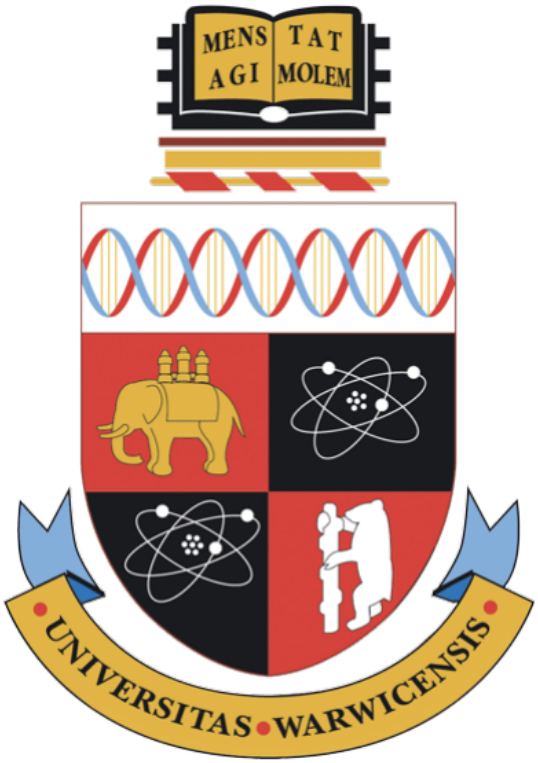
Motifcaller: A Machine Learning-Driven Decoding Tool for DNA Storage Data

Dev Thakkar

Third Year (24/25) CS344 Discrete Mathematics Project

April, 2025

Supervisor: Long Tran-Thanh



Keywords: Bioinformatics, Computational Biology, Convolutional Neural Networks, Deep Learning, LSTM (Long Short-Term Memory), Machine Learning, Time Series Data, Transformer

Abstract

Global data generation is accelerating rapidly, with over half of this data stored in cold storage. Deoxyribonucleic Acid (DNA) storage is becoming more popular as a long-term data archival solution as it offers the potential for significantly improved density, durability and sustainability. Decoding stored data from DNA relies on a process called basecalling, where raw nanopore electrical signals are converted back to individual DNA bases (then back to digital bits).

Current novel DNA synthesis methods used in DNA storage require the signals to be converted to payload motifs (collection of bases which correspond to our digital bits) as opposed to individual bases. Therefore current pipelines also require sequence alignment which can be used to identify motifs from individual bases. However, this two stage process method is slow and overly complicated.

In this work, we present *Motifcaller*, an end-to-end neural decoder that maps raw nanopore squiggles directly to sequences of payload motifs, eliminating the traditional basecalling and alignment stages. By making a model to directly identify the motifs removes any obsolete additional step and streamlines the process. Our experiments on synthetic data prove CNN–Transformer–GRU with attention, CNN–Transformer–CTC, and CNN–BiGRU–CTC achieve over 99% motif accuracy and over 99% sequence accuracy. CNN–BiGRU–CTC emerges as the best-rounded architecture when considering overall accuracy, inference time and GPU memory usage. When trained using hyperparameter tuning, the CNN–BiGRU–CTC variant reached 99.96% token and 99.64% sequence accuracy with 7 h 31 m of training and ≈ 6 ms per-read inference. These results confirm that end-to-end squiggle-to-motif decoding can achieve near-perfect accuracy with real-time performance and practical resource requirements, eliminating the need for separate basecalling and alignment stages in DNA storage retrieval.

Acknowledgements

I would like to thank Professor Long Tran-Thanh, my supervisor, for his guidance, advice, and the time he dedicated to supporting this project. I am also grateful to Professor Greg Watson for helping me collaborate with Helixworks and as the module organiser of CS344.

Special thanks to Nimesh Pinnamaneni, CEO of Helixworks, for taking the time to explain industry insights and highlighting key decoding challenges, which were crucial in shaping this research.

Thank you to Dr Chris Takahashi, Research Scientist at the University of Washington, for providing valuable early guidance about the DNA storage industry. Additionally, I appreciate Professor Julia Brettschneider's advice on nanopore sequencing data and Professor Andrew Bowman's insights into generating realistic synthetic nanopore signals.

Finally, thanks to my friends, family, and particularly Jake Turrell and Gabriela Surowiec for their support throughout this project.

Contents

Introduction

1.1 Background and Motivation

Global data generation reached approximately 123 zettabytes in 2023 and is projected to surge to 396 zettabytes by 2028 [1]. As illustrated in Figure 1, this exponential growth places significant strain on conventional data storage technologies, exacerbating the gap between data generation and storage capacity [2]. A substantial portion of this data, around 60–80%, consists of rarely accessed, or "cold" data [3].

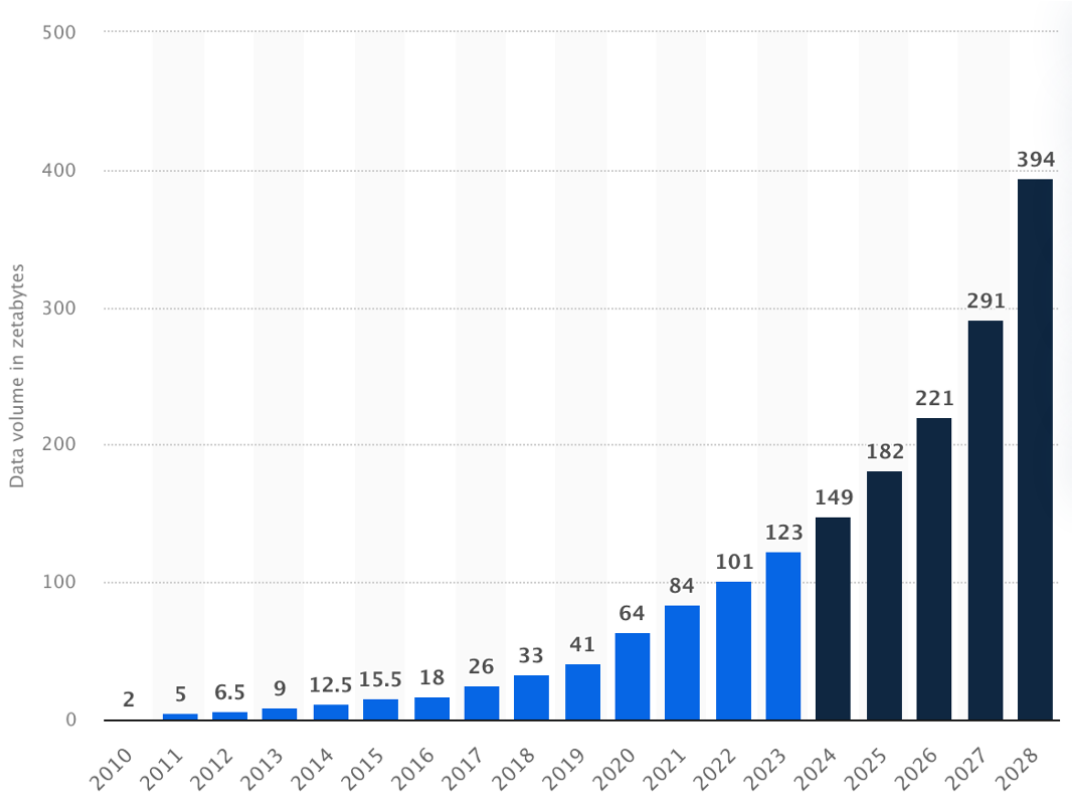
To contextualise, one zettabyte equals 1×10^{21} bytes, equivalent to the storage capacity of about 250 billion DVDs [4]. Traditional cold storage methods such as magnetic tapes and optical or hard drives are becoming increasingly impractical due to the vast physical space, high energy demands, and limited durability, typically spanning a few decades at most. Consequently, data requires frequent migration to new storage media, further compounding resource inefficiencies.

These limitations have driven research into alternative solutions, with DNA storage emerging as a particularly promising candidate. DNA storage theoretically offers densities up to 455 exabytes per gram and the capacity to preserve data integrity over millennia under optimal conditions [5]. This significantly surpasses current silicon-based or magnetic technologies.

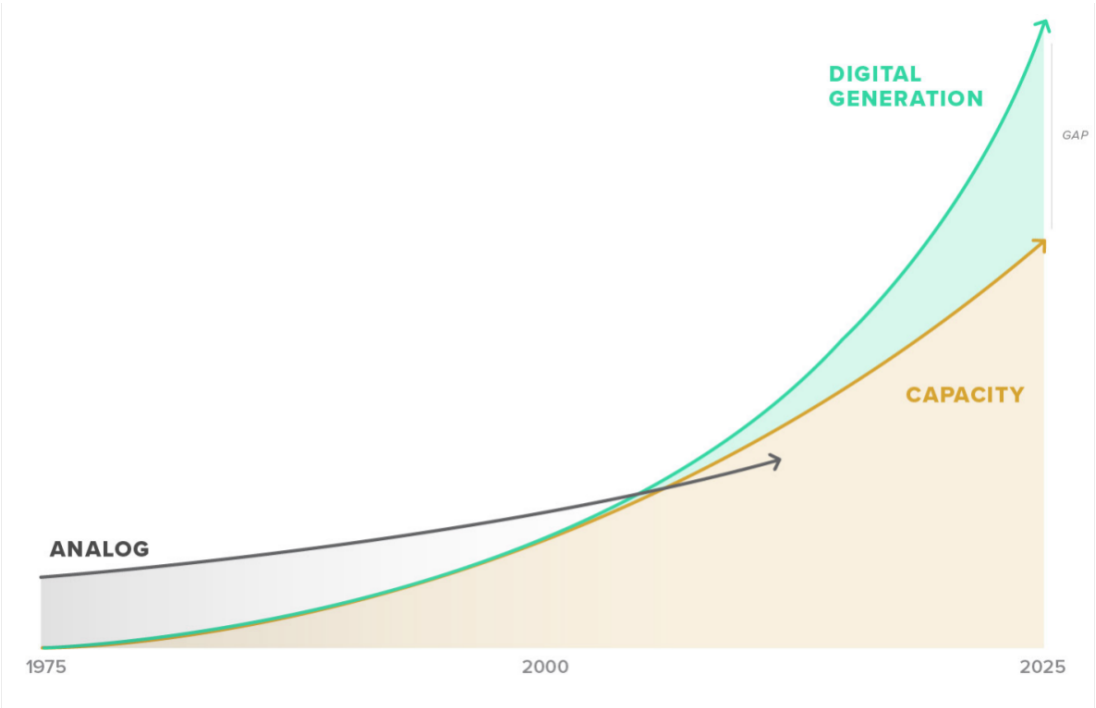
However, practical implementation faces substantial challenges, notably the high cost of synthesising DNA sequences. An innovative solution to mitigate this issue involves motif-based combinatorial synthesis, where short, predefined DNA sequences (motifs) are combined to form longer data-carrying strands [6]. Similarly to assembling words from letters, this approach simplifies synthesis, reduces cost, and improves logical density [6]. A simplified example of the entire encoding and decoding pipeline for motif-based DNA storage can be found in Appendix A.1.

Decoding stored data traditionally involves nanopore sequencing, which generates raw electrical "squiggle" signals that are later translated into nucleotide bases (basecalling), followed by sequence alignment to extract payload motifs. This two-step process, highlighted in Appendix A.2, is time-consuming and error-prone.

This report introduces *Motifcaller*, a novel decoding tool that directly maps nanopore signals to payload motifs using machine learning techniques inspired by Automatic Speech Recognition (ASR). ASR methods naturally align with nanopore signal decoding due to structural similarities between speech signals and DNA sequencing data. Through testing on synthetically generated nanopore data, this proof-of-concept demonstrates that streamlined, ASR-derived methods achieve near-perfect accuracy in motif identification, presenting a significant advancement towards practical DNA storage systems.



(a) Annual global data footprint (in zettabytes) for 2010–2023, with projections up to 2028, showcasing the exponential growth in data generation, duplication and consumption [1].



(b) Trajectories of analogue versus digital information output over time, with global storage manufacturing capacity; by 2010, storage production was already falling behind the digital data creation [7].

Figure 1: Contrasting the acceleration of data production with the increase of storage capacity. (a) maps the forecasted rise in worldwide data volumes, while (b) illustrates how storage output has struggled to keep pace, leading to a capacity gap.

Initially, we explore the broader landscape of digital storage and establish necessary foundational knowledge related to DNA storage and nanopore sequencing technology (Chapter 2). Next, we provide a detailed review of the literature, highlighting state-of-the-art methods in Automatic Speech Recognition that inspire our decoding approach (Chapter 3). We then describe the specific objectives, software, tools, and project management strategies employed (Chapters 4 and 5), followed by the generation of synthetic data and the experimental framework used (Chapter 6). The methodology detailing architecture decisions, normalisation strategies, and training procedures is presented next (Chapter 7). Finally, we analyse and discuss our results, reflecting on key insights and limitations before concluding with potential future directions (Chapters 8 and 9).

Background

This chapter presents the fundamental background required to contextualise DNA-based storage within the broader digital storage landscape. We first explore conventional cold storage technologies, their benefits, and their limitations, setting the stage for introducing DNA as a viable storage medium. The subsequent sections delve into DNA's biological foundations, the challenges and opportunities of DNA-based storage, and the specific processes involved in encoding, synthesising, and sequencing DNA data. This foundation is important for understanding the literature review and state-of-the-art methods explored in Chapter 3, which will cover traditional DNA decoding techniques and modern machine learning approaches to improve data retrieval from DNA.

2.1 Digital Storage Landscape

Modern data storage frequently involves tiered strategies categorising data by access frequency: frequently accessed or "hot" data typically resides on fast but costly storage mediums like Solid-State Drives (SSDs), whereas infrequently accessed or "cold" data is archived on more affordable, higher-capacity solutions like magnetic tape or optical disks [8]. Cold storage archives optimise density and cost-efficiency but face significant challenges, including capacity, energy consumption, physical footprint, and longevity, summarised in Table 1.

As illustrated in Figure 2, the market for cold storage solutions continues to expand exponentially due to increasing global data production. Effective cold storage systems should:

- Capture and preprocess data from multiple different sources.
- Transform and clean data for optimal storage.
- Enable efficient retrieval through metadata indexing.
- Provide secure and efficient data access.
- Implement monitoring, backup, and migration practices.

Metric (Source Year)	Magnetic Tape (LTO) (2023)	Optical Storage (Blu-ray/Archival Disc) (2021)	Hard Disc Drive (Cold/Nearline) (2024)	DNA-Based Storage (2024)
Storage Density	18TB native (uncompressed) or 45TB compressed capacity per cartridge with volumetric density 80TB/L [9].	Single Blue-ray Disc Recordable (BD-R) holds 25-100GB per disc. Sony Optical Disc Archive (ODA) cartridge can hold 5.5TB [10].	Record of commercially purchasable capacity of up to 32TB per drive [11].	Theoretical: 455EB/g [5]. Practical demos MB–GB.
Cost-Efficiency	~$2.63/TB ($79 per 30TB cartridge) [10]. Near-zero idle cost. Lowest total cost of ownership [9].	BD-R costs ~$106.50/TB (25-pack 25GB disc for $65). ODA costs ~$33.45/TB (5.5TB cartridge for $184) [10].	~$20/TB (22TB HDD for $499.99) [12]. $5–$14/TB/yr operating [13]. Higher lifetime cost.	DNA Synthesis alone is predicted to be $800M/TB on average[14].
Capacity and Scalability	Highly scalable (E.g. 351PB+ in multi-rack) [15].	BD-R scale to tens of PBs with hundreds of discs. ODA support 165TB standalone and can stack to reach 2.9PB [10].	EB-scale possible (e.g. cloud), but expensive. Add drives/enclosures.	Theoretically unlimited. In 2019, experiment lead to storing ~16GB [16].
Energy (Idle)	~0W idle. Drives only power during access [9]. Estimated 1PB of tape storage uses 300W [17].	~0W idle. Power needed only during access [18].	4–8W idle/drive [19]. Large arrays = high energy use even when idle. Estimated 1PB of HDD Storage uses 3500W [17].	0W idle. DNA stores info with no power; only read/write uses energy.
Physical Footprint	Up to 8.76PB of uncompressed data in a 10sqft library [20]. ~13PB per rack. High TB/m^3 density.	Moderate density. Individual discs slim, but high capacity requires many discs and robotic systems.	1480Gbits per square inches [11]. Cooling and power increase footprint.	Extremely small. Entire EB-scale archives in vials. Minimal footprint [5].
Longevity & Durability	30 years under optimum environmental conditions [9]. Realistically 10-20 years [21]. Very stable offline. Low maintenance.	BD-R up to 50 years and Sony ODA up to 100 years [10]. Resistant to magnetic/environmental decay.	5–10 years typical [9]. Mechanical failure risk. Needs active maintenance.	Thousands of years [22]. Stable in dry, cold conditions.

Table 1: Comparison of Cold Storage Technologies: Magnetic Tape, Optical Discs, Hard Disc Drive (HDD), and DNA-Based Storage.

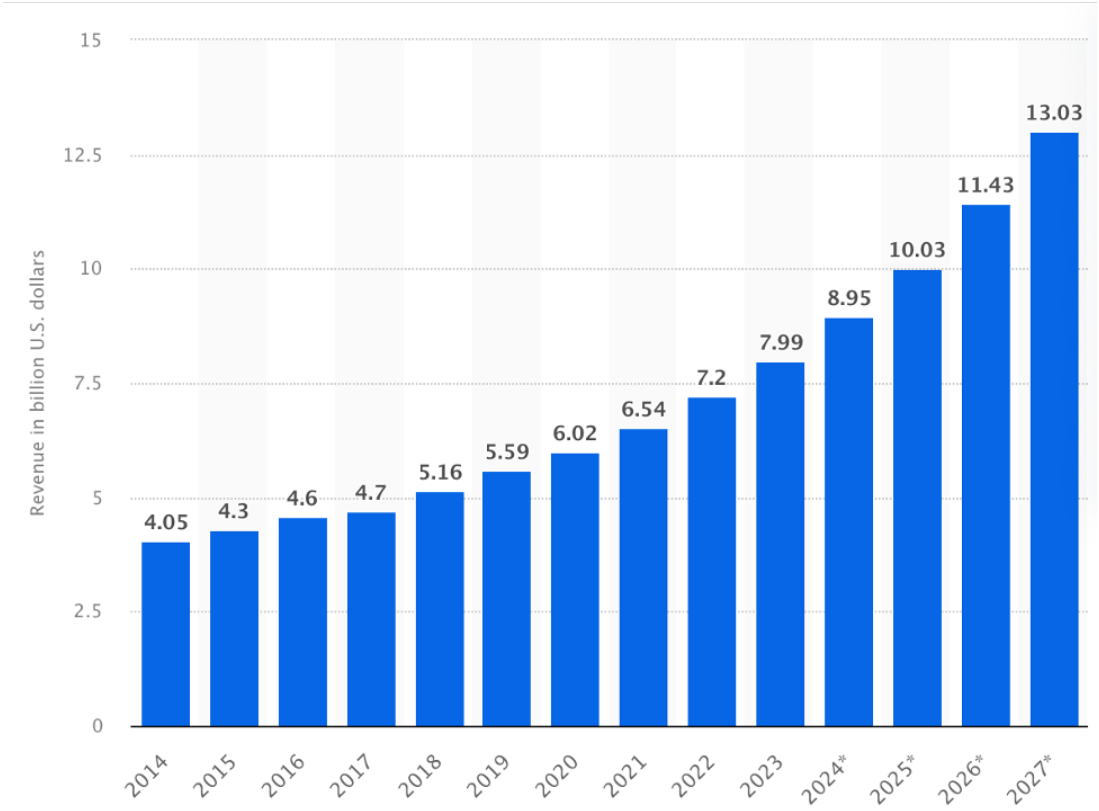
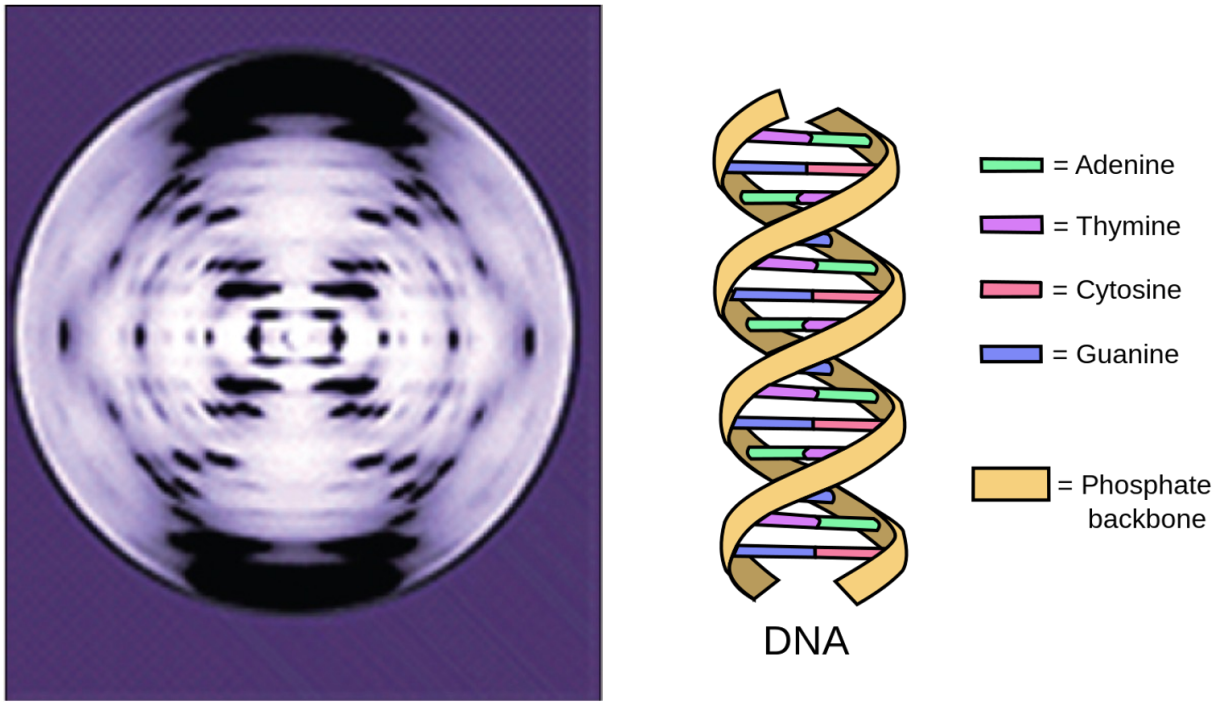


Figure 2: Annual global revenue from the data archival market (in billion U.S. dollars) for 2014–2023, with projections up to 2027, showcasing the exponential growth in the data archival market [23].

2.2 Introduction to DNA as a Storage Medium

DNA has gained recognition as a revolutionary storage medium because of its stability and long-term data preservation capabilities, theoretically lasting thousands of years without degradation.

In the 1950s, Francis Crick and James Watson, aided by the hints from Figure 3a, revealed that DNA is made up of two strands which intertwine to form the double helix [24]. This helix, shown in Figure 3b, was built up of sugar and phosphate groups with nitrogenous bases, Adenine (A), Thymine (T), Guanine (G) and Cytosine (C) which pair in a complementary manner (A with T, and G with C). This initial discovery revealed DNA as a carrier of genetic information and in the 1960s, Norbert Wiener and Mikhail Neirman sprung conversation of using the "genetic memory" [2]. DNA storage was first showcased experimentally in 1988 by artist Joe Davis who converted information from an image to a 28-base-pair DNA sequence [25]. A substantial change occurred in the early 2010s when George Church and Nick Goldman's work demonstrated that DNA could be used to reliably store large volumes of data [25]. They were able to encode hundreds of kilobytes of diverse data types, from text and images to sounds and PDFs [25]. New DNA storage are continuously proposed as DNA synthesis and sequencing techniques improve and help bridge DNA storage from a theoretical idea to a realistic appropriate option for storage.



(a) X-ray diffraction pattern of DNA which consists of an X-shaped blur hinting at its double-helix structure [24].

(b) Visual representation of DNA structure, highlighting its double-helix form, molecular composition and base pairs [24].

Figure 3: Key images illustrating the DNA's structural discovery. (a) presents a detailed diagram of a double helix and its molecular components. (b) showcases diffraction photo where the pattern was important to deduce the helix property of DNA.

Despite these advancements, DNA storage still faces challenges:

- **Cost:** High costs of DNA synthesis and sequencing [22]. Synthesis costs alone are estimated to be ~$0.0001 per base which results in a cost of $800 million/TB [2].
- **Speed:** Encoding and decoding speeds are slower compared to conventional digital storage due to the time-consuming nature of DNA synthesis and sequencing [22].
- **Error rates:** Imperfections in DNA synthesis, sequencing and sequencers produce errors [22].
- **Ethics and Regulation:** Potential ethical and legal issues regarding synthesis and storage of DNA containing digital information [22].
- **Standardisation:** As various methodologies are being explored in encoding, decoding and storing for DNA storage to find the best options, there is a lack of standards which may help in widespread adoption [22].

One effective strategy for tackling these issues is to improve both DNA synthesis and sequencing processes and the methods by which we utilise them. Once breakthrough techniques are developed to overcome cost-related hurdles, these approaches can be expected to set industry standards.

2.3 DNA Synthesis and Sequencing

Using DNA as a data medium involves two core processes: DNA synthesis (writing information by creating DNA strands with specific sequences) and DNA sequencing (reading information back by determining the sequence of DNA strands).

Typically, DNA synthesis involves the sequential addition of nucleotides to build a strand of DNA. However, to tackle the high costs incurred in each synthesis cycle, a promising alternative to be used in DNA Storage is motif-based combinatorial synthesis [6]. This method uses short prefabricated DNA motifs as modular building blocks to assemble longer, data-carrying strands. This means instead of adding nucleotides one at a time, the motifs can be combined in various combinations to represent data. By utilising a library of pre-synthesised motifs, the assembly process is more efficient and cost-effective as fewer cycles are needed to generate a complete DNA strand. This motif-based approach can also improve reliability as errors can be isolated to individual motifs and more easily corrected without affecting an entire sequence [26].

For sequencing, one effective complementary approach for motif-based synthesis is direct oligonucleotide sequencing. This nanopore-based technology processes short oligos while bypassing many preparatory steps [6]. It can help improve the speed and cost-efficiency of reading stored data.

2.4 Encoding Information into DNA Motifs

Storing digital data in DNA requires careful consideration and sophisticated approaches to ensure reliability. In motif-based DNA storage systems, the idea is to map binary data (0s and 1s) to predefined DNA motifs called payload motifs as they represent the binary data.

For example, consider a simplistic binary encoding such as "00" = A, "01" = C, "10" = G, "11" = T. This scheme would be very error prone as it does not account for DNA sequencing errors or problematic patterns like long runs of the same base [5]. One common way to circumvent around this problem is to use ternary encoding rather than binary so base-3 digits (0,1,2) map to A,C,G in a context-dependent way to ensure no two identical bases repeat too often. This is what was used in the early 2010s where the next chosen nucleotide depended on the previous nucleotide, ensuring runs of identical bases were avoided [27]. By avoiding long stretches of the same letter (e.g. "AAAA"), the scheme reduces sequencing difficulties, as they are known to result in read errors in DNA sequencing [28].

Combinatorial design techniques further enhance encoding strategies by systematically generating DNA motifs that maximise sequence differentiation and error resilience. This involves creating motifs that differ in many positions to reduce confusion during sequencing. Constraints such as maintaining balanced GC content (typically between 40% to 60%) are also integrated into these designs to ensure uniform stability [5].

Within motif-based DNA storage systems, data is encoded into predefined motifs with clearly defined roles: payload motifs carry the encoded binary data, address motifs provide unique identifiers essential for indexing and reassembly, and spacer motifs assist in separating different functional motifs clearly. Unintended interactions between motifs can lead to errors, so spacer motifs help act as buffers to ensure the payload and address motif can retain their

integrity. Figure 4 provides an overview of these structural elements within an oligo sequence, highlighting their roles and interrelationships within the encoding framework.

(a) "The general oligo structure design" showing how the address motif (A_0), spacer segments (S_1, S_2), payload motifs (P_0, P_1) and bridge motifs (B) line up from the 5' (start) to 3' (finish) [6]. (Bridge motifs are found in corresponding sequence as part of the DNA double helix and do not contribute to the decoding process.)

(b) Close-up of spacer motif: S_1 and S_2, with the strand's 5' and 3' ends [6].

Figure 4: Modular breakdown of the "bridged oligonucleotide assembly". (a) gives full layout of different motifs as part of the composition. (b) zooms in to the spacer, highlights its two parts (S_1, S_2).

2.5 Nanopore Sequencing and Signal Generation

Nanopore sequencing detects DNA bases by measuring changes in an ionic current as a single DNA strand passes through a tiny pore. By recording the ionic current over time, it generates a "squiggly" signal trace which can be decoded to determine the DNA sequence [29]. As presented in Figure 5, variations in the signal correspond to specific nucleotides, with different combinations of bases producing unique current amplitude and patterns.

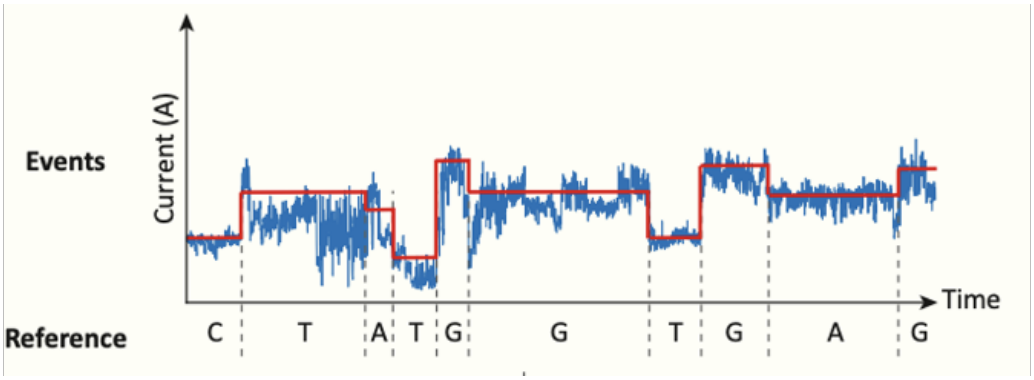


Figure 5: Raw nanopore read illustrating how DNA bases produce distinct current levels and are affected by the bases closest to them. The blue line traces the ionic current over time, the red steps show a simplified segmentation into different events, and the dashed lines link each event to its base in the reference sequence [30].

Oxford Nanopore Technologies' (ONT) sequences showcased in Figure 6, such as MinION, GridION, and PromethION are relevant for DNA data storage decoding due to their portability and real-time long-read sequencing capabilities [31]. The raw output of an ONT device is an analogue-time series with voltage or current readings at thousands of samples per second which represent the passage of bases through the sequencer.

