4017371 (visited on 10/29/2024).

[5] Chris M. Bishop. "Training with Noise is Equivalent to Tikhonov Regularization". In: *Neural Computation* 7.1 (Jan. 1, 1995), pp. 108–116. ISSN: 0899-7667. DOI: 10.1162/neco.1995.7.1.108. URL: https://doi.org/10.1162/neco.1995.7.1.108 (visited on 10/18/2024).

[6] Karlheinz Brandenburg and Gerhard Stoll. "ISO-MPEG-1 Audio: A Generic Standard for Coding of High-: Quality Digital Audio". In: 1994. URL: https://api.semanticscholar.org/CorpusID:58871544.

[7] Peter F. Brown et al. "The Mathematics of Statistical Machine Translation: Parameter Estimation". In: *Computational Linguistics* 19.2 (1993). Ed. by Julia Hirschberg. Place: Cambridge, MA Publisher: MIT Press, pp. 263–311. URL: https://aclanthology.org/J93-2003 (visited on 08/29/2024).

[8] *bzip2 : Home*. URL: https://sourceware.org/bzip2/ (visited on 10/27/2024).

[9] Tom Carter. "An introduction to information theory and entropy". Lecture Note. Lecture Note. CSU Sante Fe, Mar. 9, 2014. URL: https://csustan.csustan.edu/~tom/Lecture-Notes/Information-Theory/info-lec.pdf (visited on 10/13/2024).

[10] Center for Humans and Machines. *Toolkit for attaching, training, saving and loading of new heads for transformer models*. GitHub. URL: https://github.com/center-for-humans-and-machines/transformer-heads (visited on 10/20/2024).

[11] Guoxuan Chi et al. *RF-Diffusion: Radio Signal Generation via Time-Frequency Diffusion*. Apr. 14, 2024. arXiv: 2404.09140[cs,eess,math]. URL: http://arxiv.org/abs/2404.09140 (visited on 10/04/2024).

[12] Luca Clissa, Mario Lassnig, and Lorenzo Rinaldi. "How big is Big Data? A comprehensive survey of data production, storage, and streaming in science and industry". In: *Frontiers in Big Data* 6 (Oct. 19, 2023), p. 1271639. ISSN: 2624-909X. DOI: 10.3389/fdata.2023.1271639. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC10620515/ (visited on 09/27/2024).

[13] *Coding theory*. In: *Wikipedia*. Page Version ID: 1243423362. Sept. 1, 2024. URL: https://en.wikipedia.org/w/index.php?title=Coding_theory&oldid=1243423362#cite_note-1 (visited on 09/14/2024).

[14] *Comparison of Compression Algorithms*. LinuxReviews. URL: https://linuxreviews.org/Comparison_of_Compression_Algorithms (visited on 10/27/2024).

[15] *Conditional entropy*. In: *Wikipedia*. Page Version ID: 1233991266. July 12, 2024. URL: https://en.wikipedia.org/w/index.php?title=Conditional_entropy&oldid=1233991266 (visited on 10/13/2024).

[16] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine Learning* 20.3 (Sept. 1995), pp. 273–297. ISSN: 0885-6125, 1573-0565. DOI: 10.1007/BF00994018. URL: http://link.springer.com/10.1007/BF00994018 (visited on 10/20/2024).

[17] Thomas Cover and Joy Thomas. *Elements of Information Theory*. Wiley-Interscience, Jan. 7, 2006. ISBN: 978-0-471-24195-9.

[18] D. H. Wolpert. "The Lack of A Priori Distinctions Between Learning Algorithms". In: *Neural Computation* 8.7 (Oct. 1996), pp. 1341–1390. ISSN: 0899-7667. DOI: 10.1162/neco.1996.8.7.1341.

[19] Boris Dayma. *DallE Mega Training Journal*. Weights and Biases. Apr. 13, 2024. URL: https://wandb.ai/dalle-mini/dalle-mini/reports/DALL-E-Mega-Training-Journal--VmlldzoxODMxMDI2 (visited on 10/20/2024).

[20] Grégoire Delétang et al. *Language Modeling Is Compression*. Mar. 18, 2024. DOI: 10.48550/arXiv.2309.10668. arXiv: 2309.10668[cs,math]. URL: http://arxiv.org/abs/2309.10668 (visited on 04/16/2024).

[21] Abhimanyu Dubey et al. *The Llama 3 Herd of Models*. Aug. 15, 2024. DOI: 10.48550/arXiv.2407.21783. arXiv: 2407.21783[cs]. URL: http://arxiv.org/abs/2407.21783 (visited on 09/26/2024).

