

UNIVERSITY OF ZAGREB  
FACULTY OF ELECTRICAL ENGINEERING AND  
COMPUTING

MASTER THESIS No. 2471

**DNA Nanopore Sequencing  
Basecaller**

Stanislav Pavlić

Zagreb, June 2021

## MASTER THESIS ASSIGNMENT No. 2471

Student: **Stanislav Pavlić (0036499744)**

Study: Computing

Profile: Computer Science

Mentor: prof. Mile Šikić

Title: **DNA Nanopore Sequencing Basecaller**

### Description:

Nanopore sequencing is one of the state-of-the-art sequencing technologies. Passage of DNA through a pore changes its ionic current. Due to the pores' size, there are usually five nucleotides (5-mer) in the pore, influencing the measured signal. Each 5-mer produces different signal, and this information is used for basecalling (converting the raw signal to a sequence of nucleotides). The signal is approximately rectangular. The goal of the thesis is the development of DNA nanopore sequencing basecaller using self-supervised learning and modern deep learning architectures such as those based on transformers. For evaluation of the results, use publicly available datasets. The solution should be implemented in Python with the PyTorch or similar computational library. The source code should be documented using comments and should follow the Google Python Style Guide when possible. The complete application should be hosted on GitHub under an OSI approved license.

Submission date: 28 June 2021

## DIPLOMSKI ZADATAK br. 2471

Pristupnik: **Stanislav Pavlić (0036499744)**

Studij: Računarstvo

Profil: Računarska znanost

Mentor: prof. dr. sc. Mile Šikić

Zadatak: **Metoda za pretvaranje signala dobivenog sekvenciranjem nanoporama u niz nukleotida**

### Opis zadatka:

Sekvenciranje nanoporama je jedna od vodećih tehnologija sekvenciranja danas. Prolaskom DNA kroz poru mijenja se ionska struja. Uslijed veličine pore, obično se u pori nalazi 5 nukleotida (5-torka) koji utječu na mjereni signal. Svaka 5-torka uzrokuje različit signal i ta informacija se koristi za pretvaranje sirovog signala u slijed nukleotida. Oblik signala je približno pravokutan. Cilj rada je razvoj nove metode za pretvaranje signala dobivenog sekvenciranjem nanoporama korištenjem samo-nadziranog učenja i suvremenih arhitektura dubokog učenja poput onih temeljenih na transformerima. Za evaluaciju koristiti javno dostupne skupove podataka. Rješenje je potrebno implementirati u programskom jeziku Python koristeći PyTorch ili sličnu biblioteku za matrični izračun. Izvorni kod je potrebno dokumentirati koristeći komentare i razvijati prema Google Python Style Guide kada je to moguće. Cijeli programski proizvod potrebno je postaviti na GitHub pod jednom od OSI odobrenih licenci.

Rok za predaju rada: 28. lipnja 2021.

*I would like to thank my mentor, Mile Šikić, for showing support and giving valued advice throughout my years as a student. Thank you for the opportunity to work on compelling research projects which provided me with great insights and experience.*

*I would also like to thank Dominik Stanojević for his guidance as I worked on this thesis.*

*Last but not least, I would like to thank my family for all the support and encouragement they have given me.*

# CONTENTS

<b>1. Introduction</b>	<b>1</b>
1.1. The Technology - ONT . . . . .	2
1.2. The Problem - Basecalling . . . . .	3
<b>2. Approaches to basecalling</b>	<b>5</b>
2.1. Current & Past Approaches . . . . .	5
2.1.1. Guppy Basecaller . . . . .	5
2.1.2. Bonito . . . . .	6
2.2. Our Approach . . . . .	7
2.2.1. Motivation . . . . .	7
2.2.2. Self-supervised Pre-training . . . . .	9
2.2.3. Architecture Style . . . . .	10
<b>3. Data</b>	<b>12</b>
3.1. Raw Data . . . . .	12
3.2. Training Data . . . . .	13
3.3. Used data . . . . .	14
<b>4. Methods</b>	<b>17</b>
4.1. Model Architecture & Optimization . . . . .	17
4.1.1. Connectionist Temporal Classification . . . . .	17
4.1.2. Encoder . . . . .	22
4.1.3. Decoder . . . . .	28
4.1.4. Optimization . . . . .	29
4.2. Evaluation . . . . .	30
4.3. Experiments . . . . .	32
<b>5. Implementation</b>	<b>36</b>
5.1. AttentionCall . . . . .	36

5.2. Libraries, Frameworks, Dependencies . . . . .	37
<b>6. Results</b>	<b>38</b>
6.1. Training . . . . .	38
6.2. Comparison . . . . .	39
<b>7. Conclusion</b>	<b>41</b>
<b>Bibliography</b>	<b>42</b>

# 1. Introduction

There are many situations where we would like to know what is written in the DNA of an organism. Be it new research exploring and mapping out genomes to see how the organism functions, finding similarities and relations between species, or for medicinal purposes like screening for diseases and finding an effective therapy for a certain illness. Reading DNA provides useful information that gives important insights across many areas of life sciences.