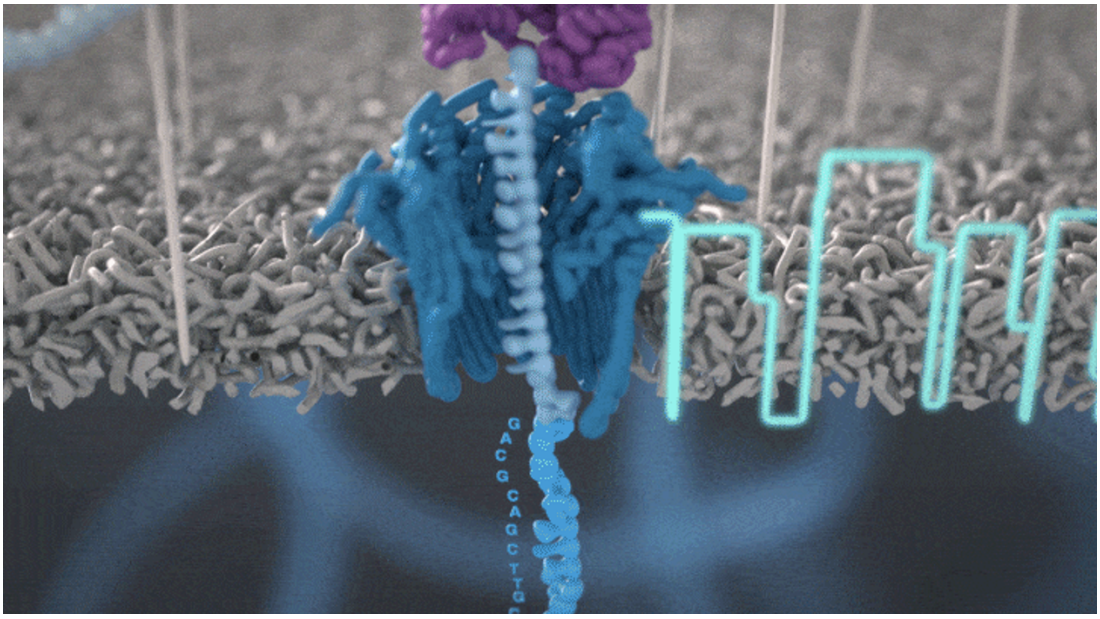


Figure 6: Nanopore sequencer process: After unwinding double helix, motor protein (pink) feeds a single DNA strand through a nanoscale pore (blue). Bases passing through this pore disturb ionic current, producing a unique "squiggle" signal [31].

2.6 FASTA/FAST5 Formats in DNA Storage

When storing and processing DNA-encoded data, bioinformatic file formats are important in representing information at different stages. Two common formats encountered in DNA data storage processes are FASTA and FAST5.

As shown in Listing 1, FASTA is a text-based format for nucleotide sequences (useful for representing composite sequences combining address, spacer and payload motifs). Each sequence is represented by a header line followed by the sequence itself. The header line begins with a ">" character and is often followed by an identifier or description of the sequence.

```
>sequence1
AGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCTAGCT
>sequence2
CGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACGTACG
>sequence3
TTGCTTGCTTGCTTGCTTGCTTGCTTGCTTGCTTGCTTGCTTGCTTGCTTGCTTGCTTGC
```

Listing 1: Example multi FASTA file (comprises multiple FASTA entries): each record starting with ">" then giving sequence ID followed by nucleotide string made up of bases.

Meanwhile, FAST5 is a binary format which holds raw signal data from nanopore sequencing. Each FAST5 file typically contains raw ionic current measurements recorded as a sequence of numbers, along with metadata such as read ID, and timestamps. These files are large and in binary form but by signal plotting the raw ionic current measurements, we see the "squiggly" lines like those in Figure 7, which must be decoded to return to our original binary data.

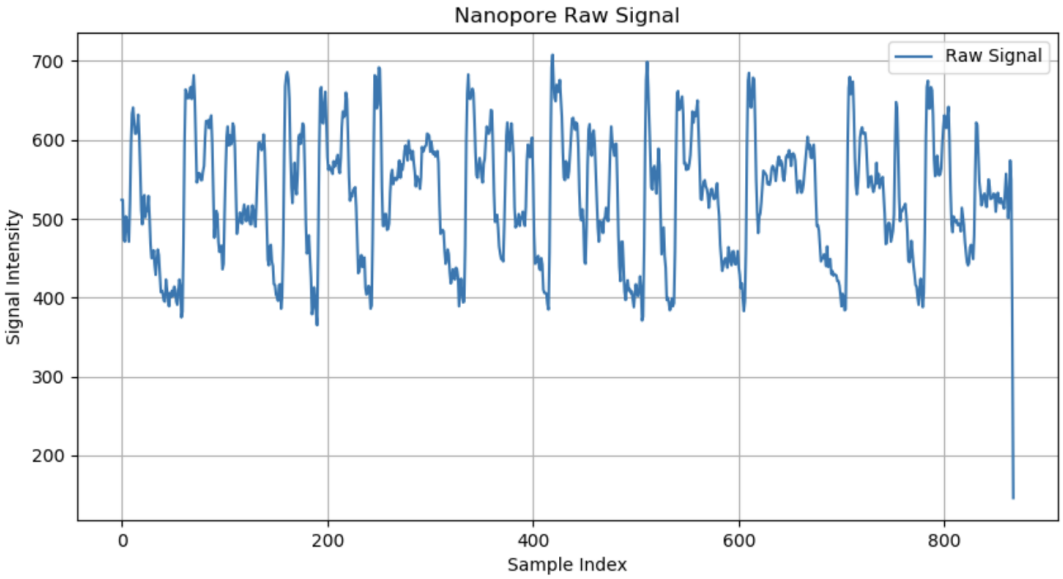


Figure 7: Example of a synthetic nanopore signal trace, where the vertical axis shows ionic current and the horizontal axis is the sample index. The data was pulled directly from a FAST5 file using the Python h5py library and plotted with matplotlib.

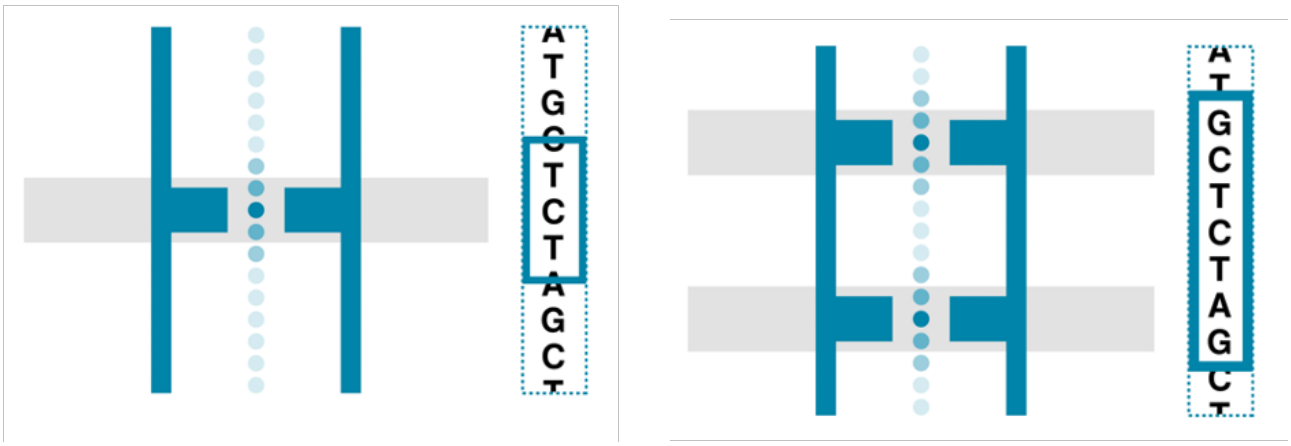
Literature Review and State of the Art

Building on the biological and technological foundations laid out in Chapter 2, this chapter reviews the progression of decoding techniques that transform raw nanopore signals into digital information. We first look at the traditional, two-stage pipelines whose separate basecalling and alignment components enabled the earliest motif-based DNA-storage demonstrations but now impose speed and accuracy limitations. We then map out the transition towards data-driven approaches, showing how convolutional, recurrent and transformer architectures—many borrowed from automatic speech recognition research—have begun to collapse those stages into a single, learnable model. This work prepares the ground for Chapter 4, where we translate those insights into a concrete system specification and experimental plan.

3.1 Traditional DNA Decoding Pipelines

The relationship between the DNA sequence and the electric signal outputted from a nanopore sequencer can be difficult to distinguish as:

- the nanopore sequencer produces an electric signal with noise [30].
- nucleotide bases do not travel through the pore at a constant speed and so can take up a varied amount of time on the amplitude to time or sample index graphs [30].
- measured signal can also be affected by approximately five other closest bases inside the pore, as shown in Figure 8 [30]. This means there could be up to $4^5 = 1024$ possible signal states.



(a) First reader which shows at one time, 3 different bases are contributing to the squiggle.

(b) Second reader which shows at one time, 7 different bases are contributing to the squiggle.

Figure 8: R10 nanopore sequencing technology. "R10 nanopore has two readers spaced along its length" so more bases with a DNA strand can contribute to a squiggle at any one time [32].

This is why traditionally for motif-based DNA storage systems, decoding has been a two stage process of basecalling following by motif identification or sequence alignment.

A simplified example of the traditional decoding pipeline can be found in Appendix A.2.

3.1.1 Basecalling from Raw Nanopore Signals

DNA data storage retrieval begins with basecalling, the process of translating raw nanopore sensor readings into nucleotide sequences. After nanopore sequencers produce a time-series electric current or "squiggle", this information is passed to a basecaller, which resembles an analogue-to-digital conversion of DNA and transforms the ionic current traces into a sequence made up of nucleotide bases A,C,G and T [33], as illustrated in Figure 9.

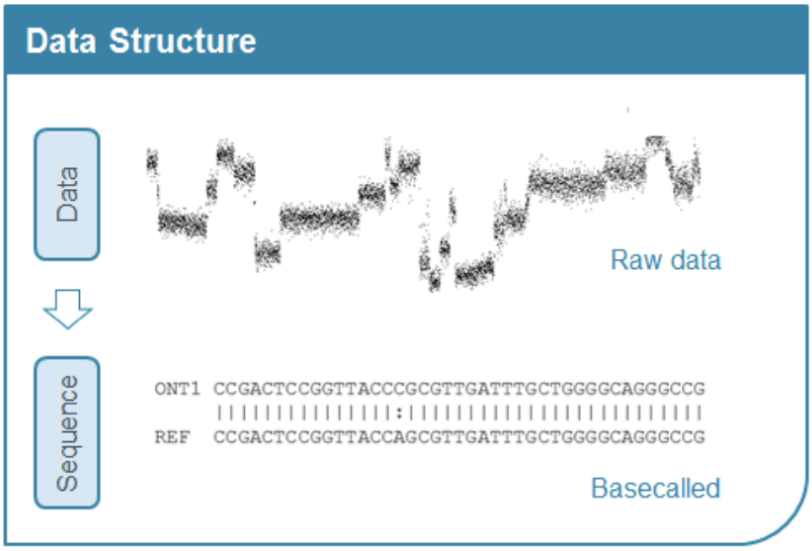
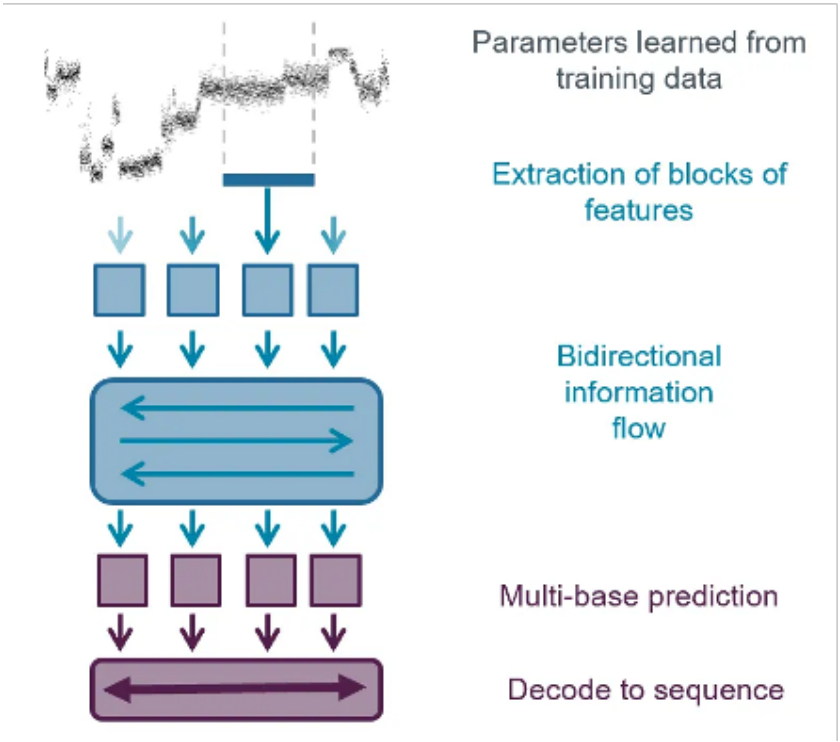


Figure 9: Raw nanopore data being converted by a basecalling algorithm to produce the base sequence of the read [33]. (The original sequence corresponds to REF, while the sequence obtained through basecalling is ONT1.)

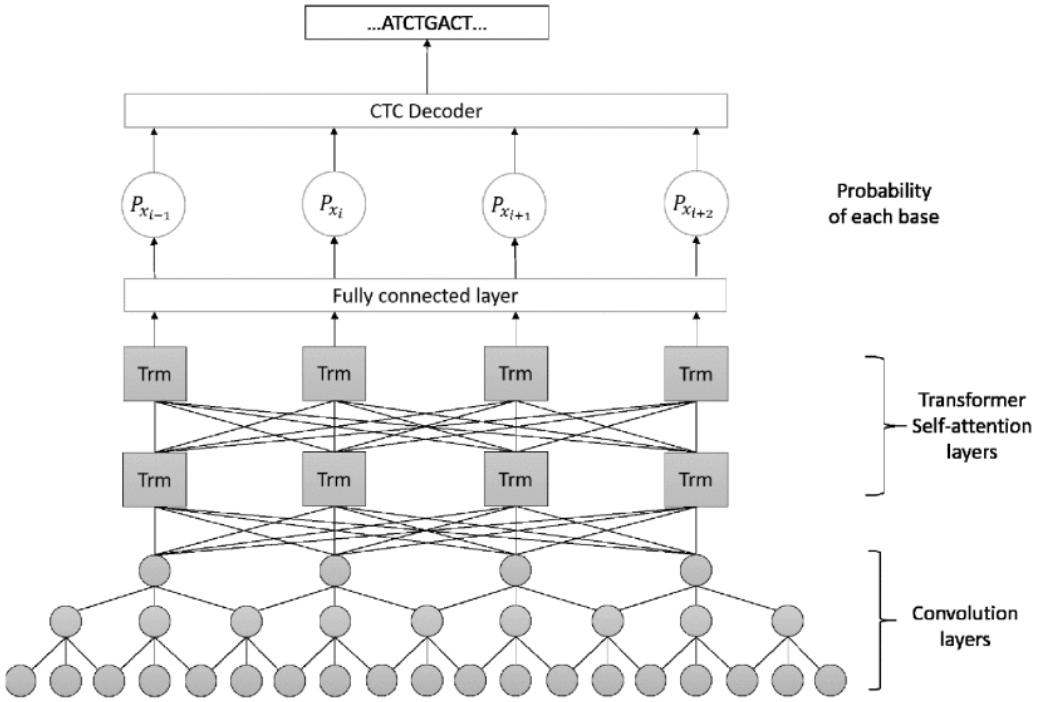
Early nanopore basecallers used statistical methods like Hidden Markov Models (HMM) to identify shifts in current and correlate them to individual bases [34]. These methods relied on segmenting the signal into partitions and then labelling each segment with a nucleotide. As nanopore technology improved, dedicated machine learning models became standard, where modern basecallers leverage deep neural networks to infer sequences directly from the raw signal with higher accuracy [34].

For example, Oxford Nanopore Technologies (ONT) initially released HMM basecallers like Metrichor and Nanocall, but later transitioned to recurrent neural network models for higher base accuracy [35]. Bonito adopts a Convolutional Neural Network + Long Short-Term Memory + Conditional Random Field (CNN-LSTM-CRF) architecture updated from Guppy which used the same architecture but substituted the CRF decoder with the Connectionist Temporal Classification (CTC) decoder to perform end-to-end basecalling from raw nanopore signal [34]. An example of a Recurrent Neural Network (RNN) unit and CTC decoder model is shown in Figure 10a

Meanwhile, another basecaller, presented in Figure 10b, achieves a lower error rate compared to some other ONT basecallers like Guppy and Albacore on a test dataset, but chooses transformer self-attention layers as opposed to recurrent alternatives [36].



(a) RNN-based Nanopore basecaller: the raw current trace is first fed through convolutional layers to pick out short-range signal features, then processed by bidirectional recurrent units that add context, after which probabilities are generated and finally decoded into a nucleotide sequence using a CTC decoder [32].



(b) Transformer-based basecaller: initial convolutional layers downsample the squiggle and extract local patterns, followed by stacked self-attention blocks that learn relationships across the entire signal, and ending with a CTC decoder to translate the learned representations into the final DNA sequence [36].

Figure 10: Two deep-learning pipelines for Nanopore basecalling. (a) CNN + Bidirectional RNN + CTC. (b) CNN + Transformer + CTC.

Modern basecallers are an established form of translation between raw nanopore signals to nucleotide bases so achieve a raw read accuracy on DNA at 96.3 - 98.8% [37]. The accuracy is important as this step is fundamental for all downstream decoding and analysis which depend on the quality of the called sequence.

3.1.2 Sequence Alignment and Motif Identification

Following basecalling, sequence alignment compares the decoded nucleotide sequences to reference sequences to identify payload motifs or locate data regions of interest. This is done by aligning the noisy reads back to either:

- a reference catalogue which consists of every known motif or,
- a payload motif dictionary with additional knowledge of the spacer motif.

As errors in nanopore sequencers and basecallers are dominated by insertions or deletions instead of simple substitutions, the aligner must adjust to frequent frame-shifts while distinguishing payload motifs that differ by only a few bases [33].

Early DNA storage demonstrations used Smith-Waterman dynamic programming algorithm at the cost of quadratic time and memory [38]. To accelerate larger scale experiments, minimap2 can be used which works on long-read alignments two orders of magnitude faster in run time than alternatives while maintaining high sensitivity on error-prone nucleotide sequences [39].

3.1.3 Limitations of Basecalling-First Architectures

While the basecalling-first pipeline is the established approach, it has notable limitations in the context of DNA data storage. First, it can be inefficient and complex, since it separates signal decoding into different stages (signal $\rightarrow$ bases, then bases $\rightarrow$ payload motifs). Each stage (neural basecalling, then motif search or alignment) adds latency and potential points of failure. Additionally, this adds to problem that information stored in DNA can only be extracted slowly, where it may "take hours to days depending on the amount of data" [40]. Basecallers are typically optimised for genomic readouts (identifying all individual bases accurately), which may be overkill for data storage where the end goal is recovering higher-level motifs. In motif-based storage especially, predicting every single base is not strictly necessary if one can determine which motif was present and strictly focus on payload motifs to further streamline the process. A basecalling-first design thus performs redundant work by outputting specific base sequences only to subsequently condense them into motifs. Any base errors or misalignments could lead to misidentification of the payload motifs.

All these limitations motivate rethinking the pipeline: instead of treating basecalling as an obligatory first step, this report looks into designing a decoder that directly maps signal to payload motifs, thereby simplifying the pipeline and enhancing both speed and accuracy of DNA data retrieval.

3.2 Key Machine-Learning Architectures

Introducing machine learning has revolutionised how raw nanopore signals are converted to sequences. The fundamental challenge is a signal-to-sequence mapping: translating a long, analogue time series into a shorter, select sequence of symbols. Two principal modelling paradigms have emerged for this task. Early nanopore decoders followed a segmentation-based method, where the time-series was first split into specific segments (each ideally corresponding to a single nucleotide or motif) and then a classifier assigned a label to each segment [41]. Another modelling paradigm is the encoder–decoder with attention, where an encoder network ingests the entire signal and a decoder network generates the sequence, guided by an attention mechanism to focus on different parts of the signal at each output step [34].

3.2.1 Use of CNNs in Temporal Signal Extraction

CNNs played an important role in nanopore signal analysis as front-end feature extractors. CNNs are extremely useful in detecting local patterns in data and reducing noise which is important for raw nanopore reads which are high frequency and noisy. In practice, state-of-the-art basecallers often begin with one or more convolutional layers to transform the raw current signal into a more compact sequence of feature vectors [42].

3.2.2 Recurrent Units for Sequence Generation

While CNNs handle local feature extraction, RNNs have traditionally handled the global sequence modelling in nanopore decoding. RNNs (particularly LSTMs as visualised in Figure 11 and Gated Recurrent Units (GRUs)) are effective for sequential data because they maintain an internal state that can carry forward information from earlier parts of the sequence to later parts [42].

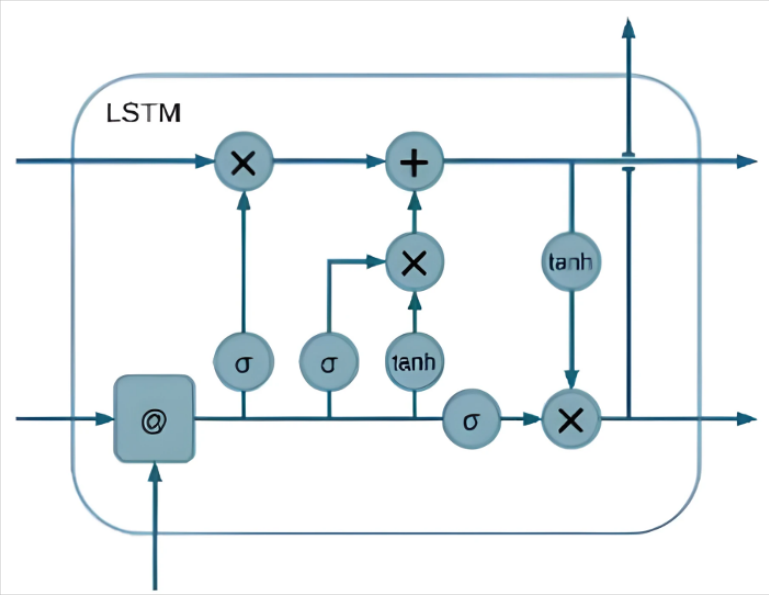


Figure 11: Diagram of internal LSTM operations [42]. The cell state is updated with a mix of old and new information (+), and the output gate controls how much of this state becomes the next hidden state, allowing the network to learn which signals to remember or forget over long sequences.

3.2.3 Transformer-Based Models in Time Series and Sequence Learning

Transformers with a block presented in Figure 12, unlike RNNs, use positional encodings added to their input embeddings to represent sequence position information [43]. These encodings can be fixed or learned vectors, or even relative position representations, but the core idea is the same: they give the model a sense of "this is the 5th element, this is the 6th element," etc., so that order-dependent patterns can be learned. Once positions are encoded, the transformer uses multi-head self-attention to allow every element of the sequence to interact with every other element [42].

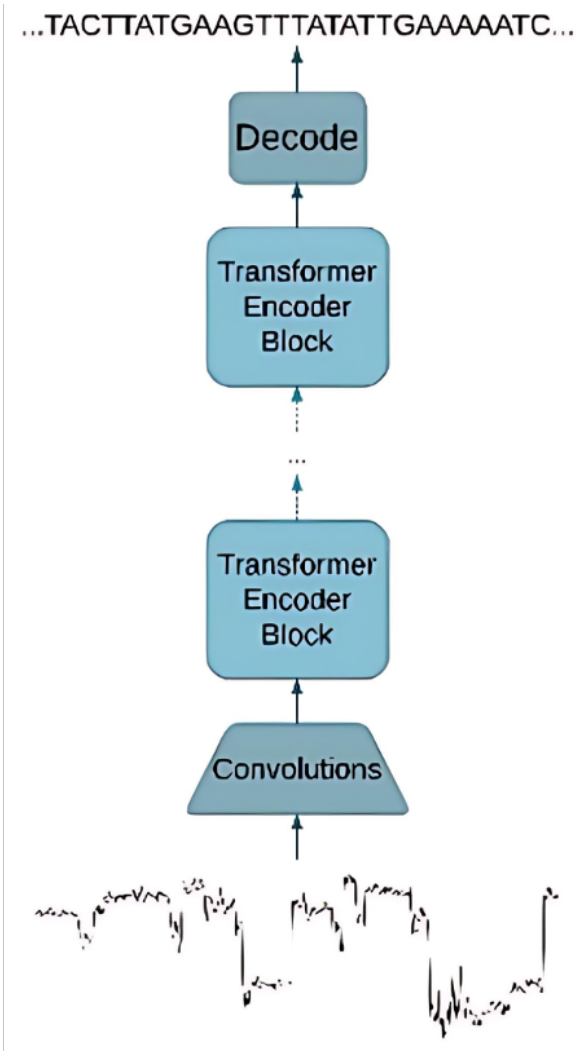


Figure 12: Transformer-based basecaller architecture: the raw nanopore signal is first down-sampled and filtered by convolutional layers, which then pass through a stack of transformer encoder blocks to later be decoded into a DNA string composed of bases [42].

A major advantage is that self-attention is parallelisable, so a transformer encoder can process all input positions simultaneously, which speeds up training on long sequences and enables scaling to very deep networks [36].

3.2.4 CTC and CRF Decoders

The CTC decoder integrates over all possible alignments of signal to output, allowing the network to learn the timing of each base without exact event segmentation [34]. This approach

was created to avoid manual signal segmentation and quickly became standard in community-developed basecallers like Chiron, which used a convolutional recurrent neural network with a CTC loss to achieve accuracy on par with earlier HMM-based methods [34]. CTC's ability to produce shorter output sequences than inputs by using blank symbols is important for basecalling, since a nanopore signal of thousands of samples must collapse into a few hundred bases.

CRF decoders build upon CTC by modelling the dependency between consecutive output symbols. Instead of predicting each base independently, the network outputs the likelihood of state transitions between bases [44]. This label-free CRF scheme ensures that if the signal stays on the same nucleotide for multiple time-steps, the decoder accounts for it, which significantly improves handling of repeat bases [44]. Observably, most of the best performing models used the CRF decoder where there would be on average a 4% increase in raw read accuracy compared to CTC models [34].

3.2.5 Limitations and Methodological Constraints

DNA storage data might contain patterns or codes that span long regions, and capturing such long-range dependencies is challenging. CNN-based models have a limited receptive field, they may perform well at local feature extraction but miss relationships between distant parts of a sequence unless many layers are stacked.

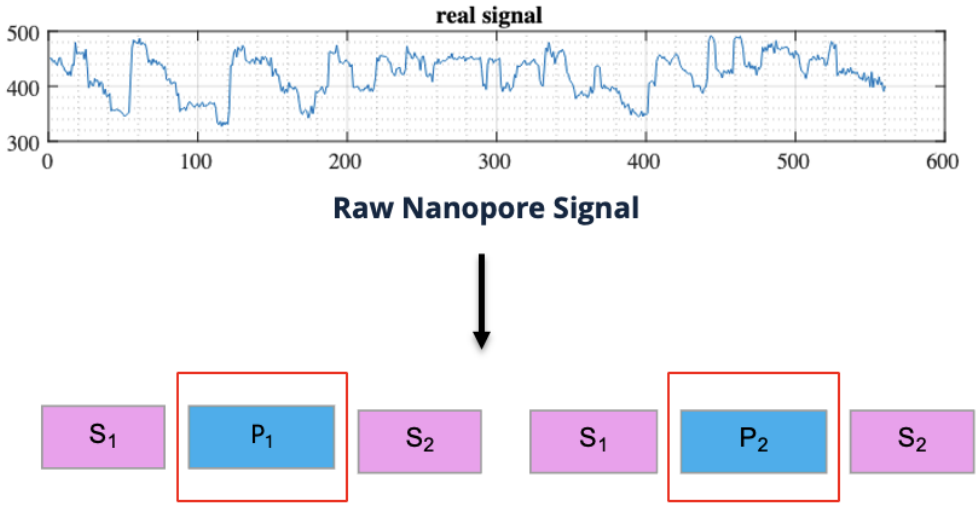
Meanwhile, recurrent units bring strengths in modelling long-range context, but they also come with drawbacks: they process input sequentially, which can be slow, and they compress all history into a fixed-size hidden state at each step [45]. This can make capturing very long-term dependencies challenging for RNNs. Despite this, well-tuned RNNs can prove to be extremely effective.

A vanilla transformer has quadratic time and space complexity in sequence length due to all-to-all attention [42]. So without a careful implementation, it can be detrimental for a real-time decoding problem. However with adaptation, Transformer basecallers have kept computation linear with respect to length [42].

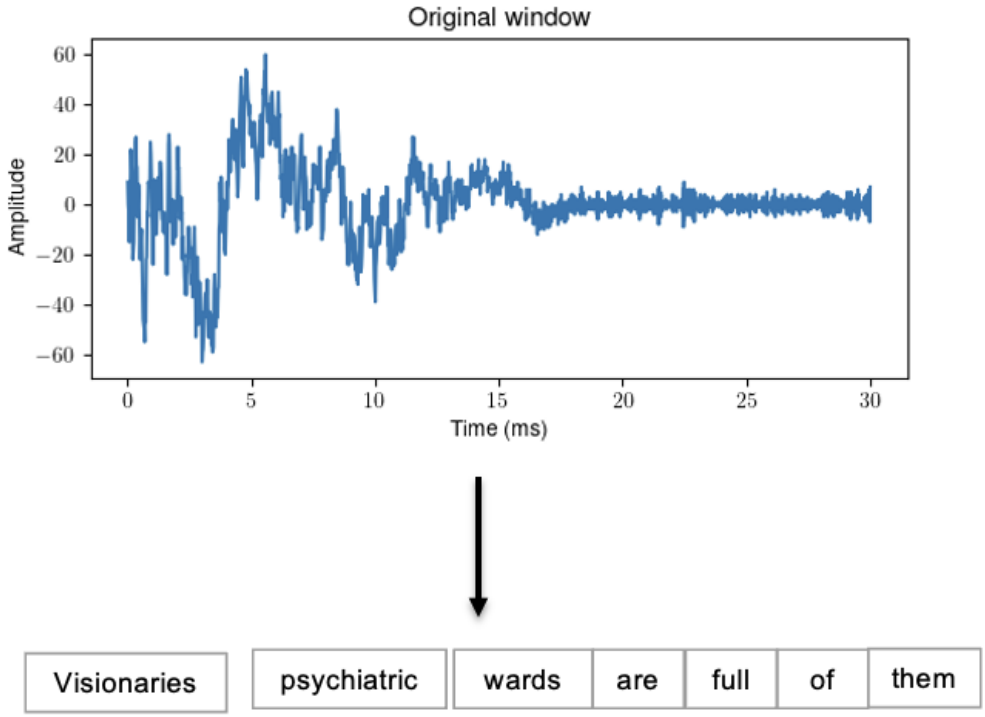
CTC, by design, assumes conditional independence between output symbols given the model's hidden representation [34]. This means the network might treat each base or motif decision in isolation, which may lead to inconsistent outputs. Meanwhile, a flip-flop CRF seen used in ONT's Bonito, doubles the state space per base, so a higher-order CRF could explode in complexity [44].