[22] *Entropy (information theory)*. In: *Wikipedia*. Page Version ID: 1249263979. Oct. 4, 2024. URL: https://en.wikipedia.org/w/index.php?title=Entropy_(information_theory)&oldid=1249263979 (visited on 10/30/2024).

[23] Sean Fox. "Specialised Architectures and Arithmetic for Machine Learning". PhD thesis. School of Electrical and Computer Engineering: The University of Sydney, Jan. 10, 2021.

[24] Ned Freed and Nathaniel S. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. Request for Comments RFC 2046. Num Pages: 44. Internet Engineering Task Force, Nov. 1996. DOI: 10.17487/RFC2046. URL: https://datatracker.ietf.org/doc/rfc2046 (visited on 11/01/2024).

[25] A. Shaji George and A.s George. "A Review of Wi-Fi 6 : The Revolution of 6th Generation Wi-Fi Technology". In: 10 (Oct. 2, 2020), pp. 56–65. DOI: 10.5281/zenodo.7024625.

[26] *Gibbs' inequality*. In: *Wikipedia*. Page Version ID: 1245674693. Sept. 14, 2024. URL: https://en.wikipedia.org/w/index.php?title=Gibbs%27_inequality&oldid=1245674693 (visited on 10/14/2024).

[27] Micah Goldblum et al. *The No Free Lunch Theorem, Kolmogorov Complexity, and the Role of Inductive Biases in Machine Learning*. June 7, 2024. arXiv: 2304.05366[cs]. URL: http://arxiv.org/abs/2304.05366 (visited on 10/20/2024).

[28] Claudio Grandi. "Big experiments computing challenges - High energy physics". CERN, June 5, 2017. URL: https://indico.cern.ch/event/605204/contributions/2440577/attachments/1471783/2277732/Calcolo-HEP-Perugia-20170605.pdf (visited on 10/30/2024).

[29] Nate Gruver et al. *Large Language Models Are Zero-Shot Time Series Forecasters*. Aug. 12, 2024. DOI: 10.48550/arXiv.2310.07820. arXiv: 2310.07820. URL: http://arxiv.org/abs/2310.07820 (visited on 10/25/2024).

[30] *gzip(1) — Arch manual pages*. URL: https://man.archlinux.org/man/gzip.1.en (visited on 10/27/2024).

[31] *gzip(1): compress/expand files - Linux man page*. URL: https://linux.die.net/man/1/gzip (visited on 10/30/2024).

[32] Saeif Al-Hazbi et al. *Radio Frequency Fingerprinting via Deep Learning: Challenges and Opportunities*. Apr. 15, 2024. DOI: 10.48550/arXiv.2310.16406. arXiv: 2310.16406. URL: http://arxiv.org/abs/2310.16406 (visited on 10/28/2024).

[33] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 15, 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735 (visited on 10/20/2024).

[34] Elad Hoffer, Itay Hubara, and Daniel Soudry. *Train longer, generalize better: closing the generalization gap in large batch training of neural networks*. Jan. 1, 2018. arXiv: 1705.08741[stat]. URL: http://arxiv.org/abs/1705.08741 (visited on 10/29/2024).

[35] Jordan Hoffmann et al. *Training Compute-Optimal Large Language Models*. Mar. 29, 2022. DOI: 10.48550/arXiv.2203.15556. arXiv: 2203.15556. URL: http://arxiv.org/abs/2203.15556 (visited on 10/20/2024).

[36] M. Hollenbach et al. *Ultralong-term high-density data storage with atomic defects in SiC*. Oct. 28, 2023. DOI: 10.48550/arXiv.2310.18843. arXiv: 2310.18843. URL: http://arxiv.org/abs/2310.18843 (visited on 10/28/2024).

[37] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (Jan. 1, 1989), pp. 359–366. ISSN: 0893-6080. DOI: 10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[38] Yuhuang Hu et al. *Overcoming the vanishing gradient problem in plain recurrent networks*. July 5, 2019. DOI: 10.48550/arXiv.1801.06105. arXiv: 1801.06105. URL: http://arxiv.org/abs/1801.06105 (visited on 10/20/2024).

[39] Cristián Huepe and Maximino Aldana-González. "Dynamical Phase Transition in a Neural Network Model with Noise: An Exact Solution". In: *Journal of Statistical Physics* 108 (Aug. 2002).

[40] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". In: *Proceedings of the IRE* 40.9 (Sept. 1952). Conference Name: Proceedings of the IRE, pp. 1098–1101. ISSN: 2162-6634. DOI: 10.1109/JRPROC.1952.273898. URL: https://ieeexplore.ieee.org/document/4051119 (visited on 04/16/2024).

[41] *Huffman coding*. In: *Wikipedia*. Page Version ID: 1209911693. Feb. 24, 2024. URL: https://en.wikipedia.org/w/index.php?title=Huffman_coding&oldid=1209911693 (visited on 04/22/2024).