3.3 Automatic Speech Recognition: Source of Inspiration

Automatic Speech Recognition (ASR) is a problem which entails converting audio waveforms to words. As demonstrated in Figure 13, the problem this report is hoping to explore and ASR are similar in the notion that we start with a noisy signal and want to convert it into groupings of smaller pieces, such as motifs or words.



(a) Decoding the raw nanopore current trace directly into the two payload motifs, P_1 and P_2 (red boxes), with the repeating spacer pattern ($S_1 + S_2$) serving as simple separators to mark payload boundaries [46].



(b) Audio waveform converted directly to the correlated words "Visionaries. psychiatric wards are full of them" [47].

Figure 13: Similarities between DNA storage decoding and ASR. Both parts show a raw analogue signal segmented and mapped into a sequence of units (payload motifs or words).

3.3.1 Architectural Patterns in ASR Systems

Looking through the evolution of automatic speech recognition offers valuable insight, and we can see that this follows the past and current blueprints for DNA decoding systems.

Traditional ASR followed a hybrid architecture: an acoustic model (often a Gaussian mixture model or DNN tied to Hidden Markov Model states) would convert audio features into

phoneme likelihoods, and a separate language model would then guide the assembly of phonemes into words [48].

In recent years, ASR has undergone a paradigm shift to end-to-end architectures. End-to-end ASR models, such as those based on CTC or sequence-to-sequence attention, collapse the pipeline by learning a direct mapping from audio waveform to text transcripts [48]. RNN-Transducer (RNN-T) and Transformer Transducer models are prevalent in voice assistants, combining an encoder (often CNN + LSTM or Transformer) with a prediction network and a joint network to produce text in a single trainable model [48]. Another pattern is leveraging convolutional front-ends (to handle raw waveforms), followed by sequential models (RNN or Transformer) for the linguistic content. We see a similar pattern followed as tools from ASR systems are repurposed to work for nanopore sequencing problems with DNA.

3.3.2 Reducing and Adapting ASR Models for DNA Storage

Despite the strong inspiration from ASR, architectures from this sector must be adapted to work in the DNA storage context, which is a simpler "language" domain. In human speech, models must handle tens of thousands of possible words, complex grammar, and varying accents or pronunciations. Accordingly, ASR models have grown extremely large and complex, often incorporating external language models to bias predictions toward valid word sequences. In DNA storage decoding, there is a closed vocabulary (the motif set) and a known pattern structure as shown in Figure 4a. Therefore certain components from ASR systems can be simplified. For example, *Motifcaller* does not need a separate language model to ensure grammatical correctness of output. The result is the creation of a proof-of-concept, inspired by state-of-the-art basecalling methods, which themselves draw inspiration from ASR systems, to optimise the DNA storage decoding pipeline [35].

3.4 Gaps in the Literature

Although machine-learning basecallers and sequence alignment have made motif-based DNA-storage decoding possible, the literature still exhibits three main overarching blind spots that our work seeks to address. The subsections below unpack each of these gaps and explain how *Motifcaller* targets them.

3.4.1 Lack of End-to-End Motif Decoding from Signals

Motif-based DNA storage is a relatively new method seen to tackle the high costs of DNA sequencing and synthesis, and therefore, the tools to read these motifs directly are still emerging. To our knowledge, *Motifcaller* is the first attempt at an end-to-end motif signal decoder, filling a gap not addressed by even the latest nanopore sequencing tools. While early reports have hinted at similar ideas, none have yet provided a fully reproducible and openly accessible implementation [49]. The need for such an approach has been hinted at by researchers who note the inefficiency of current pipelines for motif-based storage.

This gap in literature is precisely what motivates the *Motifcaller* project: it serves as a proof-of-concept that one can train an ML model to map signal → payload motif sequence

without the intermediate base sequence. By doing so, we target a new point in the design space that had not been explored.

3.4.2 Opportunity for Proof-of-Concept Using Synthetic Data

One practical hurdle in developing new DNA storage decoding methods is the scarcity of large, high-quality experimental datasets. Synthesising and sequencing DNA storage strands is expensive and time-consuming. This presents a gap that can be addressed by simulation. For instance, tools like Squigulator and DeepSimulator can generate realistic nanopore signal traces from arbitrary input sequences [50]. The idea is to test architectures and methodologies on synthetic data and the best of these methods can then be applied to real nanopore signal data.

3.4.3 Component-Level Design Justification

Another gap in the literature is the relative lack of component-level analysis and justification in end-to-end sequencing model design. Many papers introducing a new basecaller or decoder focus on overall accuracy gains, but offer limited insight into why specific architectural choices were made and offer comparisons [49]. For instance, if a model uses a CNN+Transformer, is it clear how much the CNN contributed versus the Transformer? Or if a CRF decoder is used instead of CTC, what is the accuracy vs inference time tradeoff? These questions are often underexplored in published works.

In *Motifcaller*, we intentionally break down the model into components and provide reasoning for each, informed by analogies to ASR and known properties of nanopore signals. In doing so, our work doubles as a design guide for future researchers: each component is justified either by literature precedent or by specific needs of the motif decoding task.

Project Specification and Setup

Building on the literature review and technical survey of Chapter 3, this chapter defines the concrete blueprint for *Motifcaller*. We begin by establishing precise performance and usability targets, then outline our advisory engagement with Helixworks Ltd. Following this, we discusses the software, hardware, and simulation frameworks chosen to achieve those objectives, as well as the ethical, legal, and professional factors involved in this open, reproducible development process. In doing so, we set the stage for Chapter 5, where the focus shifts from "what" and "with what" to "when" and "how".

4.1 Project Objectives

The overarching aim is to design and evaluate *Motifcaller*, a proof-of-concept decoder that maps raw nanopore signals directly to payload motif strings. Success is measured against three specific targets:

1. **Sequence-level accuracy $> 90\,\%$**. A prediction is correct if the sequence of payload motifs predicted by *Motifcaller* aligns with the ground truth sequence of payload motifs. Similar to getting a whole sentence correct in ASR.

2. **Motif-level accuracy $> 95\,\%$**. A prediction is correct if a single payload motif predicted by *Motifcaller* matches the ground truth payload motif and is located at the same index within the sequence. Akin to getting one word out of a sentence correct in ASR.

3. **Usability**. Provide a command-line interface that accepts a single `.fast5` file and returns the ordered payload motif predictions.

A secondary objective is comparative: to benchmark various architectural choices, analyse the rationale behind the use of specific components in this domain, and identify the most effective architectural compositions.

4.2 External Contacts

Early scoping discussions were held with Helixworks Ltd., an Irish DNA-storage start-up. The original plan hinged on a data-sharing agreement that would grant access to proprietary nanopore reads. Contract negotiations between the university and Helixworks were delayed extensively due to factors outside my control resulting in the agreement failing to be finalised in a sufficient enough timeframe to work on as part of the final report. Consequently, the collaboration shifted to an advisory footing where Nimesh Pinnamaneni (CEO, Helixworks) provided informal guidance on the background of motif-based DNA storage and recommended literature.

4.3 Software and Tools Used

All development and experimentation were carried out in Python 3.12.5, drawing on the standard scientific stack: NumPy and Pandas for data handling, Matplotlib for quick visualisation, PyTorch for model definition and training, and Optuna for hyper-parameter search. Python was chosen for it extensive ecosystem of scientific and machine learning libraries. Data-processing scripts rely on Python's built-ins (os, glob, csv, re, random), h5py for .fast5 parsing, and Biopython's SeqUtils.gc_fraction for sequence metrics. Squigulator was used to generate realistic synthetic nanopore squiggles from FASTA inputs [46].

Training workloads ran on the University of Warwick batch compute service:

- **"gecko" GPU node** - dual AMD EPYC 7443, 512 GB RAM, three NVIDIA A10 (24 GB) GPUs; used for the longest runs like Optuna hyperparameter tuning.
- **"eagle" GPU node** – Intel i5 host with an RTX 2080 Ti (11 GB); ideal for fast turnaround debugging and smaller tests.
- **"cpu-batch" node** – dual Xeon E5-2660 v3, 64 GB RAM; sufficient for data preparation, baseline alignment pipelines, and unit tests when no GPU was needed.

Aside from running code, the batch system is also useful to determine the training time (or how long our code has been running).

Productivity and collaboration were managed with Git for version control and a private GitHub repository that hosts the code, experiment logs and issue tracker. Meanwhile, Microsoft OneNote recorded sprint goals, meeting notes and milestone check-lists.

4.4 Legal, Social, Ethical and Professional Issues

4.4.1 Legal Issues

Although the project does not handle personal or clinical samples, several legal frameworks remain relevant:

- **Data protection.** Because all signals are synthetically generated by Squigulator from in-house DNA designs, the UK GDPR and EU Regulation on human data are not relevant.
- **Research reproducibility.** In line with the UK Research Integrity Office's Code of Practice for Research, all synthetic datasets, model checkpoints and analysis scripts will be published to ensure transparency and peer reproducibility [51].
- **Software licensing.** Every third-party dependency, including PyTorch, Squigulator and h5py, is open-source, with licences that allow academic use, modification and redistribution.
- **Contractual constraints.** Initial negotiations with Helixworks did not result in a data-sharing agreement in time, so no proprietary or confidential information is incorporated. As a result, no non-disclosure or data-use agreements apply to the outputs of this report.

4.4.2 Social and Ethical Issues

Ensuring responsible and ethical research practice extends beyond legal compliance:

- **Human and animal welfare.** No experiments involve human participants or animals, so no ethical review board approval was required.
- **Open science and inclusivity.** Outputs are fully documented and saved. This commitment supports the notion of reproducibility.
- **Responsible AI use.** Generative AI tools (ChatGPT, GitHub Copilot) have been used to brainstorm ideas, suggest Python libraries and assist with repetitive code patterns. Additionally, ChatGPT helped rewrite text in certain areas for conciseness by proposing synonyms to improve clarity. Every AI-generated suggestion was critically evaluated, fact-checked and, where necessary, refined to ensure accuracy and clarity.

4.4.3 Professional Issues

Adherence to academic and professional standards underpins the credibility of this work:

- **Attribution and acknowledgements.** Helixworks' advisory support is recorded in the Acknowledgements. No uncredited contributions from external parties are used.
- **Professional collaboration.** Regular fortnightly meetings with my supervisor have been complemented by informal conversations with Nimesh Pinnamaneni (Helixworks), ensuring that domain expertise informs methodological choices.
- **BCS Code of Conduct.** While the British Computer Society's professional code is not a formal requirement for a BSc Discrete Mathematics student, its principles have been voluntarily adopted to guide development and report writing.
- **Generative AI Policy compliance.** All use of ChatGPT and GitHub Copilot follows the Department of Computer Science's Generative AI Policy. This use aligns with Warwick's guidelines on proofreading and AI, and no AI-generated content has been presented as original without critical review.

Project Management

Building on the system specification and development environment established in Chapter 4, this chapter turns to the organisational side of *Motifcaller*. It begins by tracing the timeline and major milestones that structured work before and after the Helixworks collaboration challenges. It then analyses the key risks and describes the mitigation strategies employed to sustain progress. Finally, it outlines the agile practices and toolchain that enabled rapid iteration without excessive overhead. This prepares discussion into the actual codebase and components explored, first beginning by looking into the synthetic data generation in Chapter 6.

5.1 Timeline and Milestones

Two distinct schedules were followed:

Phase I – Original plan (weeks 0–10)

1. Literature survey and appropriate tool installations.
2. Await Helixworks data-sharing agreement.
3. Prototype data-loader for `.fast5` files.

See Gantt Chart A6 in Appendix A.3.

Phase II – Synthetic-data pivot (weeks 11–30)

1. Generate 100,000 synthetic reads with Squigulator.
2. Implement default working pipeline (CNN-Transformer-GRU with Attention) (milestone M1).
3. Design and train alternatives for CNN (milestone M2).
4. Design and train alternatives for Transformer model (milestone M3).
5. Design and train alternatives for GRU with attention model (milestone M4).
6. Integrate Command Line Interface (CLI), evaluation harness and documentation (milestone M5).
7. Draft dissertation chapters; internal hand-in (milestone M6).

Detailed updated Gantt Chart A7 in Appendix A.3.

5.2 Risk Management and Mitigation

A successful delivery of *Motifcaller* depends on four inter-related factors: (1) access to suitable data, (2) adequate compute, (3) stable model training, and (4) a timetable that tolerates setbacks. Each factor is therefore treated as a distinct risk vector, discussed below together with the measures adopted to keep the project on track.

5.2.1 Data acquisition

The original plan assumed timely access to proprietary Fast5 files from Helixworks as my training data. When contractual negotiations stalled, that dependency became the project's single largest point of failure. In response, I pivoted to fully synthetic reads generated with Squigulator. Initial experiments confirmed that the simulator reproduced raw nanopore signals with a high degree of accuracy and were able to recreate the noise artificially typically found in real nanopore data after passing a nanopore sequencer. So the synthetic data could stand in and serve as substitute to help create a suitable proof-of-concept.

5.2.2 Compute infrastructure

Transformer training on 100,000 and 50,000-read batches cannot be accommodated by a personal Mac computer. Development therefore proceeds in two passes: all code is first exercised on tiny subsets (100–1000 reads) to catch logic errors; only once the loss curves behave sensibly is the job escalated to the University's batch compute.

5.2.3 Version control and backup:

All code, configuration files and job outputs or logs are stored in a private GitHub repository. This ensures that, if local disks fail or files become corrupted, we can roll back to any previous commit and recover lost work instantly.

5.2.4 Model development

End-to-end architectures are prone to either divergence or over-confidence on synthetic data. Every training run therefore monitors validation loss and halts automatically after stagnant epochs and a learning rate schedule was used to encourage smoother convergence.

5.2.5 Schedule and scope

To guard against hardware faults or prolonged debugging, development initially involves conducting small scale tests on the personal computer; if the desktop hardware proves unreliable, the entire workflow can pivot seamlessly to the university computers and university's batch cluster. Critical tasks (data pipeline, CLI integration) have fixed deadlines (such as CLI must be completed prior to the presentation). Missing one of these deadlines would result in a scope review and potential scope reduction. Progress is also reviewed in fortnightly supervisor meetings, which serve as opportune moments to discuss challenges related to Helixworks-related

delays or unexpected technical obstacles which threaten the timeline. These collectively ensure that the project remains aligned with the dissertation schedule even when individual tasks encounter setbacks.

5.3 Tools and Management Strategies

Development occurred in sprints, following an agile development framework which offered the greatest amount of flexibility. At the start of each sprint, I used OneNote to write down the key goals, such as writing out code for a specific machine learning component or working on the CLI to ensure progress.

All code was tracked via Git on GitHub to keep the history organised. Experiments were conducted on my personal computer (and, when needed, on the university GPU cluster), with training metrics logged to job_output.out files and errors on job_output.err files.

This straightforward combination of these tools proved sufficient tracking to coordinate tasks, catch delays early and tackle issues without introducing unnecessary complication.

Synthetic Data and Experimental Design

To validate *Motifcaller*, we generate a fully synthetic dataset that mimics real nanopore squiggles. In the paragraphs that follow, we (i) design payload motifs under biochemical constraints, (ii) map bits to motifs and assemble oligos, (iii) create FASTA and metadata files, and (iv) run Squigulator to produce BLOW5/FAST5 signal traces. Although many motif-based systems use an address motif for random access, here we omit it to simplify our proof-of-concept and focus purely on payload-motif recovery. This synthetic data pipeline underpins the experiments and architectural choices elaborated in Chapter 7, where we present the full methodology rationale for the CNN–Transformer–GRU design, data flows, normalisation strategies, model variants, and evaluation metrics.

6.1 Motif Generation Under Biochemical Constraints

Payload motifs must obey the same GC-content rules that keep real DNA strands stable [40]. We therefore require:

- GC fraction between 40% and 60%.
- No homopolymer runs (repeating nucleotide bases e.g. AAAA) longer than 4 bases.

This is enforced in the FASTA file generation as seen in Listing 2.

```
import random
from Bio.SeqUtils import gc_fraction

CONSTRAINTS = {
  'gc_range': (40, 60),
  'max_homopolymer': 4
}

def generate_valid_motif(length, existing):
   while True:
      m = ''.join(random.choices('ACGT', k=length))
      if (m not in existing
         and CONSTRAINTS['gc_range'][0] <= gc_fraction(m)*100 <=
            CONSTRAINTS['gc_range'][1]
         and not
            any(h*CONSTRAINTS['max_homopolymer'] in m for h in 'ACGT')):
         return m
```

```
existing = set()
payload_motifs = [generate_valid_motif(10, existing) for _ in range(16)]
```

Listing 2: Segment from `simplified_fasta_gen.py`: defines GC-content (40–60%) and maximum homopolymer (4) constraints, and uses `generate_valid_motif()` to generate unique 10-nt (nucleotide long) motifs that satisfy these biochemical requirements.

6.2 Binary Mapping and Sequence Construction

To encode digital data into DNA motifs, we first assign each of our M payload motifs a unique B-bit binary code, where $B = \lceil \log_2 M \rceil$. In this proof-of-concept we chose $M = 8$, hence $B = 3$. Concretely, if our payload motifs are

$$\{\, m_0, m_1, \ldots, m_7 \,\},$$

we map each motif m_i to the 3-bit representation of i in standard binary, zero-padded to length 3:

Payload motif	Binary code
m_0	000
m_1	001
m_2	010
m_3	011
m_4	100
m_5	101
m_6	110
m_7	111

Here, `format(i,'03b')` is used to convert integer `i` to its 3-bit binary string (e.g. `5`→`"101"`).

Oligos are then built by concatenating a series of payload motifs sandwiched between a fixed spacer motif. Although some pipelines place an address motif prior to the payload motifs, we omit it here for simplicity: our goal is solely to measure payload recovery performance, not random access. Composite sequence construction thus becomes:

$$\underbrace{\text{spacer}_1}_{\text{first-spacer}} \,\|\, \underbrace{\text{payload}_1}_{\substack{\text{first payload}\\ \text{motif}}} \,\|\, \underbrace{\text{spacer}_2}_{\text{second-spacer}} \,\|\, \underbrace{\text{spacer}_1}_{\text{first-spacer}} \,\|\, \underbrace{\text{payload}_2}_{\substack{\text{second payload}\\ \text{motif}}} \,\|\, \underbrace{\text{spacer}_2}_{\text{second-spacer}} \,\|\, \underbrace{\text{spacer}_1}_{\text{first-spacer}} \,\|\, \underbrace{\text{payload}_3}_{\substack{\text{third payload}\\ \text{motif}}} \,\|\, \cdots$$

This ensures each payload motif is surrounded by both spacer parts in a consistent fashion, helping downstream signal segmentation and alignment.

For example, if spacer motif = `AGAGTTTCGA` = `AGAGT` + `TTCGA` = spacer motif$_1$ + spacer motif$_2$, and the first three payload motifs are m_0, m_1, m_2, the composite sequence is:

$$\texttt{AGAGT} \,\|\, m_0 \,\|\, \texttt{TTCGA} \,\|\, \texttt{AGAGT} \,\|\, m_1 \,\|\, \texttt{TTCGA} \,\|\, \texttt{AGAGT} \,\|\, m_2 \,\|\, \cdots$$

which concatenates to:

$$\texttt{AGAGT}m_0\texttt{TTCGAAGAGT}m_1\texttt{TTCGAAGAGT}m_2\cdots$$

Each header in the resulting FASTA encodes both the motif identifiers and their binary codes, ensuring traceability between the digital payload and the DNA sequence and to help with training data generation.

6.3 FASTA/Metadata File Generation

As presented in Listing 3, each oligo is represented by two lines in the multi-FASTA file:

1. **Header line**, starting with >, followed by:
   - a unique identifier (e.g. `oligo_5`)
   - the exact list of payload motifs in order (10 motifs total)
   - the corresponding binary codes for those motifs
2. **Sequence line**, which concatenates the `spacer` and each of the 10 payload motifs. Since each motif is 10 nt and the whole spacer is 10 nt for this proof-of-concept, the full composite sequence is

$$\underbrace{10 \times 10}_{\text{payload motif total length}} + \underbrace{5 \times 20}_{\text{spacer motif halves total length}} = 200 \text{ nt.}$$

   The total is greater than or equal to 200 nt, which is minimum required to satisfy Squigulator's length requirement.

```
>oligo_5_motifs=GGGATGTCCT,AGAGCTAATG,...binary=001,111,...
AGAGTGGGATGTCCTTTCGAAGAGTAGAGCTAATG...
>oligo_6_motifs=GGGATGTCCT,CCGTCGTACT,...binary=001,110,...
AGAGTGGGATGTCCTTTCGAAGAGTCCGTCGTACT...
```

Listing 3: Sample multi-FASTA records used for model training and evaluation. Each header names the oligo and lists its 10 random payload motifs alongside their binary codes. The sequence line interleaves these motifs with 20 spacer halves (5 nt each), resulting in a 200 nt strand.

6.4 Squigulator and FAST5 Simulation Pipeline

Accurate simulation of raw nanopore signals is essential for training and evaluating *Motifcaller*. We considered both DeepSimulator and Squigulator, the two only known options. Each model has the base-level noise and variability found after passing through an ONT nanopore sequencer, but ultimately chose Squigulator for three main reasons:

- **Updated pore models:** Squigulator incorporates Oxford Nanopore's latest pore characteristics, including per-base distributions and noise profiles [46].
- **Statistical accuracy:** Benchmarks show that Squigulator's model produces significantly lower false positives and therefore has a higher overall accuracy in creating realistic nanopore signals compared to DeepSimulator [46].
- **Performance and reliability:** Squigulator receives more active maintenance and updates, so there is improved reliability.

Our end-to-end pipeline proceeds as follows:

1. **Generate oligo FASTA:**

```
$ cd squigulator/Easy_FastA_Generation
$ python simplified_fasta_gen.py
```

2. **Split into per-oligo FASTA + Run Squigulator (R9 model) + Convert BLOW5 to FAST5:**

```
$ cd ..
$ ./simplified_process_oligos.sh
```

3. **Extract combined CSV to generate training, evaluation and testing data:**

```
$ python simplified_create_csv.py
```

This workflow creates a paired dataset of raw nanopore signal traces in FAST5 format and fully annotated ground-truth payload motif sequences. By using Squigulator's R9 profile, we ensure our synthetic data closely matches the noise characteristics and resolution of real nanopore experiments.

The final dataset is a CSV table pairing each `oligo_id` with its binary payload, payload motif list, reconstructed nucleotide sequence and the raw nanopore signal trace as demonstrated in Listing 4.

```
oligo_id,binary_codes,motifs,sequence,raw_signal
oligo_1,"010,010,...", "CCTGGAACGT,CCTGGAACGT,...",
   AGAGTCCTGGAACGTTTCGAAGAGTCCTGGAACGT...,
   467;454;460;455;451;460;457;...
oligo_2,"010,011,...", "CCTGGAACGT,ACCGATCATC,...",
   AGAGTCCTGGAACGTTTCGAAGAGTACCGATCATC...,
   472;456;449;462;455;448;453;467;479;...
```

Listing 4: Snippet from `longer_small_simplified_results.csv`: each row links an oligo ID to its binary code list, comma-separated payload motifs, reconstructed nucleotide sequence (with spacer and payload motifs), and the corresponding raw signal samples from Squigulator (typically varying in count despite equal sequence length).

Methodology

To implement and rigorously evaluate *Motifcaller*, this chapter first introduces the core neural pipeline that transforms raw nanopore signal traces into ordered payload motifs. We then articulate the reasoning behind our major design choices, covering feature extraction, sequence modelling and decoding strategies, and describe the data preparation system after synthetic data has been generated which can be found in every experiment. Finally, we detail the training regimen, from loss formulations and optimisation protocols to accuracy metrics and early-stopping criteria. Together, these methodological elements establish the framework for the comparative results and discussion presented in Chapter 8.

7.1 Default Architecture Overview

To demonstrate that end-to-end signal-to-motif decoding is not only feasible but capable of meeting our accuracy and usability objectives, we adopted the CNN–Transformer–GRU pipeline as our initial model. This architecture was selected because it combines three complementary strengths and each module are prominent selections respectively for sequential data [52]. Relevant code can be found in Appendix A.5, while deeper explanations of the architecture structure can be found in Appendix A.6.

7.1.1 Rationale for a CNN–Transformer–GRU Pipeline

The core challenge of *Motifcaller* is to translate a long, noisy current trace into a short sequence of discrete motifs. We adopt a default three–stage encoder–decoder architecture with attention to later compare with different architecture choices:

- **Convolutional front-end.** Two 1D convolution + batch-norm layers serve as local feature extractors, smoothing high-frequency noise and highlighting short-range patterns in the raw signal before any sequence modelling.

  A Rectified Linear Unit (ReLU) activation

  $$\mathrm{ReLU}(u) = \max(0, u)$$

  follows each convolution to introduce sparsity to help with noise suppression and gradient flow. Pooling layers then help to down-sample the sequence length which reduces the computational burden on the downstream self-attention stage.

- **Transformer encoder.** A stack of self-attention layers captures long-range dependencies and permits fully-parallel processing over the reduced time dimension. Positional encodings reintroduce order information lost by convolution and pooling.

- **GRU decoder with attention.** At each output step, a unidirectional GRU combines the previous token embedding with a context vector formed by attention over all encoder

outputs, then emits a distribution over the motif vocabulary.

This composition balances efficiency (convolutions and pooling shrink the sequence before the expensive attention) with accuracy (self-attention can relate any two positions, and the GRU decoder retains a compact, dynamic hidden state).

7.1.2 Data Flow and Component Interaction

Figure 14 summarises the end-to-end data flow:

1. **Input:** raw squiggle (1×T tensor) is normalised and padded (to ensure consistency for matrix calculations as raw signal sample counts can vary).
2. **CNN layers:** two Conv1D → ReLU → BatchNorm → MaxPool blocks reduce length by a specific factor, producing a feature map.
3. **Positional encoding:** sinusoidal embeddings are added to each time-step to encode its original position.
4. **Transformer encoder:** a multi-layer, multi-head stack applies multi-head self-attention and feed-forward networks, resulting in a sequence of latent vectors.
5. **Attention-augmented GRU decoder:** begins with a start-of-sequence token and, at each time step, attends over all encoder outputs to form a context vector that is concatenated with the current token embedding. The GRU's output layer projects to the motif vocabulary and selects the next token.

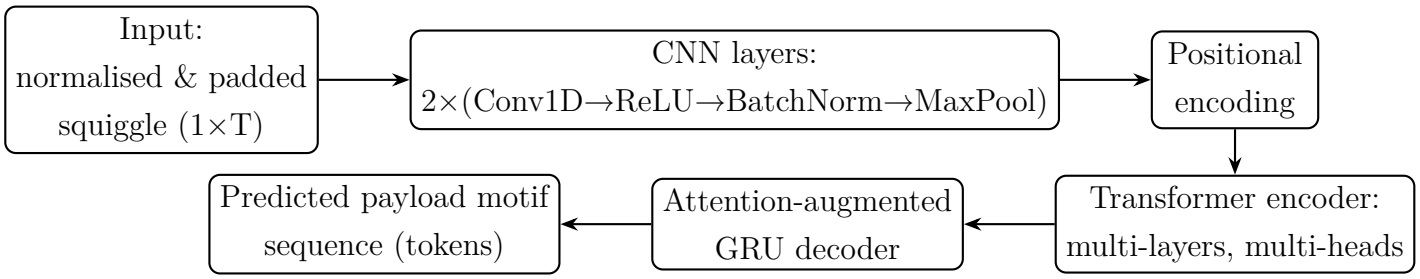


Figure 14: Visualisation of the pipeline defined in `method_1.py`: CNN-Transformer-GRU architecture and data flow. All other hyperparameters not shown here either use default values as presented in Appendix A.4 or are optimised by Optuna fine-tuning.

7.2 Design Decision Analysis

In order to pinpoint which architectural components drive performance in *Motifcaller*, we carried out a systematic comparison of alternative design choices for each major module: signal normalisation, sequence encoding and motif decoding. For each category, a set of variants was implemented as interchangeable classes (see Appendix A.5), and the structural motivations are discussed in Appendix A.6. All variants were trained and evaluated under identical conditions: same synthetic data, training schedule and metrics. So differences in motif-level accuracy, sequence-level accuracy, inference latency and GPU memory usage can be attributed solely to the design change in question.

7.2.1 Signal Normalisation Methods

Raw nanopore currents vary in scale and noise level from read to read. We compared three normalisation schemes (implemented in `normalise_signal`):

- **Z-score:** centring and scaling by the sample standard deviation. Sensitive to outliers.
- **Min–max:** linearly rescales into $[0, 1]$. Guarantees fixed range but can have a high variance if extremes are noisy.
- **Robust (no centring):** divides by the interquartile range (IQR) without subtracting the median, keeping zero as the mean but reducing the influence of extreme values.

```
def normalise_signal(signal, method):
    if method == "zscore":
        std = np.std(signal)
        return (signal - np.mean(signal)) / (std if std>0 else 1.0)
    elif method == "minmax":
        mn, mx = np.min(signal), np.max(signal)
        return (signal - mn) / ((mx - mn) if mx>mn else 1.0)
    elif method == "robust_no_centre":
        q1, q3 = np.percentile(signal, 25), np.percentile(signal, 75)
        iqr = q3 - q1 if q3>q1 else 1.0
        return signal / iqr
    else:
        raise ValueError(f"Unknown method: {method}")
```

Listing 5: Signal normalisation function for all decoding pipelines: includes z-score standardisation, min–max scaling, and an IQR-based robust scaling (without centring)

7.2.2 Comparison of CNN+Transformer Encoder Variants

To determine whether a convolutional front-end is necessary, and to compare different sequence-specific back-ends, we implemented nine encoder variants while holding the GRU–attention decoder constant. Below we summarise each architecture and our rationale for including it in the study.

- **CNN + Transformer (`Encoder_CNNTransformer`)**: See detailed analysis of this default architecture in Section 7.1.
- **CNN + BiLSTM (`Encoder_CNN_BiLSTM`)**: Replaces the Transformer with a bidirectional LSTM. Tests whether gated recurrence (with forward and backward context) can match the Transformer's ability to capture long-range interactions, at the cost of sequential computation.
- **CNN + Vanilla RNN (`Encoder_CNN_RNN`)**: Uses a simple unidirectional Elman RNN in place of gated architectures [53]. Checks the value of gating mechanisms: if performance degrades substantially, this confirms the importance of LSTM/GRU structures in handling noisy, long sequences.

- **CNN + Feed-Forward (`Encoder_CNN_FF`)**: Excludes any sequence model beyond convolution, applying a Multilayer Perception (MLP) independently to each time step. Evaluates whether the CNN's receptive field alone can encode sufficient context for motif decoding, or if explicit sequence modelling is indispensable.

- **CNN + BiGRU (`Encoder_CNN_BiGRU`)**: Swaps the BiLSTM for a bidirectional GRU in the CNN pipeline. This varient explores whether the simpler GRU gating mechanism trains faster or generalises better than LSTM, when combined with convolutional down-sampling.

- **Pure Transformer (`Encoder_Transformer`)**: Drops the CNN entirely and feeds raw, single-channel inputs (projected to the model dimension) into the self-attention stack. This explores whether the CNN down-sampling is beneficial or whether full-length attention can learn directly from the raw signal.

- **Pure BiLSTM (`Encoder_PureBiLSTM`)**: Applies a bidirectional LSTM directly to the raw signal, followed by a linear projection. This tests whether recurrent gating alone, without any convolution or attention, can capture motif-level structure in noisy traces.

- **Pure BiGRU (`Encoder_PureBiGRU`)**: Runs a bidirectional GRU on the raw signal without any CNN front-end. Serves as the simplest recurrent alternative as GRUs have fewer gates than their BiLSTM counterparts, isolating the trade-off between full-sequence gating and learned feature extraction.

- **Raw Input Projection (`Encoder_Raw`)**: The simplest model: projects each sample to the model dimension and applies positional encoding only. This minimal variant gauges the difficulty of decoding without any learned feature extractor.

By comparing these nine variants on identical training and evaluation settings, we cover (a) convolutional versus non-convolutional front-end, (b) the importance of parallel attention versus sequential recurrence versus memoryless feed-forward sequence models, and (c) bidirection versus unidirectional dynamics This comprehensive sweep allows us to identify whether the CNN is needed to denoise and down-sample, and that the final architecture choice is grounded in empirical evidence rather than convenience or prevailing trends.

7.2.3 Comparison of GRU Decoder Variants

After identifying the two strongest encoder blocks (based on a combination of motif/sequence accuracy, inference time and memory footprint), they will then be tested with eleven decoder architectures. We keep two encoder choices as by testing each encoder with the decoder variants, we can uncover whether certain decoders synergise better with a given overall structure. Code for each decoder can be found in Appendix A.5. Below we describe each variant and our motivation for its inclusion.

- **GRU with Attention (`Decoder_GRU`)**: See detailed analysis of this default architecture in Section 7.1

- **LSTM with Attention (`Decoder_LSTM`)**: Replaces the GRU with a two-gate LSTM cell. LSTMs can capture longer dependencies via separate cell and hidden states, at the cost of more parameters. We include it to test whether its extra gating improves motif boundary alignment.
- **Vanilla RNN with Attention (`Decoder_RNN`)**: Uses a simple Elman RNN cell (no gates) combined with the same attention mechanism. By stripping out gating entirely, this variant assesses whether simple recurrence plus attention suffices when paired with a strong encoder.
- **Transformer Decoder (`Decoder_Transformer`)**: An autoregressive Transformer decoder stack that attends both to its own past outputs and to the encoder outputs via cross-attention. This explores a fully parallelisable alternative to RNN-based decoders and tests whether self-attention can replace recurrent state.
- **Attention-Only (`Decoder_AttentionOnly`)**: Here, a learnable query vector attends repeatedly to the encoder outputs to generate each token, without any recurrent or self-attentive recurrence. This tests the extreme of relying on encoder summarisation and attention, eliminating any sequential state in the decoder.
- **Hybrid GRU+LSTM (`Decoder_Hybrid`)**: Runs both a GRU and an LSTM over the same attention-augmented inputs, then averages their outputs before projection. The hybrid potentially combines the fast convergence of GRUs with the long-term memory of LSTMs.
- **Conformer-Style Decoder (`Decoder_Conformer`)**: Integrates convolutional blocks into a Transformer decoder: after each self-attention step, the output passes through depthwise separable convolutions to capture local patterns. Conformers are common in speech for modelling both locality and global context, which we might expect to help with motif boundary precision.
- **GRU without Attention (`Decoder_GRU_NoAttn`)**: A pure GRU decoder that initialises its hidden state from the final encoder vector but does not attend at each step. This variant measures how much benefit the attention mechanism contributes over a simple sequence model.
- **LSTM without Attention (`Decoder_LSTM_NoAttn`)**: Similar to the GRU no-attention variant, using an LSTM cell. It further isolates the impact of recurrent gating alone in the absence of explicit context vectors.
- **CTC Decoder (`Decoder_CTC`)**: Projects each encoder time step directly to motif logits and applies a Connectionist Temporal Classification loss during training. CTC avoids the need for frame-level alignment but assumes conditional independence across outputs. This has been widely used in basecalling, so this will measure its effectiveness for motif-based decoding.

- **CRF Decoder (`Decoder_CRF`)**: Replaces CTC with a Conditional Random Field layer, which models dependencies between adjacent output labels. Recent studies report that CRF decoders vastly outperform CTC in handling homopolymer runs and small insertions/deletions, making it a compelling addition and comparison point for motif-based storage [34].

Each decoder was implemented as a subclass so that it could be swapped interchangeably. By systematically varying only the decoder while measuring accuracy, inference latency and GPU memory, this comparison isolates the relative merits of gating, attention, convolutional augmentation and structured decoding methods (CTC vs. CRF) in recovering payload motifs from noisy nanopore signals.

7.3 Data Pipeline and Preparation

This section describes how raw nanopore signals and associated motif labels are transformed into the tensors consumed by our models. We cover the steps from reading the CSV export of simulated Fast5 files, through preprocessing and tokenisation, to batching and input/output formatting.

7.3.1 Preprocessing and Tokenisation

All data originates from the `longer_large_simplified_results.csv` file produced by our Squigulator pipeline (seen in Listing 4). Each row contains:

- `raw_signal`: a semicolon-separated list of integer current levels,
- `motifs`: a comma-separated list of payload motif strings,
- (plus `binary_codes` and `sequence`, which are unused by the decoder).

Loading and normalisation. In `MotifSeqDataset.__getitem__()`, we parse `raw_signal` into a NumPy array of floats, then apply the normalisation (presented in Listing 5). A single channel is added via `unsqueeze(0)`, resulting in a tensor of shape $(1, T)$.

Vocabulary construction. During dataset initialisation we scan all `motifs` fields, split on commas, and collect the unique motif strings. We add three special tokens:

$$< \texttt{PAD} > = 0, \quad < \texttt{SOS} > = 1, \quad < \texttt{EOS} > = 2,$$

and then assign each motif a consecutive integer index starting at 3. <SOS> (Start of Sequence) marks the beginning of each motif sequence, <EOS> (End of Sequence) signals the end, and <PAD> (Padding) is used to standardise sequence lengths for matrix computations. Since raw signals comprise sequences of numbers that may vary in length (as shown in raw_signal column of Listing 4), padding ensures that all sequences have a fixed length, allowing efficient batch processing in deep learning models. This creates a vocabulary of size V, where V includes

the three special tokens. At runtime, each list of motifs $(m_1, \ldots, m_K)$ is mapped to the index sequence

$$[\,\texttt{SOS},\ \mathrm{idx}(m_1), \ldots, \mathrm{idx}(m_K),\ \texttt{EOS}\,],$$

producing a target tensor of shape $(K + 2)$.

7.3.2 Data Augmentation and Sampling Strategies

Synthetic variability. By design, all reads are simulated via Squigulator (Section 6.4), which injects realistic noise and speed variation comparable to those observed after a strand passes through ONT's nanopore sequencer. No further signal-level augmentation (e.g. additive noise) was applied, as the synthetic traces already exhibit the range of noise expected in real data.

Subsampling for rapid iteration. When debugging or generating normalisation comparison plots, we pass a `sample_size` argument (e.g. $N = 1\,000$) to `MotifSeqDataset`, which truncates the DataFrame to its first N rows. This allows fast debugging without sacrificing reproducibility.

Train/validation/test split. We use a fixed 60/20/20 split on the dataset indices with `sklearn.model_selection.train_test_split`. Each subset is wrapped is loaded with identical normalisation and tokenisation logic to ensure consistency across experiments.

7.3.3 Input/Output Formatting for Training

Batch collation. Our `collate_fn_seq` takes a list of (signal, target) pairs and:

1. Finds the maximum signal length $T_{\max}$ in the batch, pads each $(1, T)$ tensor on the right with zeros to $(1, T_{\max})$, and stacks them into a shape $(B, 1, T_{\max})$ where B is the batch size.
2. Finds the maximum target length $L_{\max}$, pads each target sequence on the right with `PAD` tokens to length $L_{\max}$, and stacks into $(B, L_{\max})$.
3. Returns the tuple (signals_tensor, targets_tensor, lengths), where lengths is a list of original target lengths (saved for masking downstream in the process).

I/O shapes.

- **Inputs:** tensors of shape $(B, 1, T_{\max})$, suitable for a 1D-CNN encoder.
- **Targets:** integer tensors of shape $(B, L_{\max})$, where each row begins with `SOS` and ends (possibly after padding) with `EOS`.

7.4 Training Setup and Evaluation Metrics

This section details how models are trained, which loss functions and optimisers are employed, how we measure accuracy, and our approach to scheduling and hyperparameter search.

7.4.1 Loss Functions and Optimisation

Cross-Entropy Loss. For all sequence-to-sequence decoders (GRU, LSTM, Transformer, attention-only, hybrid, Conformer, GRU/LSTM without attention), we use the standard negative log-likelihood by using `nn.CrossEntropyLoss(ignore_index=PAD_TOKEN)`. The loss per step is

$$\mathcal{L}_{\text{CE}} = -\sum_{b=1}^{B}\sum_{t=1}^{T_b} \log\Big[\text{softmax}(\mathbf{z}_{b,t})\Big]_{y_{b,t}},$$

where

- B is the batch size (number of sequences in one training sequence).
- $b \in \{1, \ldots, B\}$ indexes the example within the sequence.
- T_b is the true (unpadded) length of the target sequence for example b; after adding `SOS` and `EOS` it is the number of non-PAD time-steps.
- V is the vocabulary size (including the three special tokens $\{\texttt{PAD}, \texttt{SOS}, \texttt{EOS}\}$).
- $\mathbf{z}_{b,t} \in \mathbb{R}^V$ denotes the vector of pre-softmax logits produced by the model for example b at decoder time-step t.
- $y_{b,t} \in \{0, \ldots, V-1\}$ is the ground-truth token index at that same position.

which ignores time-steps padded with `PAD`.

Connectionist Temporal Classification (CTC). CTC decoder addresses the lack of information on how many signal frames correspond to each motif and where boundaries lie by introducing a "blank" symbol and defining a many-to-one collapse function which removes blanks and merges the repeated labels. During training, `nn.CTCLoss` marginalises all possible alignments between the encoder's per-time-step logits $\mathbf{z}_{b,t}$ and the ground-truth motif sequence $y_{b,1:L_b}$ and allows a length mismatch.

Conditional Random Field (CRF). CRF layer explicitly models transitions between successive motif labels. We use `torchcrf.CRF` to compute the negative log-likelihood over all valid label sequences under a first-order Markov chain. By learning transition scores alongside emission scores, the CRF decoder enforces consistency by for example, discouraging unlikely motif pairs or detecting instances of repeated bases as a result of sequencing errors and preventing them from badly affecting final decisions.

Optimiser and Regularisation. We use the Adam optimiser because its adaptive moment estimates accelerate convergence on noisy, sparse gradients, which is specifically important in long-sequence, attention-based models where gradient magnitudes can vary dramatically. A learning rate balances rapid initial learning with stability, while a weight can discourage over-confident weight growth and improve generalisation. Finally, clipping all gradients to a maximum global norm can prevent occasional large updates (common in deep recurrent or attention layers) from destabilising training or causing divergence.

Teacher forcing. Additionally, during training, we apply teacher forcing: at each decoder step, the ground-truth previous motif is fed in rather than the model's own prediction. This accelerates convergence by ensuring the hidden state remains close to the true sequence trajectory, preventing error accumulation in early steps. The use of a teacher-forcing ratio then allows a model to shift towards becoming fully autonomous, which can help improve accuracy at a faster rate.

Learning-Rate Scheduling and Early Stopping. We reduce the learning rate by a certain factor whenever the validation loss plateaus for a set number of epochs, using `ReduceLROnPlateau`. Training halts early if no improvement in validation loss occurs for a number of consecutive epochs, preventing overfitting and wasted compute.

7.4.2 Accuracy Metrics

Both token accuracy and sequence accuracy are computed by our `evaluate_model_seq` function, applied on validation and test sets with teacher-forcing turned off.

Token Accuracy. For each predicted sequence $\hat{y}_{b,1:L_b}$ and ground truth $y_{b,1:L_b}$, we align them up to the first `EOS` token and count matching positions:

$$\text{TokenAcc} = \frac{\sum_b \sum_{t=1}^{\min(L_b, \hat{L}_b)} \mathbf{1}[\hat{y}_{b,t} = y_{b,t}]}{\sum_b \max(L_b, \hat{L}_b)}.$$

where:

- B is the batch size (number of sequences in one training sequence).
- $y_{b,1:L_b}$ is the ground-truth motif sequence for example b.
- $\hat{y}_{b,1:\hat{L}_b}$ is the predicted motif sequence for example b.
- L_b is the true sequence length for example b.
- $\hat{L}_b$ is the predicted sequence length for example b.
- $\mathbf{1}[\cdot]$ is the indicator function (1 if condition holds, 0 otherwise).

This reflects the fraction of correctly identified motifs across all positions in a training sequence.

Sequence Accuracy. A prediction is deemed correct only if the entire motif sequence matches exactly (after trimming at `EOS`):

$$\text{SeqAcc} = \frac{1}{B} \sum_{b=1}^{B} \mathbf{1}\left[\hat{y}_{b,1:\hat{L}_b} = y_{b,1:L_b}\right].$$

This stringent metric measures end-to-end decoding precision and is similar to "whole-sentence" accuracy in ASR.

7.4.3 Training Schedules and Hyperparameter Tuning

Default Schedule. Unless otherwise stated, all encoder–decoder combinations are trained for up to 30 epochs with teacher-forcing ratio decaying exponentially from 0.5 to 0.0 according to

$$\alpha_t = \alpha_{\min} + (\alpha_{\max} - \alpha_{\min})\, e^{-t/\tau},$$

where $\alpha_{\max} = 0.5$, $\alpha_{\min} = 0$, $\tau = 50$ and t is the current training epoch (following all other default hyperparameters found in Appendix A.4). This training schedule with predermined hyperparameters is often used when comparing against other architecture choices as its the quickest way to effectively determine which is the best for this DNA storage decoding problem.

Refined Hyperparameter Schedule To assess each architecture's full potential, there is an alternative to the default schedule with a dedicated hyperparameter optimisation phase. Using Optuna, we conduct 20 trials on the validation split, sampling from the following search space:

$$\begin{aligned} d_{\text{model}} &\in \{64, 128, 256\}, \quad n_{\text{head}} \in \{2, \ldots, 8\}, \quad \text{hidden\_size} \in \{128, 256, 512\}, \\ \alpha_{\max} &\in [0.4, 0.8], \qquad \alpha_{\min} \in [0, 0.2], \qquad \text{learning rate} \in [10^{-4}, 10^{-3}], \\ \text{batch size} &\in \{8, 16, 32\}. \end{aligned}$$

These ranges balance model capacity against computational traceability: d_{model} and hidden_size determine the dimensionality of latent representations; n_{head} adjusts the level of self-attention; $\alpha_{\max}$ and $\alpha_{\min}$ tune the speed of transition from teacher-forced to free decoding; while learning rate and batch size control optimisation stability and GPU memory footprint. Each trial runs up to 30 epochs with early stopping on validation loss, minimising wasted cycles on unpromising configurations.

Once the best hyperparameters are identified, we retrain the corresponding model for up to 100 epochs under the same optimised hyperparameters. Early stopping with a patience of 10 epochs ensures training halts as soon as performance converges, preventing overfitting and conserving resources. This two-stage schedule helps models fully converge and provide reliable estimates of each architecture's ceiling performance.

7.5 Command-Line Interface and Usage

To make our trained models immediately accessible for decoding new data, we implemented a streamlined command-line script, `predict_motifs.py`, which encapsulates signal extraction, normalisation, model loading, decoding and output formatting in a single, user-friendly tool. The script can be used by writing the following into the terminal:

```
python predict_motifs.py <fast5_file> <output_csv> [-model_path
MODEL] [-mapping_file MAP]
```

where the first two arguments specify the input FAST5 file and the destination CSV for the predicted motif list. Optional flags allow overriding the default PyTorch checkpoint (`-model_path`)

and the JSON file that maps motif strings to integer indices (`-mapping_file`) which can be preset to the best models.

Internally, the script opens the FAST5 container with `h5py`, locates the "Raw/Signal" dataset and loads the raw nanopore signal trace into memory. It then applies the same normalisation function used during training to ensure consistency between training and inference. Next, the chosen checkpoint is unpacked using `torch.load`, recreating the `Seq2SeqMotifCaller` object along with its associated `Encoder`, `Decoder` and `PositionalEncoding` classes.

Finally, the script translates the predicted indices back into human-readable payload motif strings using the supplied FAST5 file, and writes them into the CSV in two columns: "Position" and "Predicted Motif". This output can be directly inspected or piped into downstream analysis.

Crucially, the CLI dynamically imports any definitions of `Encoder`, `Decoder`, `PositionalEncoding` and `Seq2SeqMotifCaller` present in the execution environment, so new architectures can be deployed without a lot of script modification. Once a variant achieves satisfactory motif and sequence-level accuracy, its checkpoint and corresponding mapping JSON can be passed as arguments, and `predict_motifs.py` will load and run it identically. This design ensures rapid iteration and reliable production use of any model in our encoder–decoder family.

Results and Discussion

Building on the methodological framework laid out in Chapter 7, this chapter presents the empirical evaluation of all major design decisions. We begin by examining how different signal normalisation schemes influence downstream motif-calling accuracy. Next, we explore the behaviour of our default CNN–Transformer–GRU pipeline under varying data volumes and training regimens. With that baseline established, we fix the decoder and compare nine encoder backbones to determine the importance of local feature extraction versus global context modelling. We then hold two of the strongest encoders constant and evaluate eleven decoder variants to design the most effective end-to-end architecture. Finally, we synthesise these results into a recommended final model configuration, discuss its strengths and limitations, and draw broader lessons for nanopore-based DNA storage decoding, setting the stage for the summary, limitations and future work outlined in Chapter 9.

8.1 Effect of Design Choices on Performance

In this section, we systematically quantify how much each major architectural decision (for normalising, encoding and decoding) affects payload motif-calling accuracy and efficiency.

8.1.1 Impact of Normalisation Technique

Figure 15 compares token and sequence accuracy on the test set for the three normalisation methods described in Subsection 7.2.1, using the default hyperparameters and the CNN-Transformer-GRU baseline (Appendix A.4). Two important observations emerge:

- **Token accuracy jump under `robust_no_centre`.** Both `zscore` and `minmax` achieve only around 12–13% token accuracy, whereas `robust_no_centre` exceeds 60%. This five-fold improvement indicates that scaling by the interquartile range without centring preserves the relative amplitudes of the raw nanopore squiggle more reliably than either subtracting the mean (z-score) or compressing to $[0, 1]$ (min–max). In motif-based storage, subtle current fluctuations mark motif boundaries, and overly aggressive centring or range compression can obscure these patterns. This is leading to near-random token predictions under `zscore` and `minmax`.
- **Non-zero sequence accuracy only for `robust_no_centre`.** Exact-match sequence accuracy is essentially zero for both `zscore` and `minmax`, reflecting that even correctly predicted individual motifs never align end-to-end. Under `robust_no_centre`, sequence accuracy rises to almost 3%, showing that the decoder recovers complete motif strings in a small but meaningful fraction of reads when the signal scaling retains informative amplitude structure.

Together, these results demonstrate that `robust_no_centre` is the most effective normalisation method comparatively for downstream motif decoding: it balances resilience to outlier

currents (via IQR scaling) with the preservation of baseline offsets, which appear important for distinguishing neighbouring motifs in our synthetic squiggles. Accordingly, all subsequent experiments use `robust_no_centre` as the standard preprocessing step.

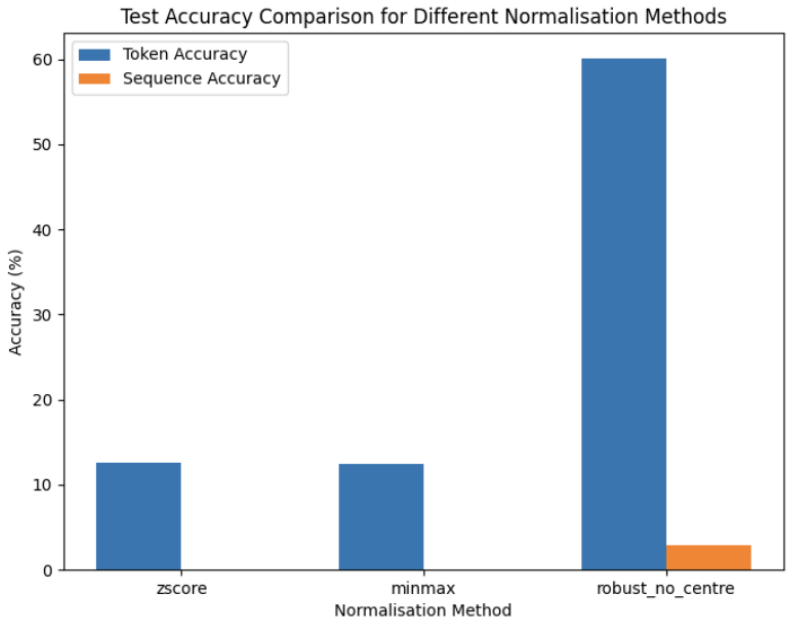


Figure 15: Comparison of token and sequence-level test accuracies for three signal normalisation methods (zscore, minmax, robust_no_centre). Models were trained on 50 000 reads for up to 30 epochs with early stopping, following the default training schedule and default hyperparameter settings (see Appendix A.4).

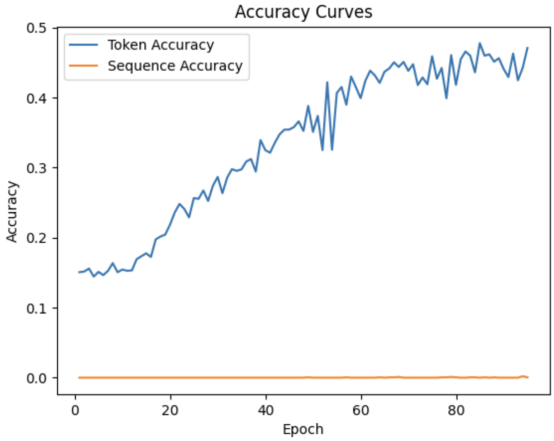
8.1.2 Performance of CNN-Transformer-GRU

We begin by evaluating the default CNN–Transformer–GRU architecture—using `robust_no_centre` normalisation, the hyperparameters listed in Appendix A.4 and our standard training schedule—as we scale from 10 000 to 100 000 training reads. We then switch to the refined hyperparameter schedule (using Optuna based tuning) and repeat the same experiments, allowing us to compare performance with and without optimisation. Figure 16 presents the accuracy and loss curves for 10 000 versus 100 000 reads under the default settings, while Figure 17 presents the corresponding results obtained with Optuna selected hyperparameters.

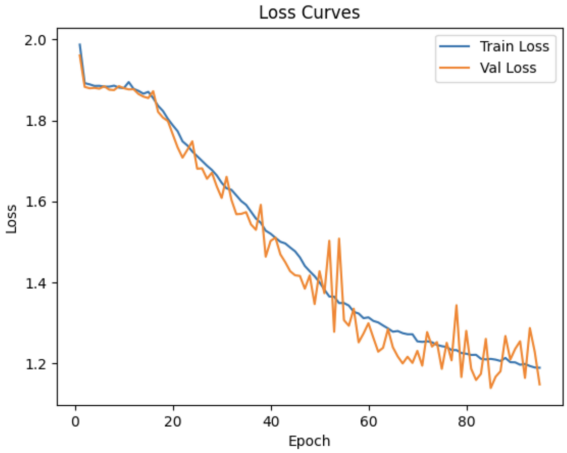
Default hyperparameters (no tuning). With only 10 000 reads (Figure 16a), token accuracy climbs slowly from around 15% up to 45% by epoch 80, while sequence accuracy remains essentially zero throughout—no full motif string is ever predicted exactly. The corresponding losses (Figure 16b) decline steadily from just under 2.0 to about 1.2, but the curves exhibit considerable volatility, with frequent upward spikes. This jitter reflects sensitivity to the fixed learning rate and teacher-forcing schedule: the model frequently over or under corrects on minibatches, leading to the observed oscillations.

Expanding to 100 000 reads (Figure 16c–16d) results in a substantial improvement: token accuracy now reaches about 75% by epoch 80, and sequence accuracy gradually rises to around 9%. The loss curve falls more sharply to approximately 0.8, but still shows ripples throughout

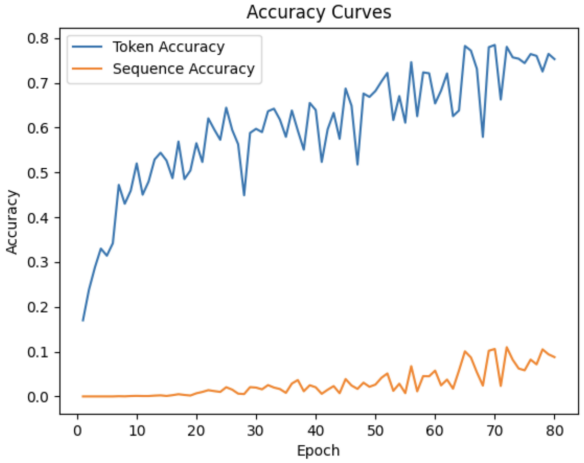
training. In both dataset sizes, the noisiness of these curves indicates that the default hyperparameters are not optimal for stable convergence: although the model learns, it does so in fits and starts. Still, the gap between token and sequence-level performance underscores that, although the model learns to predict individual motifs, assembling them in the correct order remains challenging under the default settings.



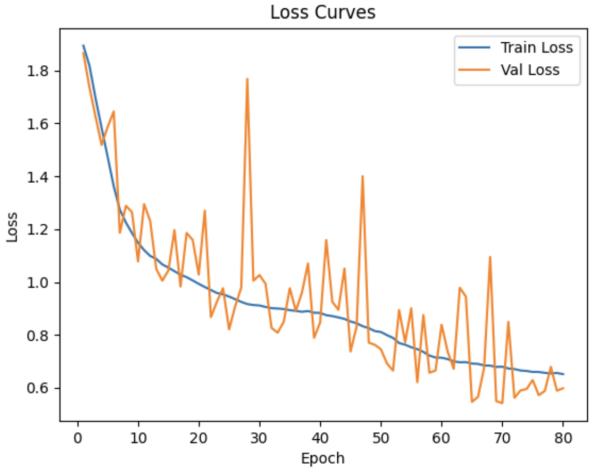
(a) Token and sequence accuracy over epochs when training on 10 000 reads: token accuracy climbs from ~15% to ~48%, while sequence accuracy remains near zero.



(b) Training and validation loss curves over epochs for the 10 000-read experiment, showing a decline from ~1.9 to ~1.2.



(c) Accuracy curves over epochs on 100 000 reads: token accuracy rises above 0.7, and sequence accuracy gradually improves to ~0.1.



(d) Loss curves over epochs for the 100 000-read run, with training and validation loss falling to ~0.6 with high fluctuation.

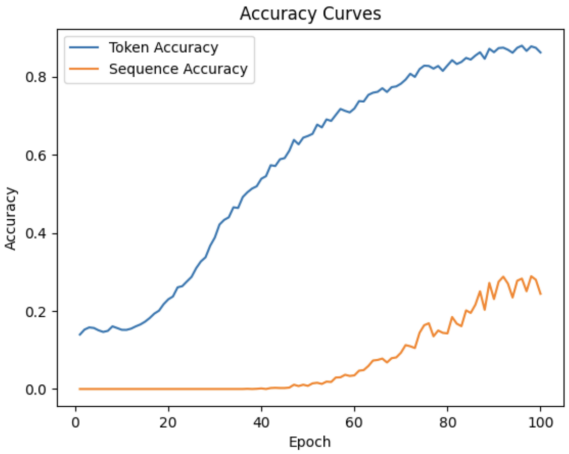
Figure 16: Effect of dataset size on training dynamics of CNN + Transformer + GRU (with attention) with predetermined hyperparameters from Appendix A.4. (a)–(b) correspond to a smaller dataset (10 000 reads), and (c)–(d) to a larger one (100 000 reads). Each run has a max of 100 epochs with early stopping implemented. The larger dataset results significantly higher token accuracy and lower loss.

With Optuna hyperparameter tuning. Introducing Optuna-driven selection (max 100 epochs with early stopping) dramatically accelerates convergence and boosts both metrics—but equally important, it produces much smoother learning curves.. On 10 000 reads (Figure 17a), token accuracy rapidly climbs above 80% and sequence accuracy to over 25%—a four-fold

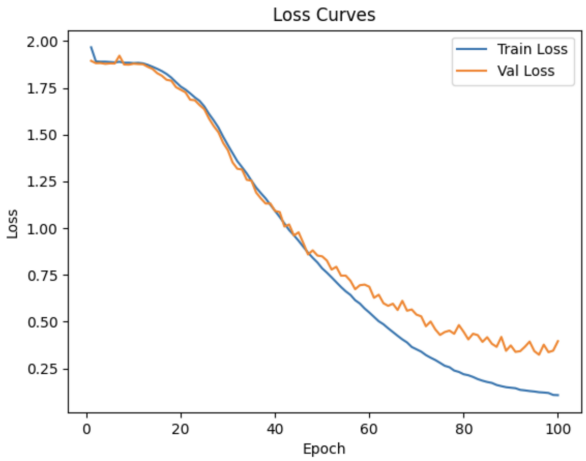
sequence-level gain compared to the untuned run and will small fluctuations. Losses (Figure 17b) fall below 0.4 by epoch 100, evidencing a much tighter fit, but also with minimal oscillation, indicating that the tuned learning rate and decay schedule stabilise minibatch updates.

On 100 000 reads with tuning (Figs. 17c–17d), the model converges within 30–40 epochs: token accuracy saturates near 100%, and sequence accuracy near 99%. Validation loss dips almost to zero, indicating near-perfect learning of the synthetic motif strings.

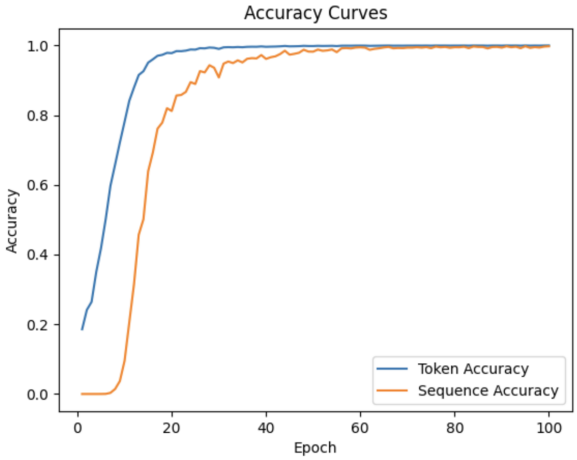
Figure 16 without hyperparameter tuning produces graphs with the jagged, noisier losses and accuracies than those seen in Figure 16. This underscores how crucial hyperparameter optimisation is—not only for final performance but for reliable, stable training.