[42] *Huffman Tree - Computer Science Field Guide*. URL: https://www.csfieldguide.org.nz/en/interactives/huffman-tree/ (visited on 04/22/2024).

[43] "IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems Local and Metropolitan Area Networks–Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 1: Enhancements for High-Efficiency WLAN". In: *IEEE Std 802.11ax-2021 (Amendment to IEEE Std 802.11-2020)* (May 19, 2021), pp. 1–767. DOI: 10.1109/IEEESTD.2021.9442429.

[44] ISO. *ISO/IEC 646:1991(en), Information technology — ISO 7-bit coded character set for information interchange*. ISO/IEC 646:1991(en) Information technology — ISO 7-bit coded character set for information interchange. 1991. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:646:ed-3:v1:en (visited on 10/20/2024).

[45] jamesdmccaffrey. *Why You Should Use Cross-Entropy Error Instead Of Classification Error Or Mean Squared Error For Neural Network Classifier Training*. James D. McCaffrey. Nov. 5, 2013. URL: https://jamesmccaffrey.wordpress.com/2013/11/05/why-you-should-use-cross-entropy-error-instead-of-classification-error-or-mean-squared-error-for-neural-network-classifier-training/ (visited on 10/20/2024).

[46] Zhiying Jiang et al. ""Low-Resource" Text Classification: A Parameter-Free Classification Method with Compressors". In: *Findings of the Association for Computational Linguistics: ACL 2023*. Findings 2023. Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 6810–6828. DOI: 10.18653/v1/2023.findings-acl.426. URL: https://aclanthology.org/2023.findings-acl.426 (visited on 09/19/2024).

[47] Jared Kaplan et al. *Scaling Laws for Neural Language Models*. Jan. 23, 2020. arXiv: 2001.08361[cs]. URL: http://arxiv.org/abs/2001.08361 (visited on 10/29/2024).

[48] Nitish Shirish Keskar et al. *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*. Feb. 9, 2017. DOI: 10.48550/arXiv.1609.04836. arXiv: 1609.04836[cs,math]. URL: http://arxiv.org/abs/1609.04836 (visited on 09/19/2024).

[49] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. Jan. 30, 2017. arXiv: 1412.6980[cs]. URL: http://arxiv.org/abs/1412.6980 (visited on 10/19/2024).

[50] Serkan Kiranyaz et al. *1D Convolutional Neural Networks and Applications: A Survey*. May 9, 2019. DOI: 10.48550/arXiv.1905.03554. arXiv: 1905.03554. URL: http://arxiv.org/abs/1905.03554 (visited on 10/29/2024).

[51] Takeshi Kojima et al. *Large Language Models are Zero-Shot Reasoners*. Jan. 29, 2023. DOI: 10.48550/arXiv.2205.11916. arXiv: 2205.11916. URL: http://arxiv.org/abs/2205.11916 (visited on 10/25/2024).

[52] A. N. Kolmogorov. "On Tables of Random Numbers". In: *Sankhyā: The Indian Journal of Statistics, Series A (1961-2002)* 25.4 (1963). Publisher: Springer, pp. 369–376. ISSN: 0581572X. URL: http://www.jstor.org/stable/25049284 (visited on 09/08/2024).

[53] A. N. Kolmogorov. "Three approaches to the quantitative definition of information *". In: *International Journal of Computer Mathematics* 2.1 (Jan. 1968), pp. 157–168. ISSN: 0020-7160, 1029-0265. DOI: 10.1080/00207166808803030. URL: http://www.tandfonline.com/doi/abs/10.1080/00207166808803030 (visited on 10/27/2024).

[54] Jan Kukačka, Vladimir Golkov, and Daniel Cremers. *Regularization for Deep Learning: A Taxonomy*. Oct. 29, 2017. DOI: 10.48550/arXiv.1710.10686. arXiv: 1710.10686. URL: http://arxiv.org/abs/1710.10686 (visited on 10/26/2024).

[55] Ben Langmead. "High order entropy". Lecture Note. Lecture Note. John Hopkins Whiting School of Engineering. URL: https://www.cs.jhu.edu/~langmea/resources/lecture_notes/215_high_order_ent_pub.pdf (visited on 10/13/2024).

[56] Liyuan Liu et al. *On the Variance of the Adaptive Learning Rate and Beyond*. Oct. 26, 2021. DOI: 10.48550/arXiv.1908.03265. arXiv: 1908.03265. URL: http://arxiv.org/abs/1908.03265 (visited on 10/20/2024).