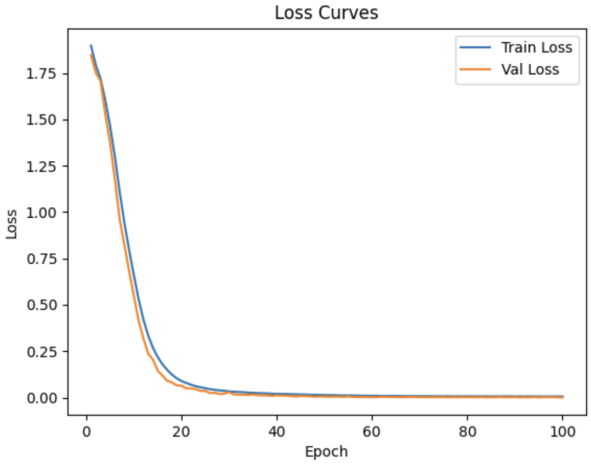
(a) Token and sequence accuracy over epochs when training on 10 000 reads: token accuracy climbs from ~17% to ~85%, while sequence accuracy starts increasing from 0 at around 50 epochs to ~25%.



(b) Loss for 10,000 iterations Training and validation loss curves over epochs for the 10 000-read experiment, showing a decline from ~1.9 where the loss diverges at ~50 epoch.



(c) Accuracy curves over epochs on 100 000 reads: token and sequence accuracy tend towards ~100% by the ~50th epoch.



(d) Loss curves over epochs for the 100 000-read run, with training and validation loss falling to ~0 with little to no fluctuation or divergence between the loss values.

Figure 17: Effect of dataset size and hyperparameter tuning via Optuna on training dynamics of CNN + Transformer + GRU (with attention). (a)–(b) correspond to a smaller dataset (10 000 reads), and (c)–(d) to a larger one (100 000 reads). Each final run has a max of 100 epochs with early stopping implemented. The larger dataset results in significantly higher token and sequence accuracy, and consequently, a lower loss.

Summary and implications. Table 2 collates final test accuracies. Without tuning, increasing data from 1 000 → 100 000 reads gradually raises token accuracy from 16.65% to 75.65% and sequence accuracy from 0% to 9.03%, at the cost of noisy convergence. Training time under this default regime scales roughly linearly—from just over one minute for 1 000 reads up to nearly three hours for 100 000 reads. With Optuna, however, even 10 000 reads suffice to achieve 87% token and 26% sequence accuracy, but its training time rises to 2h 46m (versus 20m untuned). On 50 000 reads, tuning pushes token/sequence scores to 99.78%/97.94% at the expense of a roughly ten-hour run, and on 100 000 reads, it achieves 99.98%/99.76% after over a day and three hours of compute.

Dataset Rows	**Test Token Accuracy**	**Test Sequence Accuracy**	**Training Time (D-H:M:S)**
1,000	16.65%	0.00%	00:01:01
10,000	47.04%	0.00%	00:20:27
50,000	60.06%	1.77%	01:07:00
100,000	75.65%	9.03%	02:45:29
10,000 + Optuna	**87.12%**	**26.20%**	02:46:20
50,000 + Optuna	**99.78%**	**97.94%**	10:11:35
100,000 + Optuna	**99.98%**	**99.76%**	01-03:58:34

Table 2: Test Token, Sequence Accuracy and Total Training Time with and without Hyperparameter Tuning (Optuna).

These timings highlight a clear trade-off: hyperparameter optimisation multiplies training cost by 2–4× depending on data volume, but secures near-perfect motif and sequence accuracy. These results demonstrate that (1) data volume alone improves motif prediction but cannot fully close the token–sequence gap under naive hyperparameters, and (2) careful tuning not only lifts ultimate accuracy but also achieves far more stable, predictable convergence and is required in this architecture to have an impact on sequence accuracy.

Having achieved a model with a motif accuracy $> 95\%$ and sequence accuracy $> 90\%$ means most of the main objectives have been accomplished. The remainder of this chapter therefore focuses on refining encoder and decoder components now that the baseline pipeline is proven to scale effectively.

8.1.3 Comparison of Encoder Variants and Their Impact

Figure 18 summarises the test token and sequence accuracies achieved by each encoder variant on 50 000 reads, trained for up to 30 epochs under the default schedule with default hyperparameters (Appendix A.4) including a constant GRU with attention decoder. Several clear patterns emerge:

- **CNN-only backbones** CNN_FF performs better than no-CNN baselines (Raw, Trans-

former), demonstrating that an initial convolutional frontend helps extract local signal features before any temporal modelling. This is shown as CNN_FF (linear feed-forward after CNN) attains 43.29% token accuracy, surpassing Raw (30.65%) and pure Transformer (15.13%).

- **Bidirectional recurrences** (CNN_BiGRU, BiGRU, BiLSTM) consistently outperform their unidirectional or single-pass counterparts: CNN_BiGRU reaches 66.45% token and 2.28% sequence accuracy, compared to CNN_RNN's 12.43%/0% and CNN_BiLSTM's 33.77%/0.19%. The full BiGRU (no CNN) tops the chart at 76.82% token and 7.98% sequence accuracy, confirming that bidirectional context is vital for decoding motif order under noisy signals.

- **Transformer-based encoders** without CNN (Transformer) lag far behind, capturing only 15.13% token accuracy. This suggests that pure self-attention struggles to learn raw nanopore squiggle patterns without the localised feature extraction afforded by convolutions or recurrent memory.

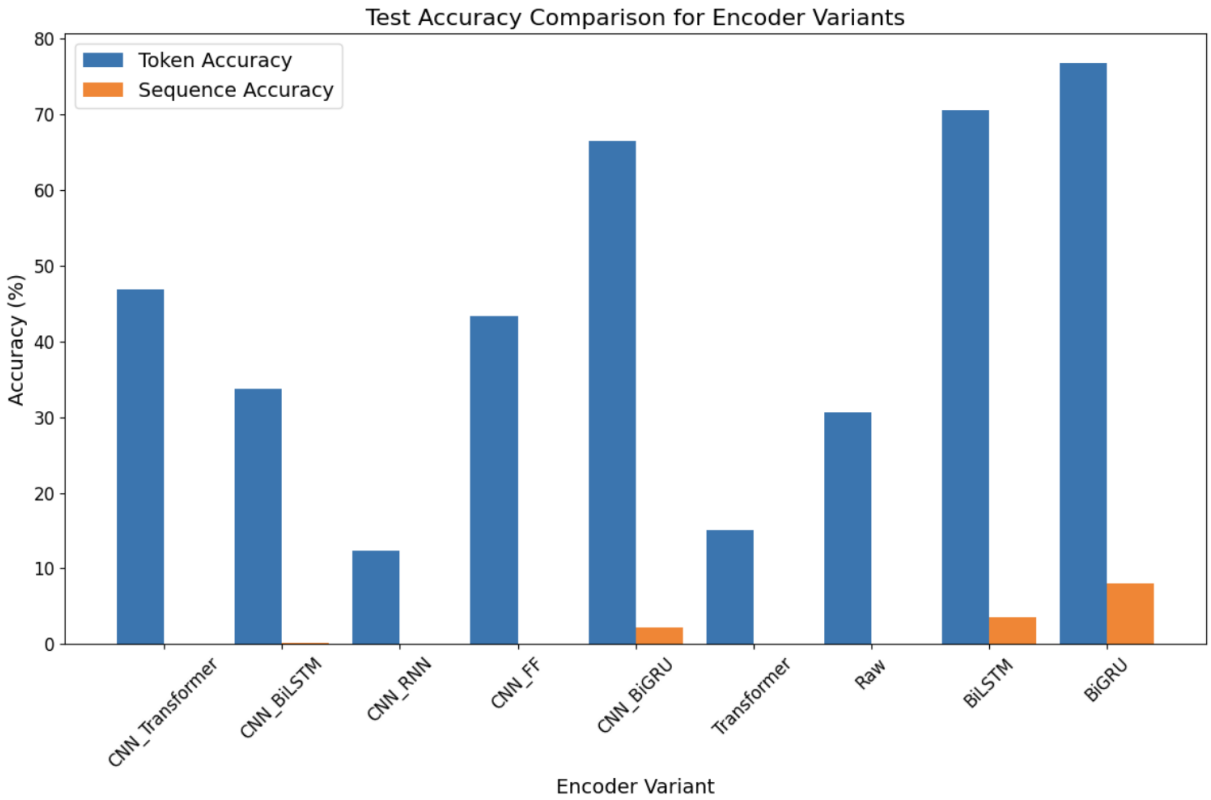


Figure 18: Test token and sequence accuracies for nine encoder variants, all coupled with the same GRU-attention decoder with a default schedule using default hyperparameters (Appendix A.4).

Inference efficiency further differentiates these encoders (Figure 19–20). Figure 19 plots the average forward-pass time on a single GPU:

- **CNN_BiGRU** is fastest (6.38ms), since its lightweight CNN reduces sequence length before the bidirectional GRU.

- **CNN_Transformer** (9.96ms) and **CNN_BiLSTM** (9.02ms) incur modest overhead from self-attention or LSTM gates.
- **Pure BiLSTM** is by far the slowest (28.54ms), reflecting the cost of two full bidirectional LSTM layers over the raw sequence.

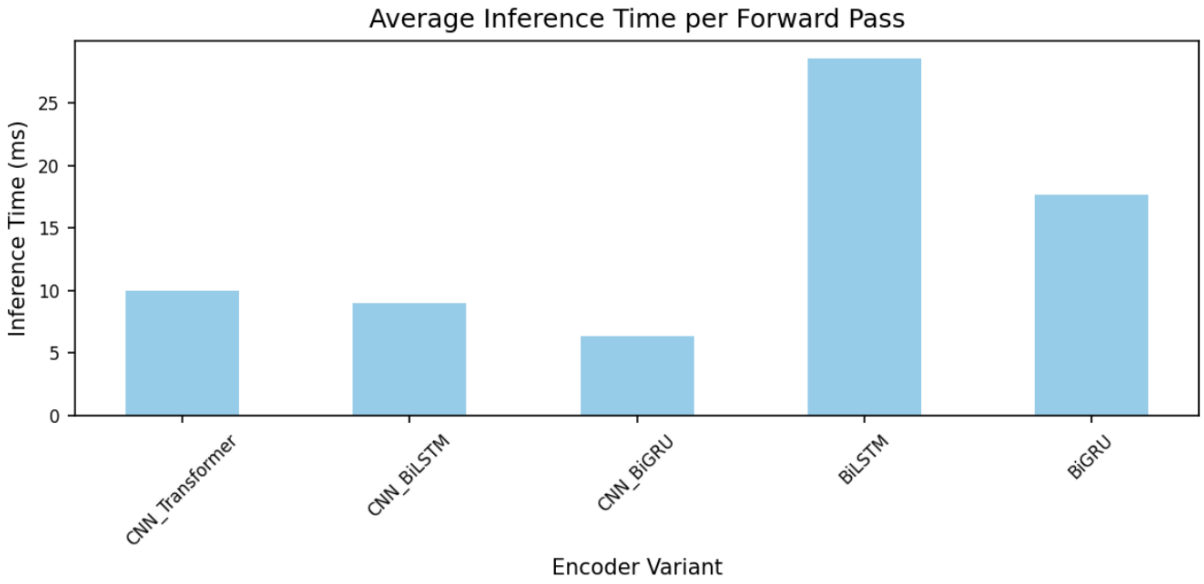


Figure 19: Average inference time per read for five encoder backbones: CNN + Transformer, CNN + BiLSTM, CNN + BiGRU, pure BiLSTM and pure BiGRU. Each time was measured on the target device by running 50 forward passes (after 10 warm-up runs) on a dummy batch of 16 reads. This comparison is chosen to compare some of the best no-CNN models with their CNN counterparts (chosen looking at accuracy scores) as well as compare with the default of CNN + Transformer.

Similarly, peak GPU memory usage during inference (Figure 20) highlights the trade-off between representational power and resource cost:

- **CNN_Transformer**, **CNN_BiGRU**, and **CNN_RNN** all stay under 200MB, making them attractive for deployment on constrained hardware.
- In contrast, **BiLSTM** and **BiGRU** without a CNN frontend consume over 400MB, limiting batch sizes in memory-bounded environments.

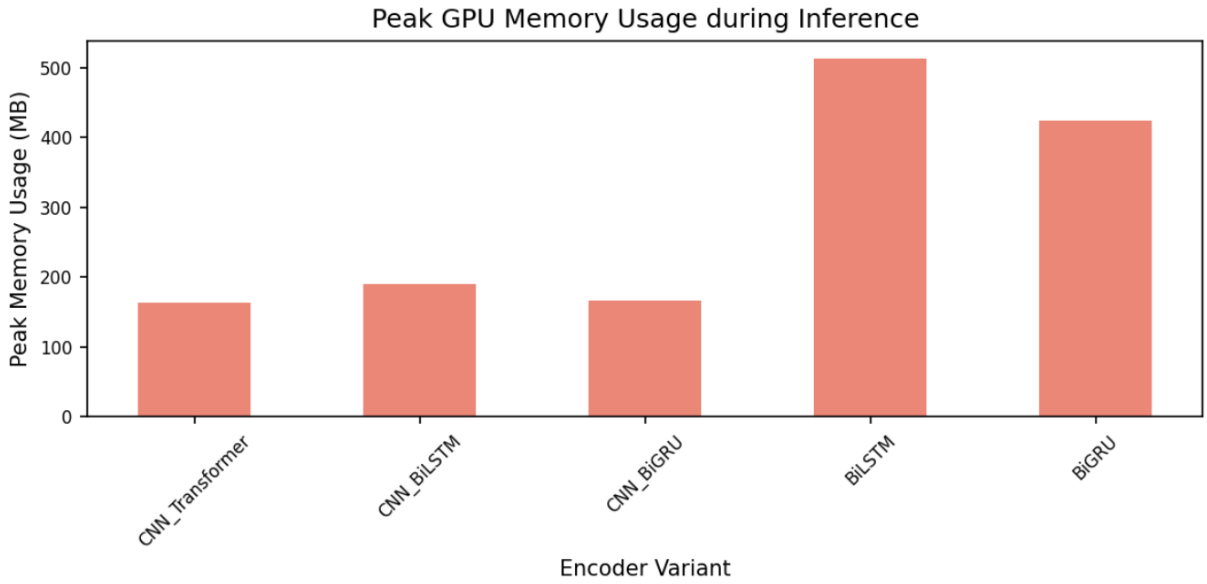


Figure 20: Peak GPU memory usage for the five encoder backbones: CNN+Transformer, CNN+BiLSTM, CNN+BiGRU, pure BiLSTM and pure BiGRU. It measured during a batch inference of 16 reads on CUDA. Memory stats were reset before 50 timed forward passes (after 10 warm-up runs) to capture the true peak allocation of each architecture.

Design implications. While the pure BiGRU encoder delivers the highest raw accuracy (76.82% token, 7.98% sequence), its inference latency (17.64ms) and memory footprint (423.34MB) are markedly higher than its CNN-based counterparts. Conversely, the CNN_BiGRU backbone (66.45% token, 2.28% sequence) combines strong motif-calling performance with the lowest latency (6.38ms) and minimal GPU usage (166.64MB). The CNN_Transformer variant trails only slightly in token accuracy (46.88% vs. 66.45%), yet offers richer long-range modelling via self-attention at a modest cost (9.96ms, 162.43MB).

Accordingly, we select both CNN_BiGRU and CNN_Transformer as our default encoder backbones for the decoder variant study: this dual choice lets us explore whether a decoder can better leverage the Transformer frontend, or whether the efficiency of the BiGRU backbone remains superior when paired with alternative decoding schemes.

8.1.4 Comparison of Decoder Variants and Their Impact

We evaluate eleven decoder variants on both the CNN_BiGRU and CNN_Transformer encoder backbones, measuring token and sequence accuracy, average inference latency, and peak GPU memory. Our goal is to understand which decoding strategy best complements each encoder, balancing accuracy with computational efficiency.

Accuracy on CNN_BiGRU backbone. Figure 21 presents the test token and sequence accuracies for all decoder variants when paired with the CNN_BiGRU encoder. The classical GRU with attention achieves a solid 59.93% token and 3.86% sequence accuracy, slightly outperforming the LSTM and vanilla RNN decoders (27.33% and 58.25% token, 2.07% sequence for RNN). The "Hybrid" decoder—averaging GRU and LSTM outputs—matches the GRU's token score while improving sequence accuracy to 1.66% compared to LSTM. Attention-only

and Conformer variants perform poorly (<15% token), indicating that coupling attention with a recurrence is critical for this task. CTC decoding jumps to 83.48% token and 23.61% sequence accuracy, but is bested by the CRF decoder (99.74% token, 97.50% sequence), confirming recent findings that CRF outperforms CTC in sequential tag decoding.

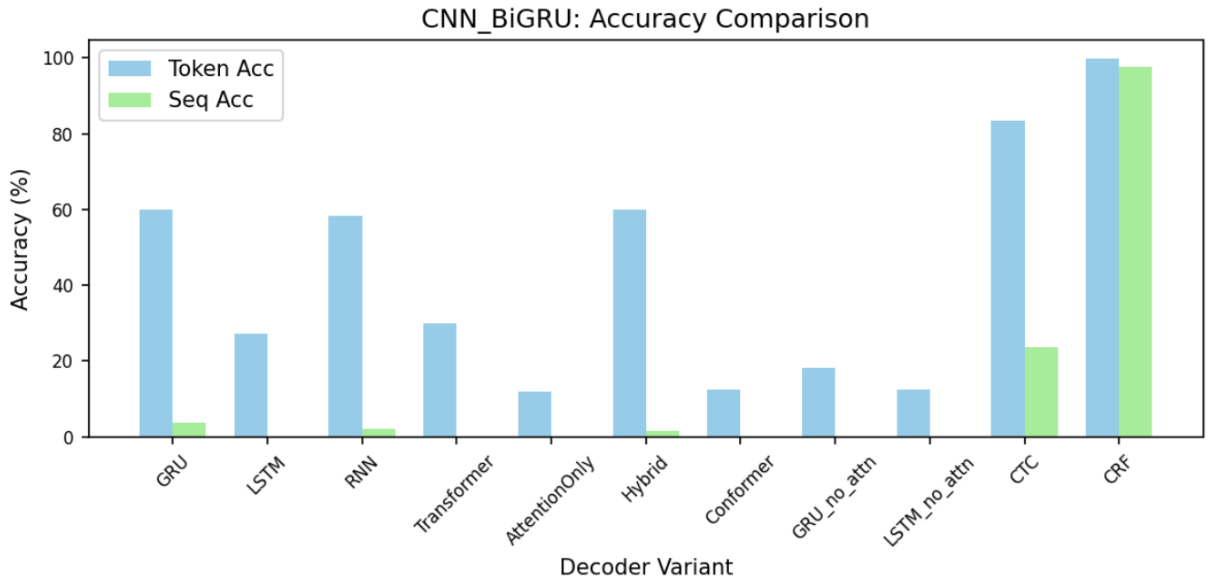


Figure 21: Token-level vs. sequence-level accuracy (%) for various decoder architectures when paired with the CNN + BiGRU encoder. Each pair of bars shows how well the model predicts individual payload motifs versus entire motif sequences across: standard GRU and LSTM decoders, a vanilla RNN, a Transformer decoder, an attention-only module, an RNN+self-attention hybrid, a Conformer block, GRU/LSTM variants with attention disabled, and two end-to-end sequence decoders using CTC and CRF losses.

Latency and memory on CNN_BiGRU backbone. Figures 22 and 23 show inference time and peak GPU memory usage for each decoder on CNN_BiGRU. Lightweight recurrent decoders (GRU, LSTM, vanilla RNN) and their non-attention counterparts (`GRU_no_attn`, `LSTM_no_attn`) achieve sub-8 ms latency and occupy only ≈175–185 MB of GPU memory. CTC is the fastest at ≈4 ms and the leanest at ≈178 MB, making it ideal for high-throughput, low-resource deployments. Transformer-based decoders (standard Transformer, Conformer) incur 36–40 ms latency and peak around 210–220 MB, reflecting the cost of multi-head attention and feed-forward layers. Finally, the CRF decoder, while delivering the highest sequence accuracy, is the slowest (≈122 ms) and uses ≈178 MB. These results underline a clear design trade-off: for real-time performance under tight resource constraints, CTC or simple RNN/GRU decoders are preferred; for applications demanding maximal accuracy, CRF remains attractive despite higher latency.

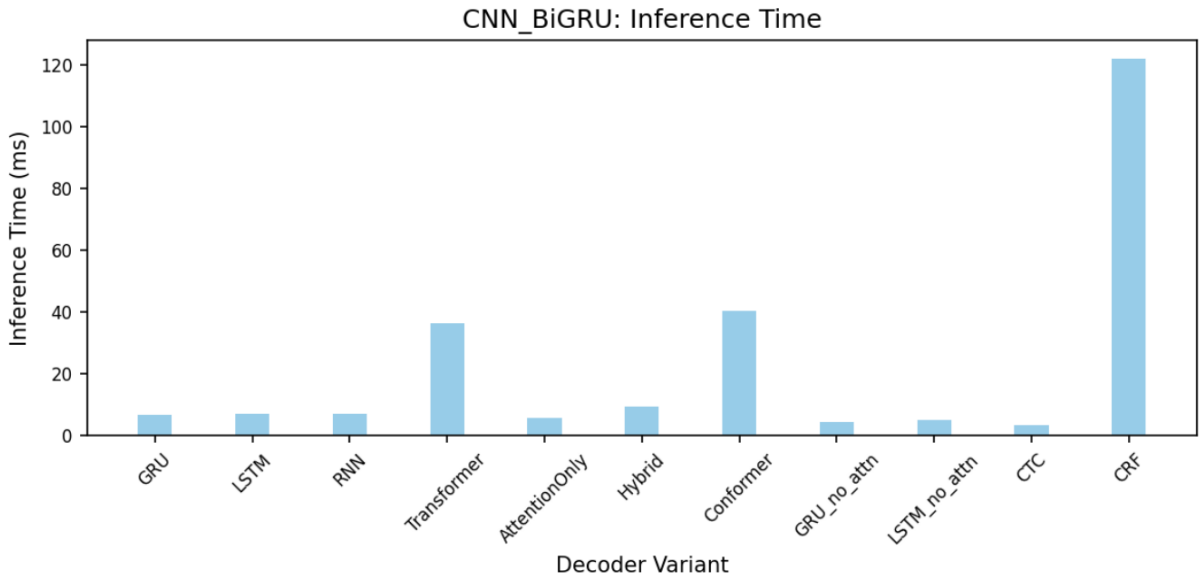


Figure 22: Average inference time per forward pass for each decoder variant paired with the CNN + BiGRU encoder. Measurements are the mean of 50 timed runs on the target device. Simple RNN, GRU and LSTM decoders execute in under 10 ms, attention-only and hybrid modules around 5–10 ms, transformer and Conformer blocks require ~30–40 ms, while the CTC has lowest latency of ~4 ms and CRF decoders incur the highest latency ~120 ms.

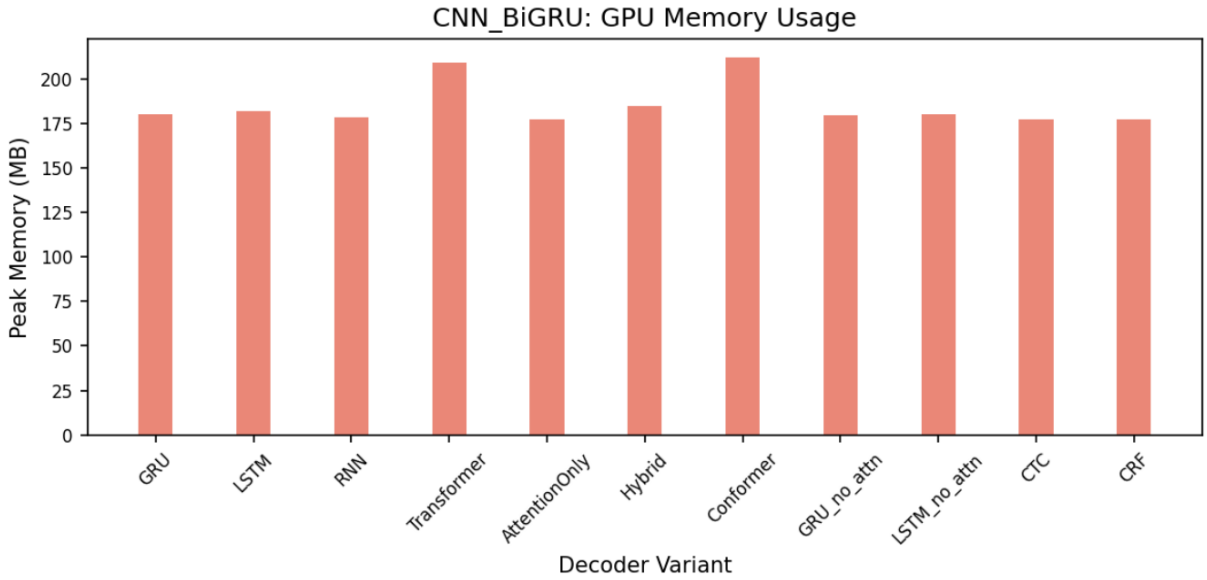


Figure 23: Peak GPU memory consumption (in MB) for each decoder paired with the CNN + BiGRU encoder during inference on a batch of reads. Values were recorded averaged over 50 runs. Transformer and Conformer-based decoders require the most memory (~210–215 MB), while the other decoders use around 180 MB.

Accuracy on CNN_Transformer backbone. When using the CNN_Transformer encoder (Figures 24), the relative ordering of decoders is similar, but overall accuracies shift upward. GRU-attention achieves 72.51% token and 13.51% sequence accuracy—reflecting the Transformer's stronger long-range representations. RNN, LSTM and most other decoders follow at $\leq$62% token, $\leq$2% sequence. CRF again delivers high token accuracy (92.12%) but only 49.48% sequence, whereas the CTC decoder reaches 88.99% sequence accuracy, closing the gap

to perfect motif order recovery.

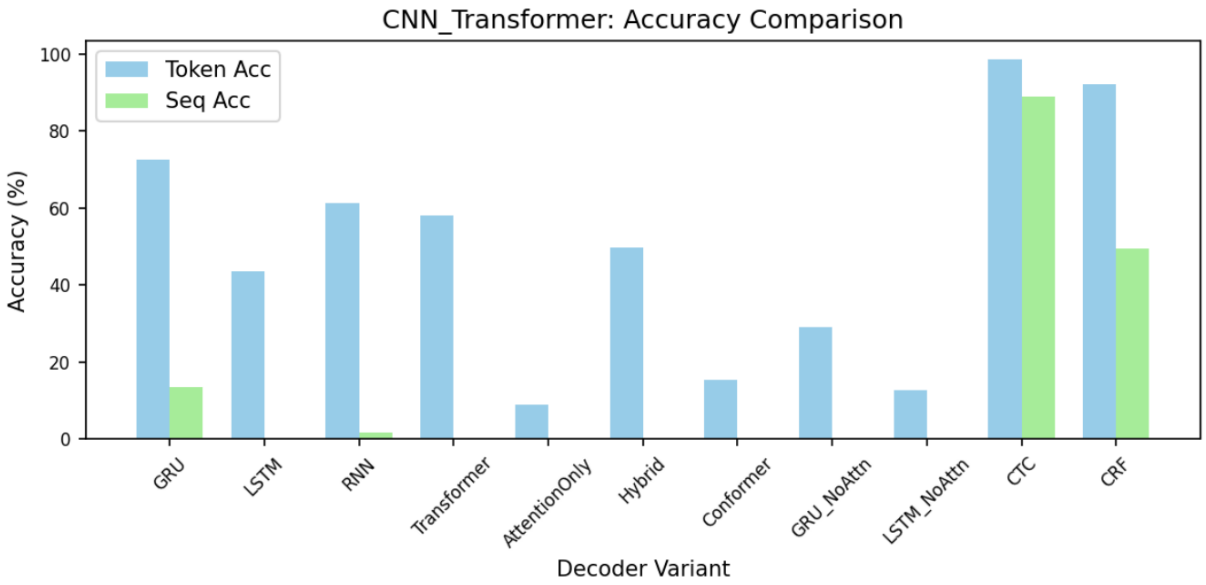


Figure 24: Token-level vs. sequence-level accuracy (%) for various decoder architectures when paired with the CNN + Transformer encoder. Each pair of bars shows how well the model predicts individual payload motifs versus entire motif sequences across: standard GRU and LSTM decoders, a vanilla RNN, a Transformer decoder, an attention-only module, an RNN+self-attention hybrid, a Conformer block, GRU/LSTM variants with attention disabled, and two end-to-end sequence decoders using CTC and CRF losses.

Latency and memory on CNN_Transformer backbone. Figures 25 and 26 report the computational costs.

The relative ordering observed in CNN_BiGRU persists: recurrent decoders and CTC run in 7–11 ms and use ≈180 MB, Transformer and Conformer decoders require ≈39–42 ms and ≈215 MB, and CRF again peaks at ≈125 ms with ≈180 MB memory. The slight latency uptick compared to `CNN_BiGRU` reflects additional encoder–decoder coupling overhead, but peak memory remains nearly identical. Together, these profiles confirm that decoder choice drives the dominant variation in inference cost, guiding practitioners to select a decoder that balances throughput, resource budget, and accuracy targets for their specific motif-calling task.

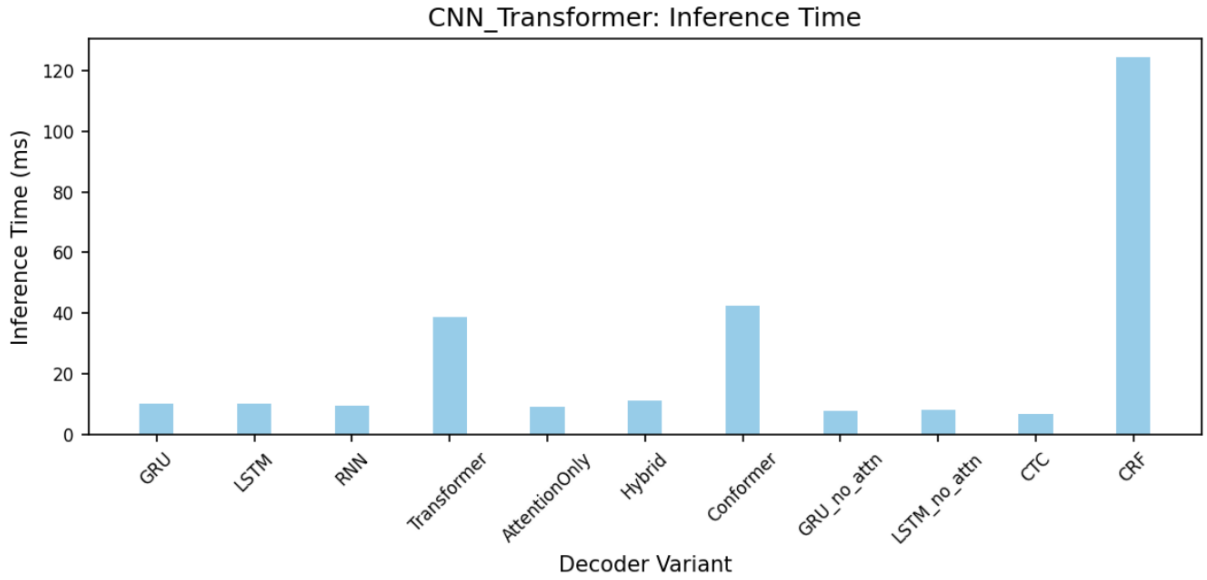


Figure 25: Average inference time per forward pass for each decoder variant paired with the CNN + Transformer encoder. Measurements are the mean of 50 timed runs on the target device. Simple RNN, GRU and LSTM decoders execute in ~10 ms, attention-only and hybrid modules around 10–15 ms, transformer and Conformer blocks require ~40–45 ms, while the CTC has the lowest latency of ~8 ms and CRF decoders incur the highest latency ~125 ms.