[57] Xudong Lu et al. *TerDiT: Ternary Diffusion Models with Transformers*. May 23, 2024. DOI: 10.48550/arXiv.2405.14854. arXiv: 2405.14854. URL: http://arxiv.org/abs/2405.14854 (visited on 10/29/2024).

[58] Calvin Luo. *Understanding Diffusion Models: A Unified Perspective*. Aug. 25, 2022. arXiv: 2208.11970[cs]. URL: http://arxiv.org/abs/2208.11970 (visited on 10/20/2024).

[59] Mazur. *A Step by Step Backpropagation Example*. Matt Mazur. Mar. 17, 2015. URL: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/ (visited on 10/16/2024).

[60] Suvir Mirchandani et al. *Large Language Models as General Pattern Machines*. Oct. 26, 2023. DOI: 10.48550/arXiv.2307.04721. arXiv: 2307.04721. URL: http://arxiv.org/abs/2307.04721 (visited on 10/20/2024).

[61] Margaret Mitchell et al. "Model Cards for Model Reporting". In: *Proceedings of the Conference on Fairness, Accountability, and Transparency*. Jan. 29, 2019, pp. 220–229. DOI: 10.1145/3287560.3287596. arXiv: 1810.03993[cs]. URL: http://arxiv.org/abs/1810.03993 (visited on 10/21/2024).

[62] Tom M. Mitchell. *Machine Learning*. McGraw-Hill series in computer science. New York: McGraw-Hill, 1997. 414 pp. ISBN: 978-0-07-042807-2.

[63] Takashi Mori and Masahito Ueda. *Improved generalization by noise enhancement*. version: 1. Sept. 28, 2020. DOI: 10.48550/arXiv.2009.13094. arXiv: 2009.13094. URL: http://arxiv.org/abs/2009.13094 (visited on 10/18/2024).

[64] Joy Morris. *1.5: Coding Theory*. Mathematics LibreTexts. June 27, 2021. URL: https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics/Combinatorics_(Morris)/01%3A_Introduction/01%3A_What_is_Combinatorics/1.05%3A_Coding_Theory (visited on 09/14/2024).

[65] Preetum Nakkiran et al. *Deep Double Descent: Where Bigger Models and More Data Hurt*. Dec. 4, 2019. DOI: 10.48550/arXiv.1912.02292. arXiv: 1912.02292. URL: http://arxiv.org/abs/1912.02292 (visited on 10/26/2024).

[66] Mark Nelson. *Data Compression With Arithmetic Coding*. Mark Nelson. Oct. 19, 2014. URL: https://marknelson.us/posts/2014/10/19/data-compression-with-arithmetic-coding.html (visited on 10/20/2024).

[67] OpenAI et al. *GPT-4 Technical Report*. Mar. 4, 2024. DOI: 10.48550/arXiv.2303.08774. arXiv: 2303.08774[cs]. URL: http://arxiv.org/abs/2303.08774 (visited on 09/26/2024).

[68] OpenSLR. *LibriSpeech ASR Corpus*. URL: https://www.openslr.org/12 (visited on 10/20/2024).

[69] *Optimizer Schedules — Optax documentation*. URL: https://optax.readthedocs.io/en/latest/api/optimizer_schedules.html#optax.schedules.cosine_onecycle_schedule (visited on 11/01/2024).

[70] Mary Phuong and Marcus Hutter. *Formal Algorithms for Transformers*. July 19, 2022. arXiv: 2207.09238[cs]. URL: http://arxiv.org/abs/2207.09238 (visited on 10/20/2024).

[71] John G. Proakis and Masoud Salehi. *Digital communications*. 5th ed. Boston: McGraw-Hill, 2008. 1150 pp. ISBN: 978-0-07-295716-7.

[72] Kashif Rasul et al. *Lag-Llama: Towards Foundation Models for Probabilistic Time Series Forecasting*. Feb. 8, 2024. arXiv: 2310.08278[cs]. URL: http://arxiv.org/abs/2310.08278 (visited on 10/20/2024).

[73] J. Rissanen and G. G. Langdon. "Arithmetic Coding". In: *IBM Journal of Research and Development* 23.2 (Mar. 1979). Conference Name: IBM Journal of Research and Development, pp. 149–162. ISSN: 0018-8646. DOI: 10.1147/rd.232.0149. URL: https://ieeexplore.ieee.org/document/5390830 (visited on 04/16/2024).

[74] Danielle Rothermel et al. *Don't Sweep your Learning Rate under the Rug: A Closer Look at Cross-modal Transfer of Pretrained Transformers*. July 26, 2021. DOI: 10.48550/arXiv.2107.12460. arXiv: 2107.12460. URL: http://arxiv.org/abs/2107.12460 (visited on 10/20/2024).

[75] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/323533a0. URL: https://www.nature.com/articles/323533a0 (visited on 10/20/2024).

[76] S. Tridgell et al. "Real-time Automatic Modulation Classification using RFSoC". In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). Journal Abbreviation: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). May 18, 2020, pp. 82–89. DOI: 10.1109/IPDPSW50202.2020.00021.

[77] David Salinas et al. "DeepAR: Probabilistic forecasting with autoregressive recurrent networks". In: *International Journal of Forecasting* 36.3 (July 1, 2020), pp. 1181–1191. ISSN: 0169-2070. DOI: 10.1016/j.ijforecast.2019.07.001. URL: https://www.sciencedirect.com/science/article/pii/S0169207019301888.

[78] Grant Sanderson. *Visualizing Attention, a Transformer's Heart — Chapter 6, Deep Learning*. 2024. URL: https://www.3blue1brown.com/lessons/3blue1brown.com

[79] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. 1st ed. Cambridge University Press, May 19, 2014. ISBN: 978-1-107-05713-5. DOI: 10.1017/CBO9781107298019. URL: https://www.cambridge.org/core/product/identifier/9781107298019/type/book (visited on 10/20/2024).

[80] Claude E. Shannon. "The Mathematical Theory of Communication". In: *The Bell System Technical Journal* Vol 27 (Jan. 7, 1948). URL: https://ieeexplore.ieee.org/document/6773024 (visited on 04/16/2024).

[81] Shardeum Content Team. *Physical Layer in OSI Model - Functions, Importance and Role*. Physical Layer in OSI Model - Functions, Importance and Role. Sept. 19, 2022. URL: https://shardeum.org/blog/physical-layer-in-osi-model/ (visited on 10/20/2024).

[82] Leslie N. Smith. "Cyclical Learning Rates for Training Neural Networks". In: *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*. 2017 IEEE Winter Conference on Applications of Computer Vision (WACV). Mar. 2017, pp. 464–472. DOI: 10.1109/WACV.2017.58. URL: https://ieeexplore.ieee.org/document/7926641 (visited on 10/29/2024).

[83] Leslie N. Smith and Nicholay Topin. *Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates*. May 17, 2018. arXiv: 1708.07120[cs]. URL: http://arxiv.org/abs/1708.07120 (visited on 11/01/2024).

[84] *Softmax function*. In: *Wikipedia*. Page Version ID: 1251409788. Oct. 16, 2024. URL: https://en.wikipedia.org/w/index.php?title=Softmax_function&oldid=1251409788 (visited on 10/20/2024).

[85] Derya Soydaner. "Attention Mechanism in Neural Networks: Where it Comes and Where it Goes". In: *Neural Computing and Applications* 34.16 (Aug. 2022), pp. 13371–13385. ISSN: 0941-0643, 1433-3058. DOI: 10.1007/s00521-022-07366-3. arXiv: 2204.13154[cs]. URL: http://arxiv.org/abs/2204.13154 (visited on 10/20/2024).

[86] Jianlin Su et al. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. Nov. 8, 2023. DOI: 10.48550/arXiv.2104.09864. arXiv: 2104.09864. URL: http://arxiv.org/abs/2104.09864 (visited on 10/22/2024).

[87] Shan Suthaharan. "Support Vector Machine". In: *Machine Learning Models and Algorithms for Big Data Classification: Thinking with Examples for Effective Learning*. Ed. by Shan Suthaharan. Boston, MA: Springer US, 2016, pp. 207–235. ISBN: 978-1-4899-7641-3. DOI: 10.1007/978-1-4899-7641-3_9. URL: https://doi.org/10.1007/978-1-4899-7641-3_9.

[88] *The .xz file format*. URL: https://tukaani.org/xz/format.html (visited on 10/27/2024).

[89] The MathWorks Inc. *MATLAB version: 9.13.0 (R2022b)*. Version 9.13.0. Natick, Massachusetts, United States, 2020. URL: https://www.mathworks.com.

[90] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. July 19, 2023. DOI: 10.48550/arXiv.2307.09288. arXiv: 2307.09288[cs]. URL: http://arxiv.org/abs/2307.09288 (visited on 09/27/2024).

[91] *Using gzip and gunzip in Linux — Baeldung on Linux*. May 29, 2020. URL: https://www.baeldung.com/linux/gzip-and-gunzip (visited on 10/30/2024).

[92] Ashish Vaswani et al. *Attention Is All You Need*. Aug. 1, 2023. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762[cs]. URL: http://arxiv.org/abs/1706.03762 (visited on 04/16/2024).

[93] Pablo Villalobos et al. "Will we run out of data? Limits of LLM scaling based on human-generated data". In: ().