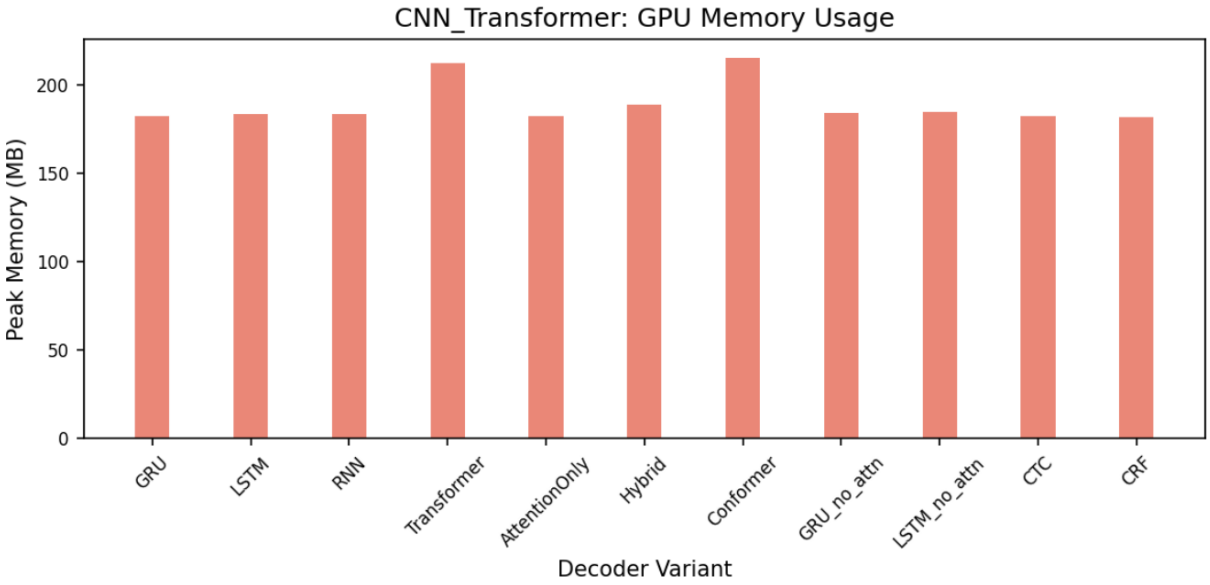


Figure 26: Peak GPU memory consumption (in MB) for each decoder paired with the CNN + Transformer encoder during inference on a batch of reads. Values were recorded and averaged over 50 runs. Transformer and Conformer-based decoders require the most memory (~210–215 MB), while the other decoders use around 180 MB.

Key insights. Taken together, these findings guide system designers to match decoder complexity to application needs:

- *Real-time, resource-constrained* deployments: CTC or simple GRU decoders offer sub-10ms throughput with moderate accuracy.
- *Balanced accuracy and latency*: The CNN_Transformer + CTC pairing hits 92.12%

sequence recall in ≈10ms. Meanwhile, CNN_BiGRU + CTC also achieves an impressive 83.48% and 23.61% in less than 10ms.

- *Maximum accuracy*: CRF decoding on either encoder backbone delivers near-perfect results when latency is secondary.

To ensure our final recommendation was not biased by any particular defaults, we retrained the leading candidates—CNN–Transformer–CTC and CNN–BiGRU–CTC—using the refined hyperparameter schedule, and then conducted a final analysis to identify the architecture that best balances sequence accuracy and latency.

8.1.5 Final Model Showcase

Table 3 presents the fully tuned performance of the two top contenders, each trained on 50 000 reads under the refined hyperparameter schedule. Both models reach virtually perfect motif recovery, with token accuracies above 99.95% and sequence accuracies above 99.6%. However, they differ notably in training cost and marginal gains.

Model	**Test Token Accuracy**	**Test Sequence Accuracy**	**Training Time (D-H:M:S)**
CNN_BiGRU_CTC (50,000 + Optuna)	99.96%	99.64%	07:31:05
CNN_Transformer_CTC (50,000 + Optuna)	99.97%	99.73%	10:59:46

Table 3: Comparison of CNN_BiGRU_CTC and CNN_Transformer_CTC on 50,000 + Optuna for Test Token Accuracy, Test Sequence Accuracy, and Training Time.

Although the Transformer variant achieves the highest sequence accuracy (an improvement of just 0.09%), the BiGRU alternative attains nearly identical results in substantially less compute time. When combined with their inference profiles (presented in Section 8.1.4), where CNN_Transformer_CTC decodes a read in ≈10 ms versus ≈6 ms for CNN_BiGRU_CTC.

In light of these results, our final model architecture choice is CNN_BiGRU_CTC due to its impressive balance of >99.6% sequence accuracy with ≈30% lower training cost and ≈40% faster inference.

This model not only achieves our initial objectives of >95% motif accuracy and >90% sequence accuracy, but the model created from this test can be used in the CLI to produce a working backend.

8.2 Training Dynamics and Convergence

In this section, we examine how our models learn over time, focusing on loss trajectories and the stability of training under different data volumes and hyperparameter schedules. By comparing default and hyperparameter schedules on both small and large synthetic datasets, we reveal how convergence behaviour correlates with final motif-calling performance.

8.2.1 Loss Curves and Learning Stability

We begin by inspecting how training and validation losses evolve under different dataset sizes and hyperparameter settings. With the default hyperparameters on 10 000 training reads (Figure 16b), both curves descend from around 1.95 to 1.20 over 90 epochs, but exhibit frequent oscillations of up to 0.1–0.2 in loss, reflecting noisy gradient updates and a non-optimised learning rate and teacher-forcing schedule. Scaling to 100 000 reads under the same defaults (Figure 16d) lowers the minimum of the loss to approximately 0.60, yet the amplitude of fluctuations actually increases. This suggests that while more data leads to stronger overall gradient estimates, the fixed hyperparameters still allow significant minibatch variability, leading to the observed ripples even as the average loss continues its downward trend.

By contrast, introducing Optuna-tuned hyperparameters dramatically smooths these curves. On 10 000 reads with the refined schedule (Figure 17b), both training and validation losses fall almost monotonically, tracking closely for the first 50 epochs before a small gap emerges, an early indication of subtle overfitting as teacher forcing decreases. With 100 000 reads and tuning (Figure 17d), both loss curves decrease to near zero within 20 epochs and then remain virtually flat, demonstrating that optimised learning rates, dropout and weight-decay collectively reduce gradient noise and accelerate convergence.

8.2.2 Overfitting and Generalisation Observations

Across all runs, training and validation losses generally track one another without sustained divergence, indicating that overfitting is well controlled as early stopping further guards against late-stage over-specialisation. However, the small 10 000-row default run still shows a noisy validation loss (Figure 16b) and its sequence accuracy remains near zero (Figure 16a), reflecting limited generalisation when data are scarce. Increasing to 100 000 reads under default settings results in a modest sequence accuracy of around 9% (Figure 16c), despite larger loss oscillations, confirming that data volume alone can improve motif ordering but cannot fully stabilise learning without hyperparameter adjustments.

With Optuna tuning, even the smaller dataset generalises much better: the 10 000 + Optuna run reaches approximately 26% sequence accuracy by epoch 100 (Figure 17a), and the 100000 + Optuna run saturates both token and sequence accuracy near 100% (Figure 17c), all without divergent loss behaviour. The combination of an exponentially decaying teacher-forcing ratio, a scheduler, and early stopping creates a productive balance, providing enough learning flexibility to fit complex patterns while imposing timely checkpoints to prevent over-memorisation.

8.3 Critical Discussion

Drawing together the quantitative results and component-level analyses, this section reflects on the practical implications of our final model choice and the broader lessons for end-to-end motif decoding (see detailed key architectural rationale in Appendix A.6). Our CNN–BiGRU–CTC pipeline aligns with the motif-based decoding architecture proposed by other literature exploring this problem, but extends it significantly: we systematically benchmark nine encoder and eleven decoder variants, introduce robust interquartile-range normalisation, employ Optuna-

driven hyperparameter optimisation for stable, high-fidelity convergence, and rigorously profile inference latency and GPU memory under real-time constraints [49].

8.3.1 Strengths of Final Model

Our chosen CNN–BiGRU–CTC pipeline delivers an exceptionally strong combination of performance and efficiency:

- **High sequence recall at low latency:** Achieving over 99.6% sequence accuracy while decoding each read in under 6 ms makes this model suitable for real-time or near-real-time applications.
- **Robustness to signal variability:** The `robust_no_centre` normalisation preserved key amplitude patterns across synthetic squiggles, reducing sensitivity to outlier currents and producing consistently high accuracy.
- **Scalable training and inference:** Training converges in just over 7 hours on 50 000 reads under the refined hyperparameter schedule, and peak GPU memory remains under 200 MB, enabling deployment on modest hardware.
- **Simplicity of integration:** The model relies solely on standard PyTorch layers (Conv1D, GRU, CTC loss), facilitating rapid incorporation into existing motif-based DNA-storage pipelines.

8.3.2 Limitations and Error Sources

Despite its advantages, our final pipeline has several caveats:

- **Accuracy ceiling vs. CRF:** While CTC decoding offers low latency, CRF variants attained marginally higher sequence recovery (up to 92.12% without tuning; Figure 24), suggesting that structured decoding can still outperform in scenarios where every error matters.
- **Hyperparameter sensitivity:** Performance and stability depend on careful tuning of learning rates, dropout and teacher-forcing schedules. Default settings produced noisy training curves, underscoring the need for a dedicated optimisation phase.
- **Synthetic data bias:** All experiments used simulated reads; real nanopore signals may exhibit additional noise characteristics or context effects not captured by Squigulator, potentially degrading out-of-sample performance.
- **Residual alignment drift:** CTC's blank symbol mechanism can still misplace motif boundaries in exceptionally dense or low-signal regions, leading to rare insertion/deletion errors.

8.3.3 Lessons Learnt from Component Comparisons

Our systematic sweep of encoders and decoders yields several transferable insights:

- **Early convolution is essential:** CNN front-ends consistently outperformed pure sequence models (Transformer or recurrent) by denoising and down-sampling, simplifying downstream pattern recognition in terms of the accuracy and inference speed tradeoff.
- **Normalisation shapes everything:** The five-fold token accuracy boost under `robust_no_centre` highlights that handling outliers via IQR scaling is critical for electrical-signal data.
- **Decoder choice drives trade-offs:** CTC provides the best speed-accuracy balance, while CRF delivers the highest accuracy when latency is secondary.
- **Hyperparameter optimisation pays dividends:** Optuna-tuned runs converged faster, with smoother loss curves and dramatically improved sequence recall, even on small datasets.

These reflections not only justify our final CNN–BiGRU–CTC selection but also offer a blueprint for future motif-calling pipelines in both research and industrial DNA-storage settings.

Conclusions and Future Work

In this final report, we have demonstrated that deep neural networks can be harnessed to decode payload motifs directly from raw nanopore signals, bypassing the traditional basecalling and sequence alignment pipeline. By systematically exploring architectural choices, training regimes and preprocessing methods, we have not only achieved near-perfect motif recovery on synthetic data but also provided a clear blueprint for bringing these methods into practical DNA storage workflows. In the sections that follow, we summarise our key contributions, acknowledge the limitations of the present work, and outline promising directions for future research.

9.1 Summary of Key Contributions

This section discusses the working end-to-end pipeline that achieves state-of-the-art accuracy on synthetic data, but also presents a set of design guidelines that can inform future developments in nanopore-based signal-to-sequence modelling.

9.1.1 Proof-of-Concept Feasibility Demonstrated

We have shown that end-to-end motif calling from raw electrical signal to discrete motif sequences is not only possible but can achieve high accuracy with limited resource requirements. By synthesising a large, labelled dataset and implementing a flexible PyTorch pipeline, we demonstrated, across multiple scales of training data, that our models can consistently recover motif tokens with >99% token accuracy and, with hyperparameter tuning, approach perfect sequence reconstruction. Having a well-performing model means its training model can be saved and used for the CLI backend to truly work as a proof-of-concept. This work supports the viability of deep neural network architectures in nanopore signal decoding beyond traditional base-calling, paving the way for rapid, on-device motif identification.

9.1.2 Insights into Model Architecture Decisions

Our extensive comparison of encoder–decoder combinations showcased clear design principles. First, a convolutional front end that extracts local signal features and suppresses noise consistently improved downstream performance, underscoring the importance of early signal denoising. Secondly, starting with CNN, Transformer encoders outperformed pure recurrent layers at capturing dependencies across extended signal windows, though bidirectional GRUs offered a compelling speed-accuracy trade-off. Thirdly, when comparing decoders, CTC emerged as the optimal choice for balancing decoding speed and order-sensitive sequence recovery, while CRF retained an advantage in absolute sequence accuracy at the cost of higher latency. Finally, the choice of normalisation proved ver important as a non-centring scaling method beat signal distributions like standard z-score or min-max approaches, by directly translating to improved accuracy. These findings not only guided our final CNN–BiGRU–CTC architecture but also provide a transferable blueprint for similar signal-to-sequence decoding tasks.

9.2 Limitations and Constraints

While our end-to-end motif-calling pipeline achieves outstanding results on synthetic datasets, several overarching constraints must be acknowledged before deployment in practical settings. These pertain to the validity of simulated data, the computational and time costs of modern neural architectures, and the opportunity for further model refinements inspired by recent advances in nanopore basecalling.

9.2.1 Synthetic Data Bias and Generalisability

While the synthetic nanopore signals generated by Squigulator provided a rich training dataset and allowed controlled experimentation, they inevitably diverge from the full complexity of real-world sequencing data. In particular, the noise characteristics, variability and subtle pore chemistry effects simulated in software cannot capture all sources of random error encountered in an actual run. As a result, models trained exclusively on synthetic traces may overfit to the simulator's particular noise profile and fail to generalise when faced with unmodelled motif contexts [54]. Bridging this domain gap will require careful calibration of the simulation parameters against empirical measurements, or the introduction of domain-adaptation techniques to transfer knowledge from synthetic to real data without an exhaustive re-annotation of experimental reads.

9.2.2 Hardware, Computational, and Time Limitations

The end-to-end neural architectures explored in this work (like those incorporating transformer layers or CRF decoding) demand substantial GPU memory and compute time during both training and inference. Although CTC decoders and lightweight recurrent alternatives can meet real-time throughput requirements, their reduced capacity can sometimes undermine sequence accuracy in the presence of extensive signal noise. In contrast, the highest-accuracy CRF pipelines incur latency that currently prevents deployment on low-power, embedded sequencing devices. These trade-offs highlight the need for further work on model compression, streaming inference optimisations and hardware-aware architecture search to reconcile the competing demands of speed, memory footprint and decoding fidelity. Moreover, transformer encoders themselves could be enhanced by drawing on innovations such as sliding-window multi-head attention, which has proven effective in recent nanopore basecallers, to reduce computational overhead while preserving long-range context modelling [42].

9.3 Future Work

Building on the foundations established in this report, there are several avenues to extend and apply our `Motifcaller` learnings. These include unsupervised pre-training to reduce label dependence, adaptation of the learned architectures to other nanopore signal tasks, validation on real sequencing runs, and seamless integration into full DNA storage pipelines.

9.3.1 Unsupervised and Self-Supervised Learning Extensions

Moving beyond purely supervised training on annotated synthetic reads, future research should investigate unsupervised or self-supervised pre-training strategies. Approaches such as masked signal reconstruction, contrastive learning of temporal embeddings or autoencoding of squiggle fragments could allow the model to learn richer signal representations without reliance on ground-truth motif labels [55]. By fine-tuning these pre-trained encoders on smaller labelled datasets, one could enable further resistance to noise variations and reduced annotation effort when adapting to new motif sets, essentially helping to future-proof a model.

9.3.2 Applications to Other Nanopore Signal Analysis Tasks

The core architectures and training regimes developed in this report are directly transferable to a range of nanopore sequencing problems. For example, targeted microbial sequencing often requires rapid classification of short amplicons; the work could be repurposed to map raw squiggles to species-specific signatures in real time [56]. Exploring alternative applications will both validate the generality of our approach and uncover domain-specific tweaks to enhance performance across various fields.

9.3.3 Expansion to Real-World Nanopore Data

An extension to this project is to validate and refine the motif-caller on genuine nanopore runs. This is possible due to access to Helixwork's data being granted which can help mitigate simulation bias and unlock the full potential of the architectures. Success in this arena will not only demonstrate practical utility but also illuminate new error modes and guide algorithmic improvements.

9.3.4 Integration into End-to-End DNA Storage Pipelines

Ultimately, the `Motifcaller` is an important seamless component of a complete DNA data storage workflow, from sequence design and synthesis through to automated decoding and error correction. Future efforts should focus on developing standardised APIs and command-line tools like shown in this report, that ingest raw Fast5 files, orchestrate decoding, perform payload reconstruction and interface with higher-level storage management software. Such integration will pave the way for truly cost-effective DNA storage systems capable of high-capacity, rapid read-out in both laboratory and field settings.

9.4 Author's Remarks

Reflecting on this journey, it is clear that DNA-based data storage remains, for the foreseeable future, an emerging technology rather than a direct challenger to magnetic tape for large-scale archival needs. In today's "warm" and "hot" data tiers, driven by AI and real-time analytics, electronic media continue to dominate where access latency and throughput dominate. Yet history shows many examples of expensive curiosities which, through breakthroughs, begin to transform into innovations that we render indispensable. DNA storage's intrinsic stability and

capability of preserving encoded information for centuries without power offers a compelling vision of truly permanent data stores, immune to media degradation and obsolescence.

Beyond the immediate DNA-storage application, the methods developed here speak to a broader bioinformatics frontier. Nanopore signal decoding underpins genomics, epigenetics, pathogen surveillance and beyond, and the same neural architectures and decoding strategies explored in this report can be re-applied wherever electrical readouts must be translated into symbolic sequences. The rapid strides in natural-language and speech model are finding equal importance in molecular informatics. Witnessing these parallels has been one of the most exciting aspects of this work: models once trained on sentences and phonemes can, with suitable adaptation, unlock decoding information from DNA effectively.

In closing, this report stands not only as a proof-of-concept for motif-level DNA storage decoding, but as a testament to the cross-pollination of ideas between AI and biotechnology. We are living in an era where advances in one domain echo across many others, and it is my hope that the lessons learned here will inspire both deeper investigations into DNA as a storage medium and creative applications of deep learning to the rich, noisy signals that emerge from our laboratory instruments. The road ahead seems exciting.

Bibliography

[1] P. Taylor, "Amount of data created, consumed, and stored 2010-2023, with forecasts to 2028," 2024, accessed: March 10, 2025. [Online]. Available: https://www.statista.com/statistics/871513/worldwide-data-created/

[2] L. Ceze, J. Nivala, and K. Strauss, "Molecular digital data storage using dna," *Nature Reviews Genetics*, vol. 20, no. 8, pp. 456–466, 2019, accessed: March 15, 2025. [Online]. Available: https://doi.org/10.1038/s41576-019-0125-3

[3] T. S. Council, "Tape storage council market outlook 2022," 2022, accessed: March 17, 2025. [Online]. Available: https://tapestorage.org/wp-content/uploads/Tape-Storage-Council-Market-Outlook-2022_Final.pdf

[4] K. Bartley, "Big data statistics: How much data is there in the world?" 2024, accessed: March 17, 2025. [Online]. Available: https://rivery.io/blog/big-data-statistics-how-much-data-is-there-in-the-world/

[5] T. Heinis, R. Sokolovskii, and J. J. Alnasir, "Survey of information encoding techniques for dna," *ACM Computing Surveys*, vol. 56, no. 4, pp. 1–30, 2023, accessed: March 17, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3626233

[6] Y. Yan, N. Pinnamaneni, S. Chalapati, C. Crosbie, and R. Appuswamy, "Scaling logical density of dna storage with enzymatically-ligated composite motifs," *Scientific Reports*, vol. 13, p. 15978, 2023, accessed: March 24, 2025. [Online]. Available: https://doi.org/10.1038/s41598-023-43172-0

[7] T. Bioscience, "Dna-based digital storage," 2018, accessed: March 24, 2025. [Online]. Available: https://www.twistbioscience.com/sites/default/files/resources/2019-03/WhitePaper_DataStorage_29Oct18_Rev1.pdf

[8] G. Editors, "Differences between hot data and cold data – system design," 2024, accessed: March 28, 2025. [Online]. Available: https://www.geeksforgeeks.org/differences-between-hot-data-and-cold-data-system-design/

[9] D. Robb, "Tape storage in 2023: What do you need to know?" 2023, accessed: April 2, 2025. [Online]. Available: https://www.enterprisestorageforum.com/backup/tape-storage-for-enterprise-backups/

[10] W. C. Preston, "Why aren't optical disks the top choice for archive storage?" 2021, accessed: April 4, 2025. [Online]. Available: https://www.networkworld.com/article/970376/why-aren-t-optical-disks-the-top-choice-for-archive-storage.html

[11] S. Editors, "Ww record hdd capacity of 32tb," 2024, accessed: April 5, 2025. [Online]. Available: https://www.storagenewsletter.com/2024/10/18/ww-record-hdd-capacity-of-32tb/

[12] W. D. Corporation, "Ultrastar dc hc580 data center hdd," 2025, accessed: April 5, 2025. [Online]. Available: https://www.westerndigital.com/products/internal-drives/data-center-drives/ultrastar-dc-hc580-hdd?sku=0F62785

[13] J. Marciniec, "The most economical way to archive your data," 2020, accessed: April 5, 2025. [Online]. Available: https://www.jacobmarciniec.com/blog/the-most-economical-way-to-archive-your-data

[14] A. Doricchi, C. M. Platnich, A. Gimpel, F. Horn, M. Earle, G. Lanzavecchia, A. L. Cortajarena, L. M. Liz-Marzán, N. Liu, R. Heckel, R. N. Grass, R. Krahne, U. F. Keyser, and D. Garoli, "Emerging approaches to dna data storage: Challenges and prospects," *ACS Nano*, vol. 16, no. 11, pp. 17 552–17 571, 2022, accessed: April 5, 2025. [Online]. Available: https://doi.org/10.1021/acsnano.2c06748

[15] IBM, "Introduction to the ibm ts4500 tape library," 2022, accessed: April 2, 2025. [Online]. Available: https://www.ibm.com/docs/en/ts4500-tape-library?topic=overview-introduction-ts4500-tape-library

[16] S. Shankland, "Startup packs all 16gb of wikipedia onto dna strands to demonstrate new storage tech," 2019, accessed: April 6, 2025. [Online]. Available: https://www.cnet.com/tech/computing/startup-packs-all-16gb-wikipedia-onto-dna-strands-demonstrate-new-storage-tech/

[17] Symply, "Lto superpowers: Longevity and reliability," 2023, accessed: April 6, 2025. [Online]. Available: https://gosymply.com/news/lto-superpowers-longevity-and-reliability/

[18] F. Corporation, "Optical disk storage vs. tape storage," accessed: April 4, 2025. [Online]. Available: https://www.tape-storage.net/en/storage_comparison/article_02/

[19] A. Rana, "Hard drive power consumption: A comprehensive guide," 2024, accessed: April 5, 2025. [Online]. Available: https://storedbits.com/hard-drive-power-consumption/

[20] I. Sunstar Company, "Ibm ts4500 tape library," accessed: April 3, 2025. [Online]. Available: https://sunstarco.com/ibm/tape-libraries/ibm-ts4500-tape-libary/

[21] Arcserve, "Data storage lifespans: How long will media really last?" 2024, accessed: April 2, 2025. [Online]. Available: https://www.arcserve.com/blog/data-storage-lifespans-how-long-will-media-really-last#:~:text=Some%20manufacturers%20claim%20that%20tape,and%20tear%20if%20used%20frequently

[22] J. Schmerker, "Using dna for stable, long-term, limitless data storage," 2024, accessed: April 6, 2025. [Online]. Available: https://eu.idtdna.com/pages/community/blog/post/using-dna-for-stable-long-term-limitless-data-storage

[23] A. Sherif, "Information archiving solutions market size worldwide 2014-2027," 2023, accessed: April 4, 2025. [Online]. Available: https://www.statista.com/statistics/498032/information-archiving-market/

[24] LibreTexts, "Storing genetic information," accessed: April 7, 2025. [Online]. Available: https://bio.libretexts.org/Courses/Lumen_Learning/Biology_for_Non-Majors_I_%28Lumen%29/08%3A_DNA_Structure_and_Replication/8.02%3A_Storing_Genetic_Information

[25] Y. Dong, F. Sun, Z. Ping, Q. Ouyang, and L. Qian, "Dna storage: research landscape and future prospects," *National Science Review*, vol. 7, no. 6, pp. 1092–1107, 2020, accessed: April 7, 2025. [Online]. Available: https://doi.org/10.1093/nsr/nwaa007

[26] R. Sokolovskii, P. Agarwal, L. A. Croquevielle, Z. Zhou, and T. Heinis, "Coding over coupon collector channels for combinatorial motif-based dna storage," *IEEE Transactions on Communications*, pp. 1–1, 2024, accessed: April 7, 2025. [Online]. Available: https://doi.org/10.1109/TCOMM.2024.3506938

[27] N. Goldman, P. Bertone, S. Chen, C. Dessimoz, E. M. LeProust, B. Sipos, and E. Birney, "Towards practical, high-capacity, low-maintenance information storage in synthesized dna," *Nature*, vol. 494, no. 7435, pp. 77–80, 2013, accessed: April 8, 2025. [Online]. Available: https://doi.org/10.1038/nature11875

[28] H. Chen, B. Wang, L. Cai, Y. Zhang, Y. Shu, W. Liu, X. Leng, J. Zhai, B. Niu, Q. Zhou, and S. Cao, "The performance of homopolymer detection using dichromatic and tetrachromatic fluorogenic next-generation sequencing platforms," *BMC Genomics*, vol. 25, no. 542, 2024, accessed: April 8, 2025. [Online]. Available: https://doi.org/10.1186/s12864-024-10474-0

[29] O. N. Technologies, "How nanopore sequencing works," 2019, accessed: March 25, 2025. [Online]. Available: https://www.youtube.com/watch?v=RcP85JHLmnI

[30] V. Menkovski, "Simulation of nanopore sequencing (with generative models)," accessed: April 8, 2025. [Online]. Available: https://vlamen.github.io/mscproject/nanopore/

[31] O. N. Technologies, "How nanopore sequencing works," accessed: April 8, 2025. [Online]. Available: https://nanoporetech.com/platform/technology

[32] ——, "How basecalling works," accessed: April 11, 2025. [Online]. Available: https://nanoporetech.com/platform/technology/basecalling

[33] ——, "Promethion documentation," accessed: April 8, 2025. [Online]. Available: https://nanoporetech.com/document/promethion

[34] M. Pagès-Gallego and J. de Ridder, "Comprehensive benchmark and architectural analysis of deep learning models for nanopore sequencing basecalling," *Genome Biology*, vol. 24, no. 71, 2023, accessed: April 9, 2025. [Online]. Available: https://doi.org/10.1186/s13059-023-02903-2

[35] Q. Li, C. Sun, D. Wang, and J. Lou, "Gcrtcall: a transformer based basecaller for nanopore rna sequencing enhanced by gated convolution and relative position embedding via joint

loss training," *Frontiers in Genetics*, vol. Volume 15 - 2024, 2024. [Online]. Available: https://www.frontiersin.org/journals/genetics/articles/10.3389/fgene.2024.1443532

[36] N. Huang, F. Nie, P. Ni, F. Luo, and J. Wang, "Sacall: A neural network basecaller for oxford nanopore sequencing data based on self-attention mechanism," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 19, no. 1, pp. 614–623, 2022. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9264702

[37] O. N. Technologies, "Nanopore sequencing accuracy," accessed: April 10, 2025. [Online]. Available: https://nanoporetech.com/platform/accuracy

[38] F. H. Rosyidi, "Matrix application in the smith-waterman algorithm for dna local alignment," 2024, accessed: April 12, 2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/AljabarGeometri/2024-2025/Makalah/Makalah-IF2123-Algeo-2024%20(83).pdf

[39] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018, accessed: April 12, 2025.

[40] J. H. D. Brandao, H. da Costa Oliveira, A. G. da Costa Martins, J. B. Pesquero, B. M. Verona, and N. N. P. Cerize, "How close are we to storing data in dna?" *Trends in Biotechnology*, vol. 42, no. 2, pp. 156–167, 2024, accessed: March 24, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167779923002354

[41] J. Silvestre-Ryan and I. Holmes, "Pair consensus decoding improves accuracy of neural network basecallers for nanopore sequencing," *Genome Biology*, vol. 22, no. 38, 2021, accessed: April 10, 2025. [Online]. Available: https://doi.org/10.1186/s13059-020-02255-1

[42] S. Davis, "Transforming basecalling in genomic sequencing," 2024, accessed: April 12, 2025. [Online]. Available: https://nanoporetech.com/blog/transforming-basecalling-in-genomic-sequencing

[43] Q. Wen, T. Zhou, C. Zhang, W. Chen, Z. Ma, J. Yan, and L. Sun, "Transformers in time series: A survey," in *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence (IJCAI-23)*. International Joint Conferences on Artificial Intelligence, 2023, pp. 6778–6786, accessed: April 12, 2025.

[44] O. N. Technologies, "Q system data analysis," accessed: April 13, 2025. [Online]. Available: https://nanoporetech.com/ja/document/q-system-data-analysis

[45] I. D. Mienye, T. G. Swart, and G. Obaido, "Recurrent neural networks: A comprehensive review of architectures, variants, and applications," *Information*, vol. 15, no. 9, p. 517, 2024, accessed: April 12, 2025. [Online]. Available: http://dx.doi.org/10.3390/info15090517

[46] H. Gamaarachchi, J. M. Ferguson, H. Samarakoon, K. Liyanage, and I. W. Deveson, "Squigulator: simulation of nanopore sequencing signal data with tunable

noise parameters," 2023, accessed: March 28, 2025. [Online]. Available: https://github.com/hasindu2008/squigulator

[47] T. Bäckström, O. Räsänen, A. Zewoudie, P. P. Zarazaga, L. Koivusalo, S. Das, E. G. Mellado, M. B. Mansali, D. Ramos, and M. H. Vali, "Short-time analysis of speech and audio signals," accessed: April 12, 2025. [Online]. Available: https://speechprocessingbook.aalto.fi/Representations/Short-time_analysis.html

[48] J. M. Perero-Codosero, F. M. Espinoza-Cuadros, and L. A. Hernández-Gómez, "A comparison of hybrid and end-to-end asr systems for the iberspeech-rtve 2020 speech-to-text transcription challenge," *Applied Sciences*, vol. 12, no. 2, p. 903, 2022, accessed: April 12, 2025. [Online]. Available: https://doi.org/10.3390/app12020903

[49] P. Agarwal and T. Heinis, "Motif caller: Sequence reconstruction for motif-based dna storage," *arXiv*, vol. 2412.16074, 2024, accessed: April 13, 2025. [Online]. Available: https://arxiv.org/abs/2412.16074

[50] H. Gamaarachchi, J. M. Ferguson, H. Samarakoon, K. Liyanage, and I. W. Deveson, "Simulation of nanopore sequencing signal data with tunable parameters," *Genome Research*, vol. 34, no. 5, pp. 778–783, 2024, accessed: April 13, 2025. [Online]. Available: https://genome.cshlp.org/content/34/5/778.full

[51] U. R. I. Office, "Code of practice for research: Promoting good practice and preventing misconduct," 2023, accessed: April 13, 2025. [Online]. Available: https://ukrio.org/wp-content/uploads/UKRIO-Code-of-Practice-for-Research.pdf

[52] G. Editors, "Rnn vs lstm vs gru vs transformers," 2025, accessed: April 16, 2025. [Online]. Available: https://www.geeksforgeeks.org/rnn-vs-lstm-vs-gru-vs-transformers/

[53] N. V. Otten, "Understanding elman rnn — uniqueness how to implement," 2023, accessed: Feb 27, 2025. [Online]. Available: https://spotintelligence.com/2023/02/01/elman-rnn/

[54] G. Ahammed, "Sim-to-real: Uniting simulation and reality for unprecedented results," 2024, accessed: April 26, 2025. [Online]. Available: https://nothingbutai.com/sim-to-real-bridging-the-gap-between-simulation-and-reality/

[55] J. Lee, I. Ham, Y. Kim, and H. Ko, "Time-series representation feature refinement with a learnable masking augmentation framework in contrastive learning," *Sensors*, vol. 24, no. 24, p. 7932, 2024, accessed: April 26, 2025. [Online]. Available: https://www.mdpi.com/1424-8220/24/24/7932

[56] J. L. Gehrig, D. M. Portik, M. D. Driscoll, E. Jackson, S. Chakraborty, D. Gratalo, M. Ashby, and R. Valladares, "Finding the right fit: evaluation of short-read and long-read sequencing approaches to maximize the utility of clinical microbiome data," *Microbial Genomics*, vol. 8, no. 3, p. 000794, 2022, accessed: April 26, 2025. [Online]. Available: https://pmc.ncbi.nlm.nih.gov/articles/PMC9176275/

[57] B. O. Editors, “Single-stranded dna - definition and examples,” 2022, accessed: March 25, 2025. [Online]. Available: https://www.biologyonline.com/dictionary/single-stranded-dna

Appendices

A.1 Simplified Example of DNA Encoding and Decoding Process

Goal: To encode and decode 00 11.

Encoding Process

Binary →	Motifs
00	ACGT
01	TGCA
10	CAGT
11	GTAC

Table A1: Predetermined binary and payload motif mapping.

Binary →	Motifs
00	ACGT
01	TGCA
10	CAGT
11	GTAC

Table A2: Highlighted 00 and 11 binary values and their corresponding payload motif mappings.

Due to biological constraints, there is a need for a spacer motif which will sandwich our original payload motifs.

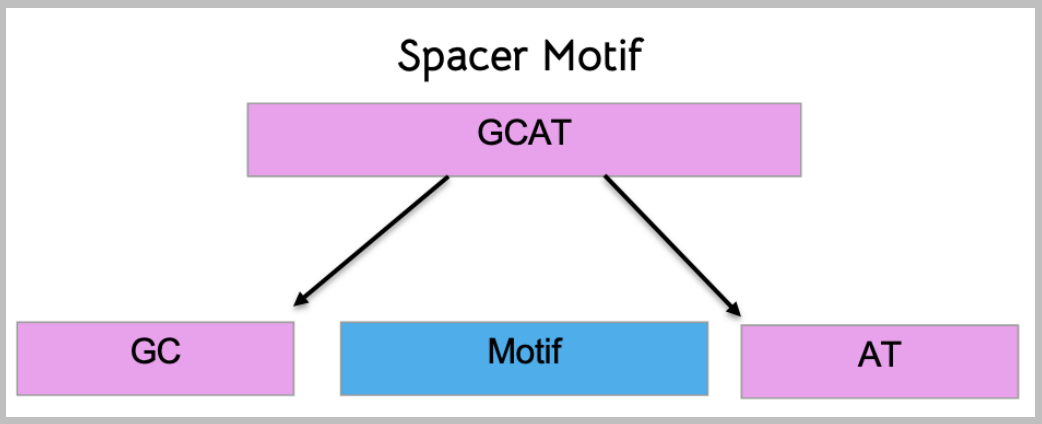


Figure A1: Fixed spacer motif being split into half to sandwich payload motifs.

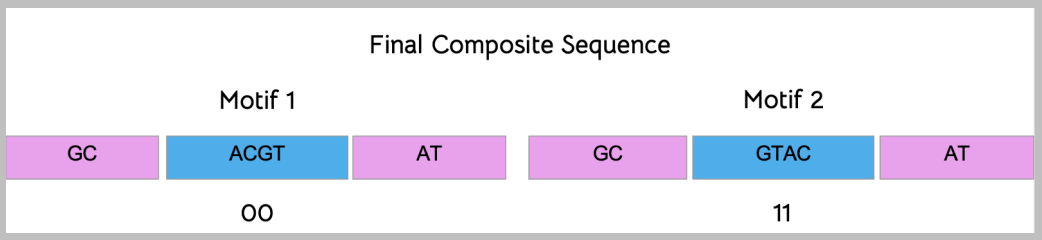


Figure A2: Final composite sequence created after sandwiching the payload motifs of 00 and 11 between GC and AT which make up the spacer motif.

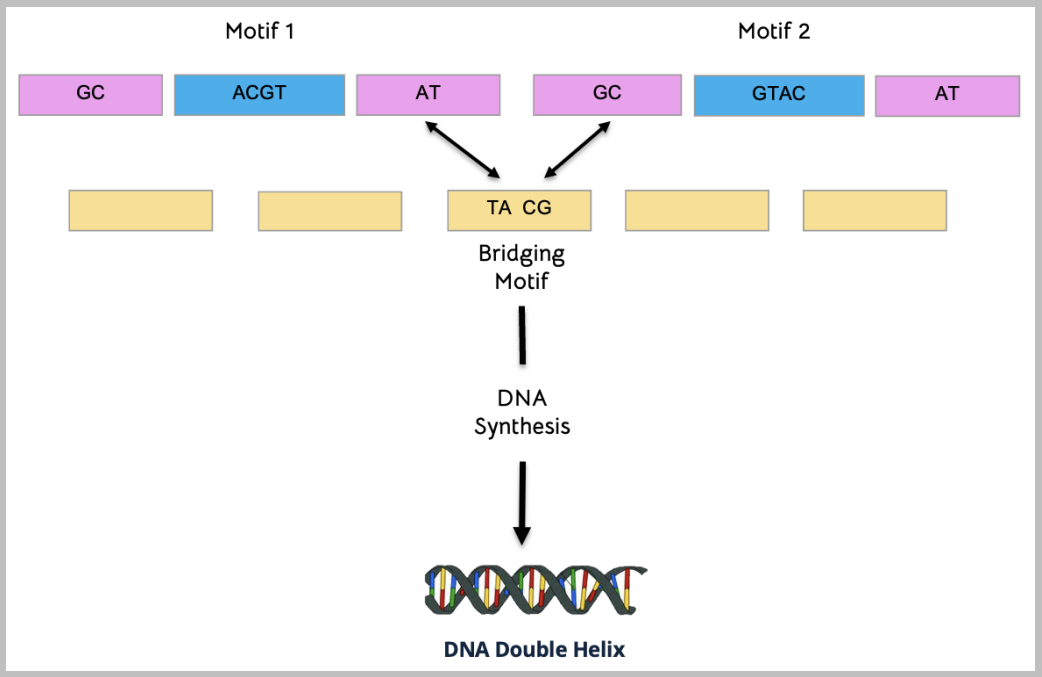


Figure A3: Each composite sequence has its own corresponding bridging sequence made up of bridging motifs which are base pairs of the composite sequence. These two sequences go through DNA synthesis to create the DNA double helix [57].

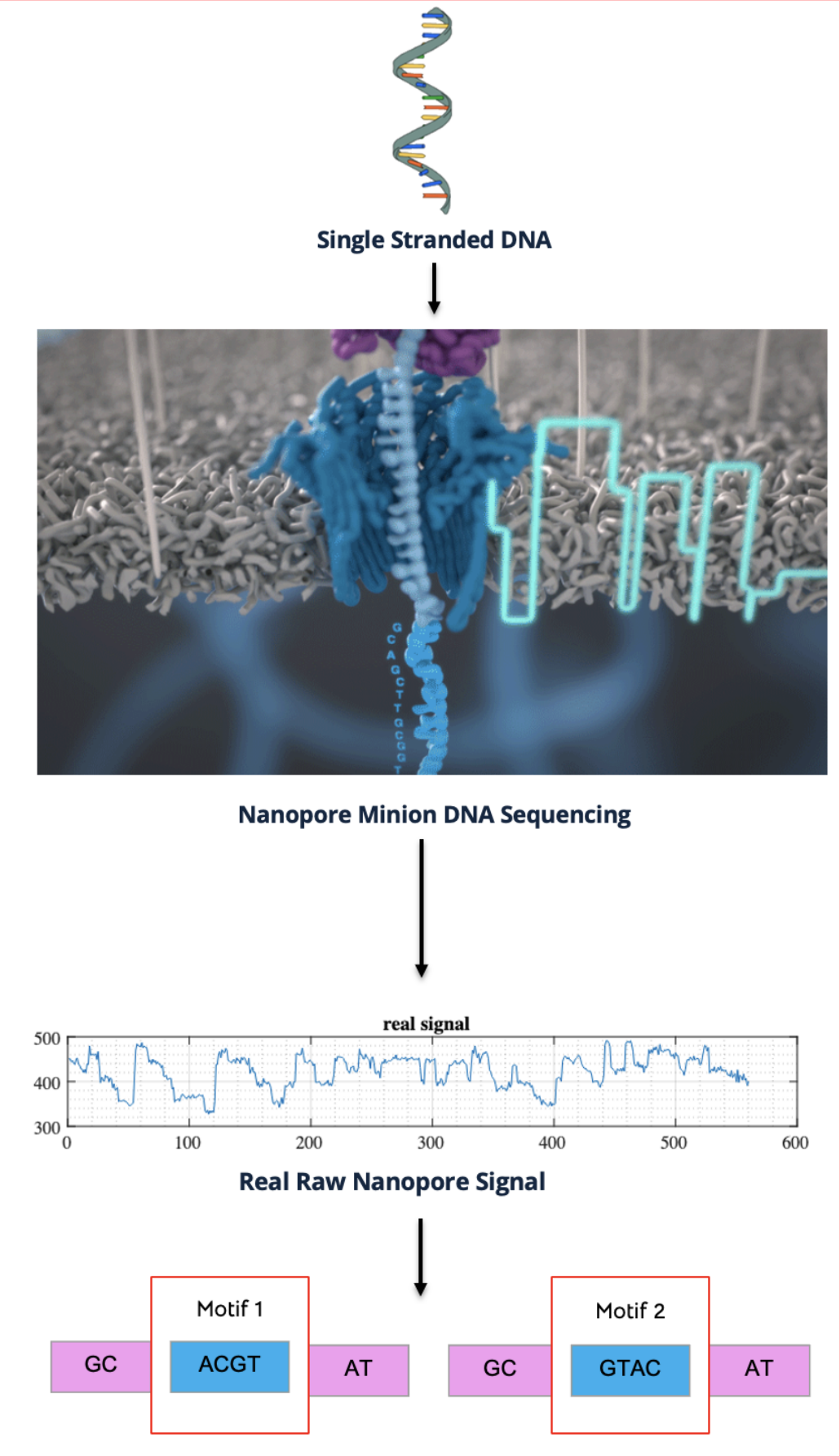


Figure A4: As the payload motifs are only contained in the original composite sequence, this is the sequence passing through the nanopore sequencer to create raw nanopore signals [57] [29] [46]. The nanopore signal is then decoded to the specific payload motifs given in red boxes (and can then be mapped back to their binary correspondences of 00 and 11).

Notable Simplifications:

- **No Error Correction:** DNA synthesis and sequencing can result in errors [22].
- **Unique Spacer Motif:** Typically, spacer motifs vary and use a preset low number of motifs, as this variety is needed to adhere to biological constraints.
- **Simplified Binary to Motif Mapping** Due to biological constraints, typically in DNA storage a simple mapping as provided is not seen as a motif typically relies on a preceding motif and so there are more complicated mapping schemes

These simplifications were done to improve understanding and establish clearer boundaries within the problem space.

A.2 Simplified Example of Basecalling + Sequence Alignment in DNA Storage Decoding

Following from the example in Appendix A.1:

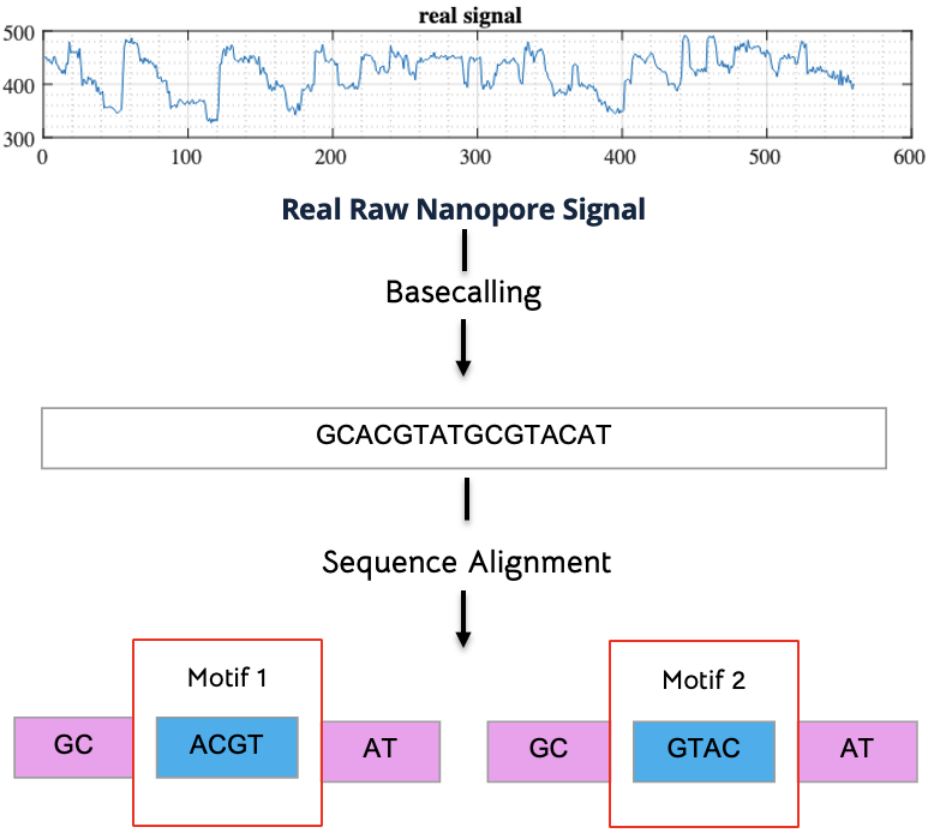


Figure A5: Raw nanopore signal converted into nucleotide bases which composes of both the spacer and payload motifs thorugh basecalling. Then the nucelotide bases are converted to the payload motifs given in red boxes using sequence alignment [46].

A.3 Gantt Charts of Timelines and Milestones

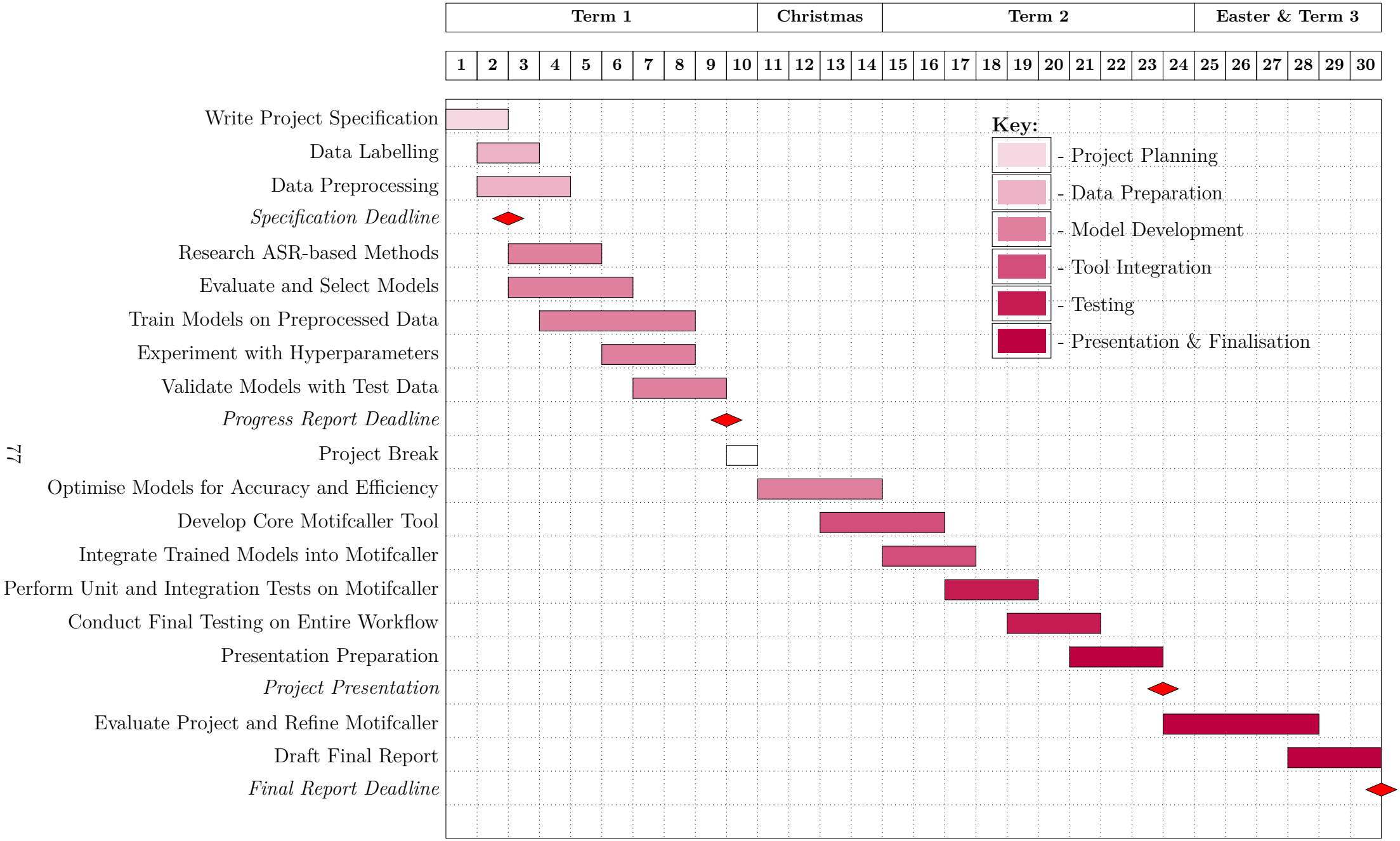


Figure A6: Project timeline broken down by academic terms and key phases. Bars shaded in increasing purple intensity denote planning, data preparation, model development, tool integration and testing, culminating in presentation and finalisation. Red triangles mark major milestones (specification deadline, progress report, presentation and final report deadlines), while the unshaded white bar in week 10 indicates the scheduled project break. (Only followed from weeks 0 to 10.)

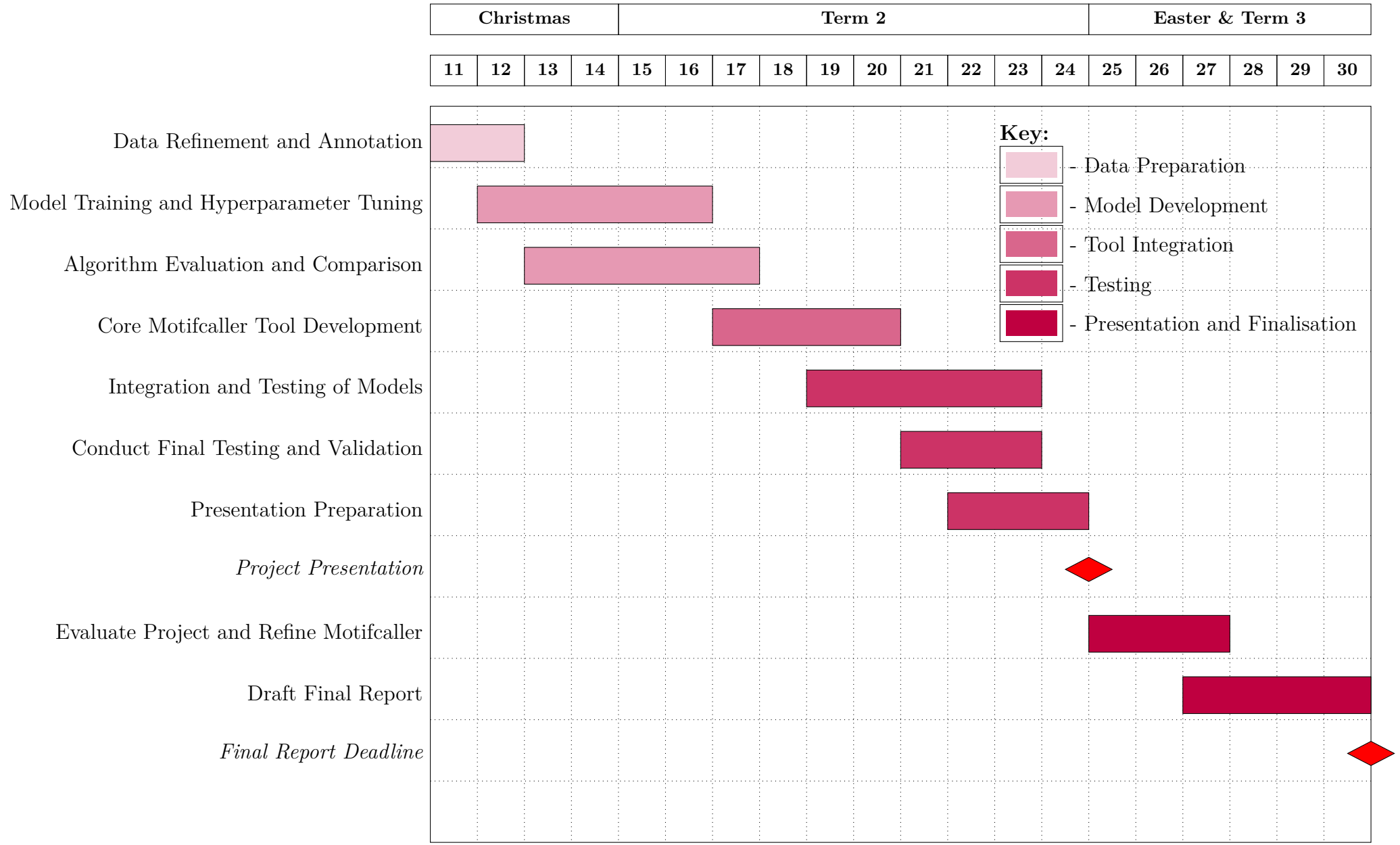


Figure A7: Revised project schedule covering weeks 11–30. Shaded bars track data preparation (light purple), model development and tuning (medium purple), core tool integration (darker purple), testing phases (deep purple) and final presentation and report writing (solid purple). Red diamonds mark key milestones: project presentation at week 24 and the final report deadline at week 30.

A.4 Default Hyperparameters

When Optuna is not in use, the following fixed hyperparameters were applied across all models (where applicable). Models that do not use a given parameter simply ignore it.

Parameter	Value
sample_size	50 000 reads
final_d_model	256
final_nhead	4
final_num_layers	2
final_dropout	0.1
final_hidden_size	128
teacher_forcing_init	0.5
teacher_forcing_final	0.0
decay constant	50
learning_rate	1×10^{-3}
weight_decay	1×10^{-4}
batch_size	16
num_epochs	30

Table A3: Default hyperparameter values used when Optuna tuning is disabled. These values were chosen as sensible, heuristic defaults rather than the result of systematic optimisation.

A.5 Encoder and Decoder Code Snippets

This section presents code snippets for various encoder and decoder architectures used in our experiments. These implementations span combinations of convolutional, recurrent, and attention-based components. Readers interested in replicating or extending the key models may wish to focus on the following:

- **CNN_Transformer_GRU (with attention)** – Utilises `Encoder_CNNTransformer` encoder, and `Decoder_GRU` decoder with attention.
- **CNN_Transformer_CTC** – Combines `Encoder_CNNTransformer` encoder with `Decoder_CTC` decoder.
- **CNN_BiGRU_CTC** – Uses `Encoder_CNN_BiGRU` encoder and `Decoder_CTC` decoder.

All encoder and decoder variants are modular and interchangeable. Researchers exploring alternative architectures may refer to individual listings to locate specific combinations of interest.

A.5.1 Encoder Variants

```
# (1) CNN+Transformer Encoder (CNN backbone + Transformer layers)
class Encoder_CNNTransformer(nn.Module):
   def __init__(self, input_channels=1, d_model=256, nhead=4, num_layers
      =2, dropout=0.1):
      super(Encoder_CNNTransformer, self).__init__()
      self.conv1 = nn.Conv1d(in_channels=input_channels, out_channels=32,
          kernel_size=3, padding=1)
      self.bn1 = nn.BatchNorm1d(32)
      self.pool1 = nn.MaxPool1d(kernel_size=2, stride=2)
      self.conv2 = nn.Conv1d(in_channels=32, out_channels=d_model,
         kernel_size=3, padding=1)
      self.bn2 = nn.BatchNorm1d(d_model)
      self.pool2 = nn.MaxPool1d(kernel_size=2, stride=2)
      self.pos_enc = PositionalEncoding(d_model, dropout=dropout, max_len
         =5000)
      encoder_layer = nn.TransformerEncoderLayer(d_model=d_model, nhead=
         nhead, dropout=dropout)
      self.transformer_encoder = nn.TransformerEncoder(encoder_layer,
         num_layers=num_layers)

   def forward(self, x):
      # x: (batch, 1, T)
      x = self.conv1(x)
      x = self.bn1(x)
      x = F.relu(x)
      x = self.pool1(x)
      x = self.conv2(x)
      x = self.bn2(x)
      x = F.relu(x)
      x = self.pool2(x) # (batch, d_model, T_reduced)
      x = x.permute(2, 0, 1) # (T_reduced, batch, d_model)
      x = self.pos_enc(x)
      x = self.transformer_encoder(x)
      return x

# (2) CNN+BiLSTM Encoder (CNN backbone + BiLSTM)
class Encoder_CNN_BiLSTM(nn.Module):
   def __init__(self, input_channels=1, d_model=256, num_layers=2,
      dropout=0.1):
      super(Encoder_CNN_BiLSTM, self).__init__()
      self.conv1 = nn.Conv1d(in_channels=input_channels, out_channels=32,
          kernel_size=3, padding=1)
      self.bn1 = nn.BatchNorm1d(32)
```

```
        self.pool1 = nn.MaxPool1d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=d_model,
            kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm1d(d_model)
        self.pool2 = nn.MaxPool1d(kernel_size=2, stride=2)
        self.bi_lstm = nn.LSTM(input_size=d_model, hidden_size=d_model//2,
                           num_layers=num_layers, batch_first=True,
                           dropout=dropout, bidirectional=True)
        self.fc = nn.Linear(d_model, d_model)
        self.pos_enc = PositionalEncoding(d_model, dropout=dropout, max_len
            =5000)

    def forward(self, x):
        # x: (batch, 1, T)
        x = self.conv1(x)
        x = self.bn1(x)
        x = F.relu(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = F.relu(x)
        x = self.pool2(x) # (batch, d_model, T_reduced)
        x = x.permute(0, 2, 1) # (batch, T_reduced, d_model)
        x, _ = self.bi_lstm(x) # (batch, T_reduced, d_model)
        x = self.fc(x)
        x = x.transpose(0, 1) # (T_reduced, batch, d_model)
        x = self.pos_enc(x)
        return x

# (3) CNN+Vanilla RNN Encoder (CNN backbone + RNN)
class Encoder_CNN_RNN(nn.Module):
    def __init__(self, input_channels=1, d_model=256, num_layers=2,
        dropout=0.1):
        super(Encoder_CNN_RNN, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=input_channels, out_channels=32,
            kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm1d(32)
        self.pool1 = nn.MaxPool1d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=d_model,
            kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm1d(d_model)
        self.pool2 = nn.MaxPool1d(kernel_size=2, stride=2)
        self.rnn = nn.RNN(input_size=d_model, hidden_size=d_model,
                      num_layers=num_layers, batch_first=True,
```

```
                    nonlinearity='tanh', dropout=dropout, bidirectional=
                        False)
        self.fc = nn.Linear(d_model, d_model)
        self.pos_enc = PositionalEncoding(d_model, dropout=dropout, max_len
            =5000)

    def forward(self, x):
        # x: (batch, 1, T)
        x = self.conv1(x)
        x = self.bn1(x)
        x = F.relu(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = F.relu(x)
        x = self.pool2(x) # (batch, d_model, T_reduced)
        x = x.permute(0, 2, 1) # (batch, T_reduced, d_model)
        x, _ = self.rnn(x) # (batch, T_reduced, d_model)
        x = self.fc(x)
        x = x.transpose(0, 1) # (T_reduced, batch, d_model)
        x = self.pos_enc(x)
        return x

# (4) CNN+Feedforward Encoder (CNN+FF)
class Encoder_CNN_FF(nn.Module):
    def __init__(self, input_channels=1, d_model=256, dropout=0.1):
        super(Encoder_CNN_FF, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=input_channels, out_channels=32,
             kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm1d(32)
        self.pool1 = nn.MaxPool1d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=d_model,
            kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm1d(d_model)
        self.pool2 = nn.MaxPool1d(kernel_size=2, stride=2)
        self.ff = nn.Sequential(
            nn.Linear(d_model, d_model),
            nn.ReLU(),
            nn.Linear(d_model, d_model)
        )
        self.pos_enc = PositionalEncoding(d_model, dropout=dropout, max_len
            =5000)

    def forward(self, x):
        # x: (batch, 1, T)
```

```
        x = self.conv1(x)
        x = self.bn1(x)
        x = F.relu(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = F.relu(x)
        x = self.pool2(x) # (batch, d_model, T_reduced)
        x = x.permute(0, 2, 1) # (batch, T_reduced, d_model)
        x = self.ff(x)    # (batch, T_reduced, d_model)
        x = x.transpose(0, 1) # (T_reduced, batch, d_model)
        x = self.pos_enc(x)
        return x

# (5) Pure Transformer Encoder (No CNN)
class Encoder_Transformer(nn.Module):
    def __init__(self, d_model=256, nhead=4, num_layers=2, dropout=0.1):
        super(Encoder_Transformer, self).__init__()
        self.input_projection = nn.Linear(1, d_model)
        self.pos_enc = PositionalEncoding(d_model, dropout=dropout, max_len
            =5000)
        encoder_layer = nn.TransformerEncoderLayer(d_model=d_model, nhead=
            nhead, dropout=dropout)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer,
            num_layers=num_layers)

    def forward(self, x):
        # x: (batch, 1, T) -> (batch, T, 1)
        x = x.squeeze(1).unsqueeze(2)
        x = self.input_projection(x)
        x = x.transpose(0, 1)
        x = self.pos_enc(x)
        x = self.transformer_encoder(x)
        return x

# (6) Raw Input Encoder (Minimal Processing)
class Encoder_Raw(nn.Module):
    def __init__(self, d_model=256, dropout=0.1, max_len=5000):
        super(Encoder_Raw, self).__init__()
        self.input_projection = nn.Linear(1, d_model)
        self.pos_enc = PositionalEncoding(d_model, dropout=dropout, max_len
            =max_len)

    def forward(self, x):
        # x: (batch, 1, T) -> (batch, T, 1)
```

```
        x = x.squeeze(1).unsqueeze(2)
        x = self.input_projection(x)
        x = x.transpose(0, 1)
        x = self.pos_enc(x)
        return x

# (7) Pure BiLSTM Encoder (No CNN)
class Encoder_PureBiLSTM(nn.Module):
    def __init__(self, d_model=256, num_layers=2, dropout=0.1):
        super(Encoder_PureBiLSTM, self).__init__()
        self.bi_lstm = nn.LSTM(input_size=1, hidden_size=d_model//2,
            num_layers=num_layers,
                        batch_first=True, dropout=dropout, bidirectional=
                            True)
        self.fc = nn.Linear(d_model, d_model)
        self.pos_enc = PositionalEncoding(d_model, dropout=dropout, max_len
            =5000)

    def forward(self, x):
        # x: (batch, 1, T) -> (batch, T, 1)
        x = x.squeeze(1).unsqueeze(-1)
        output, _ = self.bi_lstm(x)
        x = self.fc(output)
        x = x.transpose(0, 1)
        x = self.pos_enc(x)
        return x

# (8) CNN+BiGRU Encoder (CNN backbone + bidirectional GRU)
class Encoder_CNN_BiGRU(nn.Module):
    def __init__(self, input_channels=1, d_model=256, num_layers=2,
        dropout=0.1):
        super(Encoder_CNN_BiGRU, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=input_channels, out_channels=32,
            kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm1d(32)
        self.pool1 = nn.MaxPool1d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=d_model,
            kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm1d(d_model)
        self.pool2 = nn.MaxPool1d(kernel_size=2, stride=2)
        self.bi_gru = nn.GRU(input_size=d_model, hidden_size=d_model//2,
                        num_layers=num_layers, batch_first=True, dropout=
                            dropout, bidirectional=True)
        self.fc = nn.Linear(d_model, d_model)
```

```
        self.pos_enc = PositionalEncoding(d_model, dropout=dropout, max_len
            =5000)

    def forward(self, x):
        # x: (batch, 1, T)
        x = self.conv1(x)
        x = self.bn1(x)
        x = F.relu(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.bn2(x)
        x = F.relu(x)
        x = self.pool2(x) # (batch, d_model, T_reduced)
        x = x.permute(0, 2, 1) # (batch, T_reduced, d_model)
        x, _ = self.bi_gru(x) # (batch, T_reduced, d_model)
        x = self.fc(x)
        x = x.transpose(0, 1) # (T_reduced, batch, d_model)
        x = self.pos_enc(x)
        return x

# (9) Pure BiGRU Encoder (No CNN)
class Encoder_PureBiGRU(nn.Module):
    def __init__(self, d_model=256, num_layers=2, dropout=0.1):
        super(Encoder_PureBiGRU, self).__init__()
        self.bi_gru = nn.GRU(input_size=1, hidden_size=d_model//2,
            num_layers=num_layers,
                        batch_first=True, dropout=dropout, bidirectional=
                            True)
        self.fc = nn.Linear(d_model, d_model)
        self.pos_enc = PositionalEncoding(d_model, dropout=dropout, max_len
            =5000)

    def forward(self, x):
        # x: (batch, 1, T) -> (batch, T, 1)
        x = x.squeeze(1).unsqueeze(-1)
        x, _ = self.bi_gru(x)
        x = self.fc(x)
        x = x.transpose(0, 1) # (T, batch, d_model)
        x = self.pos_enc(x)
        return x
```