[94] Gregory K Wallace. "The JPEG Still Picture Compression Standard". In: *IEEE Transactions on Consumer Electronics* (Jan. 12, 1991).

[95] *Wi-Fi 6 (802.11ax) Technical Guide*. Cisco Meraki Documentation. Jan. 19, 2021. URL: https://documentation.meraki.com/MR/Wi-Fi_Basics_and_Best_Practices/Wi-Fi_6_(802.11ax)_Technical_Guide (visited on 10/19/2024).

[96] Ian H. Witten, Radford M. Neal, and John G. Cleary. "Arithmetic coding for data compression". In: *Communications of the ACM* 30.6 (June 1987), pp. 520–540. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/214762.214771. URL: https://dl.acm.org/doi/10.1145/214762.214771 (visited on 04/26/2024).

[97] Thomas Wolf et al. *HuggingFace's Transformers: State-of-the-art Natural Language Processing*. July 14, 2020. DOI: 10.48550/arXiv.1910.03771. arXiv: 1910.03771. URL: http://arxiv.org/abs/1910.03771 (visited on 10/23/2024).

[98] Yonghui Wu et al. *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. Oct. 8, 2016. DOI: 10.48550/arXiv.1609.08144. arXiv: 1609.08144[cs]. URL: http://arxiv.org/abs/1609.08144 (visited on 08/29/2024).

[99] Ruibin Xiong et al. *On Layer Normalization in the Transformer Architecture*. June 29, 2020. DOI: 10.48550/arXiv.2002.04745. arXiv: 2002.04745. URL: http://arxiv.org/abs/2002.04745 (visited on 10/20/2024).

[100] Chiyuan Zhang et al. *Understanding deep learning requires rethinking generalization*. Feb. 26, 2017. arXiv: 1611.03530[cs]. URL: http://arxiv.org/abs/1611.03530 (visited on 10/26/2024).

[101] Hongyi Zhang et al. *mixup: Beyond Empirical Risk Minimization*. Apr. 27, 2018. DOI: 10.48550/arXiv.1710.09412. arXiv: 1710.09412. URL: http://arxiv.org/abs/1710.09412 (visited on 10/27/2024).

[102] Qiquan Zhang et al. *An Empirical Study on the Impact of Positional Encoding in Transformer-based Monaural Speech Enhancement*. Feb. 13, 2024. DOI: 10.48550/arXiv.2401.09686. arXiv: 2401.09686. URL: http://arxiv.org/abs/2401.09686 (visited on 10/22/2024).

[103] Bohan Zhuang et al. *A Survey on Efficient Training of Transformers*. May 4, 2023. arXiv: 2302.01107[cs]. URL: http://arxiv.org/abs/2302.01107 (visited on 10/20/2024).

[104] J Ziv and A Lempel. "A universal algorithm for sequential data compression". In: *IEEE Transactions on Information Theory* 23 (Jan. 5, 1977). URL: https://ieeexplore.ieee.org/document/1055714 (visited on 04/16/2024).

Appendices

Appendix A

Model cards

The model cards [61] are given below. Models that were trained but not presented in this report are omitted.

MODEL CARD - naive_model

HYPERPARAMETERS:
Num Heads: 8
Layers: 16
Embedding Dimension: 64
Alphabet Size: 128
Widening Factor: 4
Positional Embedding: SPE

TRAINING:
Noise: False
Stride: Block
Scaled: False

EPOCHS:
1: Learning Rate: 1e−5, **Learning Schedule:** Constant AdamW, **Iterations:** 25e5, **Batch Size:** 32

MODEL CARD - noise_cos

HYPERPARAMETERS:
Num Heads: 8
Layers: 16
Embedding Dimension: 64
Alphabet Size: 128
Widening Factor: 4
Positional Embedding: SPE

TRAINING:
Noise: True
Stride: Block
Scaled: False

EPOCHS:
1: Learning Rate: $1e-5 \rightarrow 1e-3 \rightarrow 1e-6$, **Learning Schedule:** RU + Cos Anneal, **Iterations:** 25e5, **Batch Size:** 32

MODEL CARD - noise_model

HYPERPARAMETERS:
Num Heads: 8
Layers: 16
Embedding Dimension: 64
Alphabet Size: 128
Widening Factor: 4
Positional Embedding: SPE

TRAINING:
Noise: True
Stride: Block
Scaled: False

EPOCHS:
1: Learning Rate: $1e-5$, **Learning Schedule:** AdamW, **Iterations:** 25e5, **Batch Size:** 32

MODEL CARD - noise_cos_rope

HYPERPARAMETERS:
Num Heads: 4
Layers: 8
Embedding Dimension: 64
Alphabet Size: 128
Widening Factor: 4
Positional Embedding: RoPE

TRAINING:
Noise: True
Stride: Block
Scaled: False

EPOCHS:
1: Learning Rate: $1e-5 \rightarrow 1e-3 \rightarrow 1e-6$, **Learning Schedule:** RU + Cos Anneal, **Iterations:** 25e5, **Batch Size:** 32

MODEL CARD - final_big

HYPERPARAMETERS:
Num Heads: 4
Layers: 8
Embedding Dimension: 128
Alphabet Size: 128
Widening Factor: 4
Positional Embedding: RoPE

TRAINING:
Noise: True
Stride: Unit
Scaled: False

EPOCHS:
1: Learning Rate: 1e−4 → 1e−3 → 1e−5, **Learning Schedule:** RU + Cos Anneal, **Iterations:** 2e6, **Batch Size:** 16
2: Learning Rate: 1e−5 → 1e−4 → 1e−6,, **Learning Schedule:** RU + Cos Anneal, **Iterations:** 1e6, **Batch Size:** 32
3: Learning Rate: 1e−6 → 1e−5 → 1e−7, **Learning Schedule:** RU + Cos Anneal, **Iterations:** 500e3, **Batch Size:** 64

MODEL CARD - final_small

HYPERPARAMETERS:
Num Heads: 4
Layers: 4
Embedding Dimension: 64
Alphabet Size: 128
Widening Factor: 4
Positional Embedding: RoPE

TRAINING:
Noise: True
Stride: Unit
Scaled: False

EPOCHS:
1: Learning Rate: 1e−4 → 1e−3 → 1e−5, **Learning Schedule:** RU + Cos Anneal, **Iterations:** 2e6, **Batch Size:** 16
2: Learning Rate: 1e−5 → 1e−4 → 1e−6,, **Learning Schedule:** RU + Cos Anneal, **Iterations:** 1e6, **Batch Size:** 32
3: Learning Rate: 1e−6 → 1e−5 → 1e−7, **Learning Schedule:** RU + Cos Anneal, **Iterations:** 500e3, **Batch Size:** 64

MODEL CARD - single_scale

HYPERPARAMETERS:
Num Heads: 4
Layers: 8
Embedding Dimension: 128
Alphabet Size: 128
Widening Factor: 4
Positional Embedding: RoPE

TRAINING:
Noise: True
Stride: Unit
Scaled: True

EPOCHS:
1: Learning Rate: 1e−4 → 1e−3 → 1e−5, **Learning Schedule:** RU + Cos Anneal, **Iterations:** 2e6, **Batch Size:** 16
2: Learning Rate: 1e−5 → 1e−4 → 1e−6,, **Learning Schedule:** RU + Cos Anneal, **Iterations:** 1e6, **Batch Size:** 32
3: Learning Rate: 1e−6 → 1e−5 → 1e−7, **Learning Schedule:** RU + Cos Anneal, **Iterations:** 500e3, **Batch Size:** 64

Appendix B

Raw model results

SNR / Scale	1.1	1.0	0.9	0.8	0.7
10	88.95 %	87.17 %	85.20 %	83.05 %	80.77 %
20	87.62 %	85.85 %	83.89 %	81.82 %	79.61 %
30	87.29 %	85.46 %	83.45 %	81.29 %	78.94 %
40	82.96 %	80.66 %	78.11 %	75.32 %	72.46 %
∞	65.37 %	64.06 %	62.64 %	61.13 %	59.52 %

Table B.1: Compression results for `gzip`.

SNR / Scale	1.1	1.0	0.9	0.8	0.7
10	88.93 %	87.07 %	85.18 %	83.53 %	82.08 %
20	88.02 %	86.29 %	84.56 %	83.04 %	81.70 %
30	87.93 %	86.22 %	84.50 %	82.99 %	81.67 %
40	87.93 %	86.21 %	84.49 %	82.99 %	81.66 %
∞	87.93 %	86.21 %	84.49 %	82.99 %	81.65 %

Table B.2: Compression results for naive arithmetic coding.

SNR / Scale	1.1	1.0	0.9	0.8	0.7
10	88.25 %	86.33 %	84.60 %	82.95 %	81.26 %
20	87.22 %	85.47 %	83.78 %	82.17 %	80.82 %
30	87.05 %	85.36 %	83.83 %	82.19 %	80.69 %
40	87.12 %	85.39 %	83.74 %	82.25 %	80.86 %
∞	87.09 %	85.50 %	83.67 %	82.23 %	80.75 %

Table B.3: Compression results for naive transformer (`naive_model`).