Listing 6: Summary of the nine encoder architectures implemented in `method_3.py`: (1) CNN+Transformer, (2) CNN+BiLSTM, (3) CNN+Vanilla RNN, (4) CNN+Feedforward, (5) pure Transformer (linear projection + positional encoding), (6) Raw input linear projection, (7) pure BiLSTM, (8) CNN+BiGRU, and (9) pure BiGRU.

A.5.2 Decoder Variants

```
# (1) GRU Decoder with Attention
class Decoder_GRU(nn.Module):
   def __init__(self, vocab_size, d_model, hidden_size, dropout=0.1):
      super().__init__()
      self.embedding = nn.Embedding(vocab_size, d_model, padding_idx=
         PAD_TOKEN)
      self.gru   = nn.GRU(d_model*2, hidden_size, batch_first=True,
         dropout=dropout)
      self.attn  = nn.Linear(hidden_size + d_model, 1)
      self.out   = nn.Linear(hidden_size, vocab_size)

   def forward(self, enc_out, targets=None, teacher_forcing_ratio=0.5,
      max_length=100):
      # enc_out: (T_enc, batch, d_model)  (batch, T_enc, d_model)
      enc = enc_out.permute(1,0,2)
      batch = enc.size(0)
      hidden = torch.zeros(1, batch, self.gru.hidden_size, device=enc.
         device)
      token = torch.full((batch,1), SOS_TOKEN, dtype=torch.long, device=
         enc.device)
      outputs = []

      for _ in range(max_length):
         emb = self.embedding(token)
         # attention scores over encoder time steps
         h_last = hidden[-1].unsqueeze(1)
         scores = F.softmax(self.attn(torch.cat([h_last.repeat(1,enc.size
            (1),1), enc], dim=2)).squeeze(-1), dim=1)
         context = torch.bmm(scores.unsqueeze(1), enc)
         rnn_in = torch.cat([emb, context], dim=2)
         out, hidden = self.gru(rnn_in, hidden)
         pred = self.out(out.squeeze(1))
         outputs.append(pred.unsqueeze(1))
         token = (targets[:,0:1] if targets is not None and torch.rand(1)
            <teacher_forcing_ratio
                else pred.argmax(-1,keepdim=True))

      return torch.cat(outputs, dim=1)

# (2) LSTM Decoder with Attention
class Decoder_LSTM(nn.Module):
   def __init__(self, vocab_size, d_model, hidden_size, num_layers=1,
      dropout=0.1):
```

```
        super(Decoder_LSTM, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model, padding_idx=
            PAD_TOKEN)
        self.lstm = nn.LSTM(
            d_model + d_model, # input is concatenated embedding + context
                vector
            hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout if num_layers > 1 else 0
        )
        self.attn = nn.Linear(hidden_size + d_model, 1) # attention score
            over encoder outputs
        self.out = nn.Linear(hidden_size, vocab_size) # project hidden
            state to vocab logits
        self.hidden_size = hidden_size

    def forward(self, encoder_outputs, targets=None, teacher_forcing_ratio
        =0.5, max_length=100):
        # encoder_outputs: (T_enc, B, d_model)
        T_enc, batch_size, d_model = encoder_outputs.size()
        # convert to (B, T_enc, d_model) for easier batch ops
        encoder_outputs = encoder_outputs.permute(1, 0, 2)
        device = encoder_outputs.device

        # initialise LSTM hidden and cell states to zeros
        hidden = (
            torch.zeros(1, batch_size, self.hidden_size, device=device),
            torch.zeros(1, batch_size, self.hidden_size, device=device)
        )

        # start with SOS token for each sample
        input_token = torch.full((batch_size, 1), SOS_TOKEN, dtype=torch.
            long, device=device)
        outputs = []

        for t in range(max_length):
            # embed the current input token
            embedded = self.embedding(input_token) # (B, 1, d_model)

            # compute attention weights
            hidden_last = hidden[0][-1].unsqueeze(1) # (B,1,hidden)
            hidden_repeat = hidden_last.repeat(1, T_enc, 1) # (B,T_enc,
                hidden)
            attn_input = torch.cat((hidden_repeat, encoder_outputs), dim=2)
```

```
            attn_weights = F.softmax(self.attn(attn_input).squeeze(2), dim
                =1)
            context = torch.bmm(attn_weights.unsqueeze(1), encoder_outputs)
                # (B,1,d_model)

            # concatenate embedding and context, pass through LSTM
            lstm_input = torch.cat((embedded, context), dim=2) # (B,1,
                d_model*2)
            output, hidden = self.lstm(lstm_input, hidden) # output (B,1,
                hidden)

            # project to vocabulary and store
            prediction = self.out(output.squeeze(1)) # (B,vocab)
            outputs.append(prediction.unsqueeze(1)) # collect for each time
                step

            # decide next input via teacher forcing or greedy
            if targets is not None and t < targets.size(1):
                if torch.rand(1).item() < teacher_forcing_ratio:
                    input_token = targets[:, t].unsqueeze(1)
                else:
                    input_token = prediction.argmax(dim=1, keepdim=True)
            else:
                input_token = prediction.argmax(dim=1, keepdim=True)

        # concatenate all time steps: (B, max_length, vocab)
        return torch.cat(outputs, dim=1)


# (3) Vanilla RNN Decoder with Attention
class Decoder_RNN(nn.Module):
    def __init__(self, vocab_size, d_model, hidden_size, num_layers=1,
        dropout=0.1):
        super(Decoder_RNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model, padding_idx=
            PAD_TOKEN)
        self.rnn = nn.RNN(
            d_model + d_model, # input embedding + context
            hidden_size,
            num_layers=num_layers,
            batch_first=True,
            nonlinearity='tanh',
            dropout=dropout
        )
        self.attn = nn.Linear(hidden_size + d_model, 1)
```

```
        self.out = nn.Linear(hidden_size, vocab_size)
        self.hidden_size = hidden_size

    def forward(self, encoder_outputs, targets=None, teacher_forcing_ratio
        =0.5, max_length=100):
        # encoder_outputs: (T_enc, B, d_model) -> (B, T_enc, d_model)
        T_enc, batch_size, d_model = encoder_outputs.size()
        encoder_outputs = encoder_outputs.permute(1, 0, 2)
        device = encoder_outputs.device

        # initialise hidden state
        hidden = torch.zeros(1, batch_size, self.hidden_size, device=device
            )
        input_token = torch.full((batch_size, 1), SOS_TOKEN, dtype=torch.
            long, device=device)
        outputs = []

        for t in range(max_length):
            embedded = self.embedding(input_token) # (B,1,d_model)

            # attention over encoder outputs
            hidden_last = hidden[-1].unsqueeze(1)
            hidden_repeat = hidden_last.repeat(1, T_enc, 1)
            attn_input = torch.cat((hidden_repeat, encoder_outputs), dim=2)
            attn_weights = F.softmax(self.attn(attn_input).squeeze(2), dim
                =1)
            context = torch.bmm(attn_weights.unsqueeze(1), encoder_outputs)

            # RNN step
            rnn_input = torch.cat((embedded, context), dim=2)
            output, hidden = self.rnn(rnn_input, hidden)

            prediction = self.out(output.squeeze(1))
            outputs.append(prediction.unsqueeze(1))

            # next token choice
            if targets is not None and t < targets.size(1):
                if torch.rand(1).item() < teacher_forcing_ratio:
                    input_token = targets[:, t].unsqueeze(1)
                else:
                    input_token = prediction.argmax(dim=1, keepdim=True)
            else:
                input_token = prediction.argmax(dim=1, keepdim=True)

        return torch.cat(outputs, dim=1)
```

```
# (4) Transformer Decoder
class Decoder_Transformer(nn.Module):
   def __init__(self, vocab_size, d_model, num_layers=2, nhead=4, dropout
      =0.1):
      super(Decoder_Transformer, self).__init__()
      self.embedding = nn.Embedding(vocab_size, d_model, padding_idx=
         PAD_TOKEN)
      self.pos_enc = PositionalEncoding(d_model, dropout=dropout)
      decoder_layer = nn.TransformerDecoderLayer(
         d_model=d_model,
         nhead=nhead,
         dropout=dropout
      )
      self.transformer_decoder = nn.TransformerDecoder(
         decoder_layer,
         num_layers=num_layers
      )
      self.out = nn.Linear(d_model, vocab_size)

   def forward(self, encoder_outputs, targets=None, teacher_forcing_ratio
      =0.5, max_length=100):
      # encoder_outputs: (T_enc, B, d_model)
      device = encoder_outputs.device
      outputs = []

      # start with SOS token
      input_token = torch.full((encoder_outputs.size(1), 1), SOS_TOKEN,
         dtype=torch.long, device=device)
      tgt_seq = self.embedding(input_token).transpose(0,1) # (1, B,
         d_model)
      tgt_seq = self.pos_enc(tgt_seq)

      for t in range(max_length):
         # generate subsequent mask for autoregressive decoding
         tgt_mask = nn.Transformer.generate_square_subsequent_mask(t+1).
            to(device)
         dec_output = self.transformer_decoder(
            tgt_seq,
            encoder_outputs,
            tgt_mask=tgt_mask
         )
         # take last time step and project
         out_t = self.out(dec_output[-1]) # (B, vocab)
```

```
            outputs.append(out_t.unsqueeze(1))

            # choose next token
            next_token = out_t.argmax(dim=1, keepdim=True)
            if targets is not None and t < targets.size(1):
                if torch.rand(1).item() < teacher_forcing_ratio:
                    input_token = targets[:, t].unsqueeze(1)
                else:
                    input_token = next_token
            else:
                input_token = next_token

            # append new embedding
            new_emb = self.embedding(input_token).transpose(0,1)
            new_emb = self.pos_enc(new_emb)
            tgt_seq = torch.cat((tgt_seq, new_emb), dim=0)

        return torch.cat(outputs, dim=1)


# (5) Attention-only Decoder
class Decoder_AttentionOnly(nn.Module):
    def __init__(self, vocab_size, d_model, dropout=0.1):
        super(Decoder_AttentionOnly, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model, padding_idx=
            PAD_TOKEN)
        self.query = nn.Parameter(torch.zeros(1, d_model)) # learned global
             query
        self.attn = nn.Linear(d_model + d_model, 1)
        self.out = nn.Linear(d_model, vocab_size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, encoder_outputs, targets=None, teacher_forcing_ratio
        =0.5, max_length=100):
        # encoder_outputs: (T_enc, B, d_model) -> (B, T_enc, d_model)
        T_enc, batch_size, d_model = encoder_outputs.size()
        enc_out = encoder_outputs.permute(1, 0, 2)
        device = enc_out.device
        query = self.query.expand(batch_size, -1)
        outputs = []

        for _ in range(max_length):
            # compute attention weights over encoder outputs using current
                query
            query_exp = query.unsqueeze(1).expand(-1, T_enc, -1)
```

```
            attn_input = torch.cat((query_exp, enc_out), dim=2)
            attn_weights = F.softmax(self.attn(attn_input).squeeze(2), dim
                =1)
            context = torch.bmm(attn_weights.unsqueeze(1), enc_out).squeeze
                (1)

            # project context directly to vocab
            output = self.out(self.dropout(context))
            outputs.append(output.unsqueeze(1))

            # update query embedding for next step
            next_token = output.argmax(dim=1, keepdim=True)
            query = 0.5 * query + 0.5 * self.embedding(next_token).squeeze
                (1)

        return torch.cat(outputs, dim=1)


# (6) Hybrid Decoder (GRU + LSTM)
class Decoder_Hybrid(nn.Module):
    def __init__(self, vocab_size, d_model, hidden_size, num_layers=1,
        dropout=0.1):
        super(Decoder_Hybrid, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model, padding_idx=
            PAD_TOKEN)
        self.gru = nn.GRU( d_model + d_model, hidden_size, num_layers=
            num_layers, batch_first=True, dropout=dropout if num_layers > 1
            else 0 )
        self.lstm = nn.LSTM(d_model + d_model, hidden_size, num_layers=
            num_layers, batch_first=True, dropout=dropout if num_layers > 1
            else 0)
        self.attn = nn.Linear(hidden_size + d_model, 1)
        self.out = nn.Linear(hidden_size, vocab_size)
        self.hidden_size = hidden_size

    def forward(self, encoder_outputs, targets=None, teacher_forcing_ratio
        =0.5, max_length=100):
        # encoder_outputs: (T_enc, B, d_model) -> (B, T_enc, d_model)
        enc = encoder_outputs.permute(1, 0, 2)
        device = enc.device
        hidden_gru = torch.zeros(1, enc.size(0), self.hidden_size, device=
            device)
        hidden_lstm = (
            torch.zeros(1, enc.size(0), self.hidden_size, device=device),
            torch.zeros(1, enc.size(0), self.hidden_size, device=device)
```

```
        )
        input_token = torch.full((enc.size(0), 1), SOS_TOKEN, dtype=torch.
            long, device=device)
        outputs = []

        for t in range(max_length):
            embedded = self.embedding(input_token)
            last_gru = hidden_gru[-1].unsqueeze(1).repeat(1, enc.size(1), 1)
            attn_input = torch.cat((last_gru, enc), dim=2)
            attn_weights = F.softmax(self.attn(attn_input).squeeze(2), dim
                =1)
            context = torch.bmm(attn_weights.unsqueeze(1), enc)

            # run both GRU and LSTM on same input+context
            rnn_input = torch.cat((embedded, context), dim=2)
            out_gru, hidden_gru = self.gru(rnn_input, hidden_gru)
            out_lstm, hidden_lstm = self.lstm(rnn_input, hidden_lstm)

            # average their outputs
            combined = (out_gru + out_lstm) / 2.0
            prediction = self.out(combined.squeeze(1))
            outputs.append(prediction.unsqueeze(1))

            # next input
            if targets is not None and t < targets.size(1) and torch.rand(1)
                .item() < teacher_forcing_ratio:
                input_token = targets[:, t].unsqueeze(1)
            else:
                input_token = prediction.argmax(dim=1, keepdim=True)

        return torch.cat(outputs, dim=1)


# (7) Conformer Decoder (Simplified)
class Decoder_Conformer(nn.Module):
    def __init__(self, vocab_size, d_model, num_layers=2, nhead=4, dropout
        =0.1):
        super(Decoder_Conformer, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model, padding_idx=
            PAD_TOKEN)
        self.pos_enc = PositionalEncoding(d_model, dropout=dropout)
        self.conv_block = nn.Sequential(
            nn.Conv1d(d_model, d_model, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv1d(d_model, d_model, kernel_size=3, padding=1),
```

```
            nn.ReLU()
        )
        decoder_layer = nn.TransformerDecoderLayer(d_model=d_model, nhead=
            nhead, dropout=dropout)
        self.transformer_decoder = nn.TransformerDecoder(decoder_layer,
            num_layers=num_layers)
        self.out = nn.Linear(d_model, vocab_size)

    def forward(self, encoder_outputs, targets=None, teacher_forcing_ratio
        =0.5, max_length=100):
        # encoder_outputs: (T_enc, B, d_model)
        device = encoder_outputs.device
        outputs = []
        input_token = torch.full((encoder_outputs.size(1), 1), SOS_TOKEN,
            dtype=torch.long, device=device)
        tgt_seq = self.embedding(input_token).transpose(0,1) # (1, B,
            d_model)
        tgt_seq = self.pos_enc(tgt_seq)

        for t in range(max_length):
            tgt_mask = nn.Transformer.generate_square_subsequent_mask(t+1).
                to(device)
            dec_output = self.transformer_decoder(tgt_seq, encoder_outputs,
                tgt_mask=tgt_mask)

            # apply convolutional block on decoder output
            conv_in = dec_output.transpose(0,1).transpose(1,2)
            conv_out = self.conv_block(conv_in).mean(dim=2)
            prediction = self.out(conv_out) # (B, vocab)
            outputs.append(prediction.unsqueeze(1))

            # next token
            next_token = prediction.argmax(dim=1, keepdim=True)
            if targets is not None and t < targets.size(1) and torch.rand(1)
                .item() < teacher_forcing_ratio:
                input_token = targets[:, t].unsqueeze(1)
            else:
                input_token = next_token

            new_emb = self.embedding(input_token).transpose(0,1)
            new_emb = self.pos_enc(new_emb)
            tgt_seq = torch.cat((tgt_seq, new_emb), dim=0)

        return torch.cat(outputs, dim=1)
```

```
# (8) GRU Decoder without Attention
class Decoder_GRU_NoAttn(nn.Module):
   def __init__(self, vocab_size, d_model, hidden_size, num_layers=1,
      dropout=0.1):
      super(Decoder_GRU_NoAttn, self).__init__()
      self.embedding = nn.Embedding(vocab_size, d_model, padding_idx=
         PAD_TOKEN)
      self.gru = nn.GRU(
         d_model,
         hidden_size,
         num_layers=num_layers,
         batch_first=True,
         dropout=dropout if num_layers > 1 else 0
      )
      self.out = nn.Linear(hidden_size, vocab_size)
      self.fc_init = nn.Linear(d_model, hidden_size) # map encoder final
         vector to hidden_size

   def forward(self, encoder_outputs, targets=None, teacher_forcing_ratio
      =0.5, max_length=100):
      # encoder_outputs: (T_enc, B, d_model)
      context = encoder_outputs[-1] # final time step as summary (B,
         d_model)
      hidden = self.fc_init(context).unsqueeze(0) # initial hidden (1, B,
          hidden_size)
      input_token = torch.full((context.size(0), 1), SOS_TOKEN, dtype=
         torch.long, device=context.device)
      outputs = []

      for t in range(max_length):
         embedded = self.embedding(input_token) # (B,1,d_model)
         out, hidden = self.gru(embedded, hidden) # (B,1,hidden)
         prediction = self.out(out.squeeze(1)) # (B,vocab)
         outputs.append(prediction.unsqueeze(1))

         if targets is not None and t < targets.size(1) and torch.rand(1)
            .item() < teacher_forcing_ratio:
            input_token = targets[:, t].unsqueeze(1)
         else:
            input_token = prediction.argmax(dim=1, keepdim=True)

      return torch.cat(outputs, dim=1)
```

```
# (9) LSTM Decoder without Attention
class Decoder_LSTM_NoAttn(nn.Module):
   def __init__(self, vocab_size, d_model, hidden_size, num_layers=1,
      dropout=0.1):
      super(Decoder_LSTM_NoAttn, self).__init__()
      self.embedding = nn.Embedding(vocab_size, d_model, padding_idx=
         PAD_TOKEN)
      self.lstm = nn.LSTM(
         d_model,
         hidden_size,
         num_layers=num_layers,
         batch_first=True,
         dropout=dropout if num_layers > 1 else 0
      )
      self.out = nn.Linear(hidden_size, vocab_size)
      self.fc_init = nn.Linear(d_model, hidden_size)

   def forward(self, encoder_outputs, targets=None, teacher_forcing_ratio
      =0.5, max_length=100):
      # encoder_outputs: (T_enc, B, d_model)
      context = encoder_outputs[-1] # (B, d_model)
      # initialise hidden and cell from context
      hidden = (
         self.fc_init(context).unsqueeze(0),
         self.fc_init(context).unsqueeze(0)
      )
      input_token = torch.full((context.size(0), 1), SOS_TOKEN, dtype=
         torch.long, device=context.device)
      outputs = []

      for t in range(max_length):
         embedded = self.embedding(input_token)
         out, hidden = self.lstm(embedded, hidden)
         prediction = self.out(out.squeeze(1))
         outputs.append(prediction.unsqueeze(1))

         if targets is not None and t < targets.size(1) and torch.rand(1)
            .item() < teacher_forcing_ratio:
            input_token = targets[:, t].unsqueeze(1)
         else:
            input_token = prediction.argmax(dim=1, keepdim=True)

      return torch.cat(outputs, dim=1)
```

```
# (10) CTC Decoder
class Decoder_CTC(nn.Module):
   def __init__(self, vocab_size, d_model, dropout=0.1):
      super(Decoder_CTC, self).__init__()
      # direct linear projection from encoder features to vocab logits
      self.out = nn.Linear(d_model, vocab_size)

   def forward(self, encoder_outputs, targets=None, teacher_forcing_ratio
      =0.0, max_length=None):
      # encoder_outputs: (T_enc, B, d_model)
      logits = self.out(encoder_outputs) # (T_enc, B, vocab)
      return logits.transpose(0,1)    # (B, T_enc, vocab)


# (11) CRF Decoder
class Decoder_CRF(nn.Module):
   def __init__(self, vocab_size, d_model, dropout=0.1):
      super().__init__()
      self.hidden2tag = nn.Linear(d_model, vocab_size)
      self.dropout = nn.Dropout(dropout)
      self.crf = CRF(vocab_size, batch_first=True)

   def forward(self, encoder_outputs, targets=None, mask=None, **_ignored
      ):
      # encoder_outputs: (T_enc, B, d_model)
      emissions = self.hidden2tag(self.dropout(encoder_outputs)).permute
         (1, 0, 2)
      # if targets provided, return negative log-likelihood for training
      if targets is not None:
         return -self.crf(emissions, targets, mask=mask)
      # otherwise decode best tag sequence for inference
      best_paths = self.crf.decode(emissions, mask=mask)
      max_len = emissions.size(1)
      # pad decoded paths to max_len
      padded = [seq + [PAD_TOKEN]*(max_len-len(seq)) for seq in
         best_paths]
      return torch.tensor(padded, device=emissions.device) # (B, T_enc)
```

Listing 7: Overview of the eleven decoder variants implemented in `method_4.py`: (1) GRU with attention, (2) LSTM with attention, (3) vanilla RNN with attention, (4) Transformer decoder, (5) attention-only module, (6) hybrid GRU+LSTM, (7) Conformer block, (8) GRU without attention, (9) LSTM without attention, (10) CTC linear projection decoder, and (11) CRF sequence decoder.

A.6 Detailed Explanation of Key Model Structures

This section provides a comprehensive discussion of the key architectural structures used in *Motifcaller*.

A.6.1 Convolutional Front-End for Signal Processing

The majority of encoder architectures within *Motifcaller* begin with a convolutional front-end consisting of two sequential 1D convolutional layers, each followed by batch normalisation, a ReLU activation, and max pooling. This design serves multiple functions crucial for processing raw squiggle signals. First, the convolutional layers act as learned filters that detect low-level features and patterns in short spans of the input signal. This is particularly important given the noisy, variable-length nature of squiggle data, which lacks clear-cut boundaries between information units. Second, the inclusion of pooling layers progressively reduces the temporal resolution of the input signal, compressing it into more manageable lengths and focusing the model's capacity on higher-level representations. Importantly, this architectural choice aligns with the nature of motifs, which span longer segments than individual nucleotide bases. Therefore, downsampling is not only computationally efficient but also conceptually consistent with motif-scale decoding.

A.6.2 Positional Encoding to Restore Sequence Awareness

Because convolution and pooling discard absolute temporal positions, positional encoding is introduced before feeding the sequence into attention-based or recurrent components. The sinusoidal positional encoding allows the model to infer both absolute and relative positions of signal features, which is essential when motifs are identified by their local shape and also by their context within the entire squiggle sequence. This ensures that the downstream model can distinguish between the same local signal pattern appearing in different temporal locations, which may correspond to different motifs or non-motif noise.

A.6.3 Transformer Encoder for Long-Range Dependency Modelling

Several encoder variants employ Transformer layers following the convolutional front-end, most notably in the `CNN_Transformer_GRU` and `CNN_Transformer_CTC` architectures. The Transformer encoder uses multi-head self-attention mechanisms to model dependencies across all positions in the input simultaneously. This is a critical feature when decoding motifs, as certain motifs may share prefixes or contain similar substructures that are only distinguishable when the model considers global signal context. Additionally, Transformers are inherently parallel in their computations, which dramatically improves training and inference speed, an important consideration in the context of large-scale DNA storage systems.

A.6.4 Bidirectional GRUs for Efficient Sequence Encoding

For models such as `CNN_BiGRU_CTC`, BiGRUs are used in place of Transformers. GRUs offer a lightweight, efficient alternative to LSTMs while retaining the ability to model sequential

dependencies. Their bidirectional configuration enables the encoder to integrate both past and future context at each time step, which is essential for resolving ambiguous signal patterns that could belong to multiple motifs. GRUs were selected over LSTMs to reduce parameter count and computational overhead, particularly in configurations intended for faster, more resource-constrained inference.

A.6.5 Attention-Based GRU Decoder for Context-Aware Motif Generation

In the `CNN_Transformer_GRU` model, decoding is performed using a GRU equipped with an attention mechanism. The attention module computes a set of dynamic weights over the encoder outputs at each decoding step, allowing the decoder to selectively focus on the most relevant parts of the signal. This is a key advantage over fixed-alignment methods, particularly for nanopore signals where motif boundaries are not well defined. By learning to align its outputs to informative regions of the encoder sequence, the model effectively bypasses the need for heuristic alignment algorithms. This not only improves interpretability but also ensures that each motif prediction is conditioned on the correct portion of the input.

A.6.6 Connectionist Temporal Classification for Alignment-Free Decoding

In both `CNN_Transformer_CTC` and `CNN_BiGRU_CTC` variants, decoding is performed using CTC. CTC is a probabilistic framework that models all valid alignments between input and output sequences without requiring explicit alignment supervision. It introduces a blank symbol and uses dynamic programming to account over possible output paths during training. This approach is particularly advantageous for direct motif prediction, as it removes the need for alignment to a reference sequence and allows the model to produce varying length motif sequences directly from encoder outputs. In practice, this significantly reduces system complexity and inference latency, making it ideal for real-time or the high processing in DNA storage applications.

A.6.7 Architectural Justifications in the Context of Motifcaller

The key model architectures, `CNN_Transformer_GRU`, `CNN_Transformer_CTC`, and `CNN_BiGRU_CTC`, were chosen to explore different trade-offs between decoding accuracy, speed, and architectural simplicity. The `CNN_Transformer_GRU` model offers powerful context modelling capabilities by combining a globally aware encoder with a flexible, autoregressive decoder. This configuration is well suited for high-accuracy applications where model interpretability and precision are critical. In contrast, the `CNN_Transformer_CTC` variant eliminates the recurrent decoder altogether, offering a streamlined inference pipeline that is faster and more scalable, albeit with slightly lower alignment sensitivity. Finally, the `CNN_BiGRU_CTC` is the best model to balance performance and efficiency by using a simpler recurrent encoder in place of the Transformer, making it suitable for embedded or edge deployments where computational resources are limited and accuracy is highly important.