SNR / Scale	1.1	1.0	0.9	0.8	0.7
10	89.96 %	87.99 %	85.91 %	83.81 %	81.89 %
20	88.80 %	86.91 %	84.85 %	83.05 %	81.29 %
30	88.67 %	86.79 %	84.87 %	83.00 %	81.09 %
40	88.66 %	86.84 %	84.90 %	83.04 %	81.11 %
∞	88.72 %	86.76 %	84.93 %	83.05 %	81.16 %

Table B.4: Compression results for noisy transformer (`noise_model`).

SNR / Scale	1.1	1.0	0.9	0.8	0.7
10	89.51 %	87.58 %	85.66 %	83.62 %	81.43 %
20	87.50 %	85.14 %	83.65 %	81.36 %	79.53 %
30	86.63 %	84.29 %	82.72 %	80.78 %	78.46 %
40	86.32 %	84.34 %	82.43 %	80.78 %	78.56 %
∞	86.13 %	84.00 %	82.39 %	80.26 %	78.32 %

Table B.5: Compression results for noisy transformer with cosine annealing (`noise_cos`).

SNR / Scale	1.1	1.0	0.9	0.8	0.7
10	89.13 %	87.23 %	85.13 %	82.95 %	81.09 %
20	86.63 %	84.70 %	82.86 %	81.48 %	79.62 %
30	85.45 %	83.43 %	82.44 %	80.22 %	78.82 %
40	85.04 %	83.85 %	82.16 %	80.08 %	78.98 %
∞	85.10 %	83.03 %	82.04 %	80.06 %	78.61 %

Table B.6: Compression results for noisy transformer with cosine annealing and RoPE (`noise_cos_rope`).

SNR / Scale	1.1	1.0	0.9	0.8	0.7
10	87.31 %	85.25 %	83.17 %	81.12 %	79.14 %
20	83.09 %	80.91 %	79.40 %	77.67 %	75.59 %
30	81.20 %	80.25 %	77.73 %	76.35 %	74.43 %
40	80.67 %	79.44 %	77.74 %	75.95 %	74.27 %
∞	81.03 %	78.80 %	77.55 %	75.28 %	74.54 %

Table B.7: Compression results for small model (`final_small`).

SNR / Scale	1.1	1.0	0.9	0.8	0.7
10	83.67 %	81.33 %	78.98 %	76.86 %	74.92 %
20	72.99 %	70.90 %	69.13 %	67.46 %	65.75 %
30	65.78 %	63.92 %	62.42 %	61.36 %	59.69 %
40	61.44 %	59.87 %	58.56 %	57.78 %	56.77 %
∞	57.59 %	56.85 %	56.13 %	53.99 %	53.99 %

Table B.8: Compression results for final transformer model (`final_big`).

SNR / Scale	0.7	0.8	0.9	1.0	1.1
10	82.49 %	85.33 %	85.87 %	85.01 %	83.89 %
20	80.59 %	82.13 %	84.14 %	85.61 %	86.53 %
30	76.31 %	79.39 %	81.05 %	81.43 %	82.42 %
40	71.49 %	73.04 %	73.87 %	76.13 %	77.66 %
∞	69.26 %	70.34 %	73.76 %	71.58 %	69.80 %

Table B.9: Compression results for LLaMA

SNR / Scale	0.7	0.8	0.9	1.0	1.1
10	81.9 %	83.4 %	85.7 %	87.8 %	90.1 %
20	81.1 %	82.4 %	84.8 %	86.3 %	89.0 %
30	77.7 %	80.7 %	83.4 %	84.5 %	86.6 %
40	73.1 %	75.9 %	78.2 %	79.3 %	82.4 %
∞	70.9 %	72.1 %	74.3 %	75.3 %	76.2 %

Table B.10: Compression results for Headed LLaMA

SNR / Scale	**0.7**	**0.8**	**0.9**	**1.0**	**1.1**
10	81.9 %	83.4 %	85.7 %	87.8 %	90.1 %
20	81.1 %	82.4 %	84.8 %	86.3 %	89.0 %
30	77.7 %	80.7 %	83.4 %	84.5 %	86.6 %
40	73.1 %	75.9 %	78.2 %	79.3 %	82.4 %
∞	70.9 %	72.1 %	74.3 %	75.3 %	76.2 %

Table B.11: Compression results for Headed LLaMA

Scale / σ	**1.1**	**1.0**	**0.9**	**0.8**	**0.7**	**0.6**
3.0	73.8 %	79.94 %	76.44 %	79.82 %	79.72 %	78.82 %
2.0	71.64 %	78.58 %	75.4 %	79.08 %	79.64 %	78.14 %
1.5	70.16 %	77.64 %	74.64 %	78.56 %	79.54 %	78.08 %
1.0	68.22 %	76.4 %	73.58 %	77.86 %	79.26 %	77.98 %

Table B.12: Compression results for single-scaled model (`single_scale`).