Currently, there are two main methods for sequencing DNA developed by two companies that produce long reads: Pacific Biosciences and Oxford Nanopore Technologies. The methods they developed for sequencing DNA are fundamentally different. Pacific Biosciences developed a sequencing platform based on a method called SMRT sequencing (*single-molecule, real-time*), which can be summarised as a method that uses fluorescent dyes for each of the four bases which give off signals that are observed and picked out by a detector [7]. Oxford Nanopore Technologies (*ONT*) developed a platform based on reading the current level that is influenced by the nucleotides. The data generated by the latter of these methods will be the focus of this thesis and is described in more detail in section 1.1. Once the sequencing method is covered we can take a look at the problem that exists with ONT sequencing in section 1.2 whose solution is researched in this thesis.

After that, we will go through some of the past approaches to solving this problem as well as some of the current approaches that are used today in section 2.1 to give context for the general idea and motivation for the approach presented in this thesis covered in section 2.2.

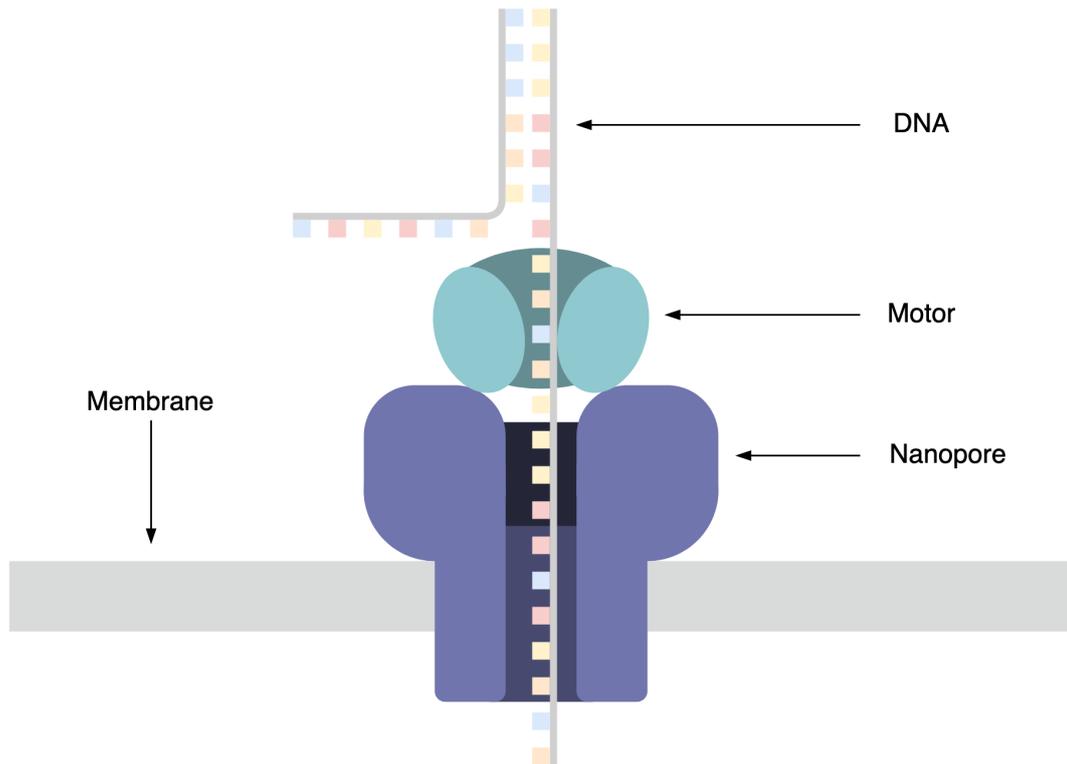
With the introduction covered, we will look into the data we are dealing with and how it is prepared for training a basecalling model. This will allow us to begin with the main part of this thesis, the model architecture, training process, evaluation, and experimentation details that we iterated through. Then, in chapter 5, we can see the final result in the form of a command-line implementation, cover the technical aspects of things described in chapter 4, and see how the implementation compares to other

selected methods in chapter 6. Finally, the thesis will provide a conclusion with a summary of important information, key takeaways, and a vision of future work to be done in this area.

## **1.1. The Technology - ONT**

Oxford Nanopore Technologies is a company based in the United Kingdom that produces devices for sequencing biological data including DNA using a technology called nanopore sequencing that measures changes in ionic current passing through the components that are in the device to infer the DNA sequence that is causing the changes in the current level. The most notable of those devices is a lightweight, portable device called MinION.

The central part of MinION is a flow cell that carries up to 2048 nanopores set in an electrically resistant membrane that ensures all current passes through the pores [15]. Each of those nanopores can be tracked individually and independently of other nanopores. A nanopore is essentially a nano-scale opening through which a DNA strand can pass and as the DNA strand passes through it, the changes in ionic current level in the pore are measured and stored. The DNA samples are prepared for sequencing by attaching a motor protein on the 5' end and ligating adapter sequences to both ends of DNA. The motor protein (or enzyme motor) attaches to the pore and then unwinds the DNA strands, and ensures the DNA moves through the pore at a steady pace to prevent bases from being skipped. The ligated adapter at the 5' end increases the chances the DNA will connect to the pore by using tethers and helps the motor protein attach to the pore. The ligated adapter at the 3' end can be used to sequence the other strand of the DNA too by forming a hairpin loop that pulls the other strand into the motor protein once the sequencing of the original strand is finished. This way we can obtain 2D reads as opposed to the usual 1D reads. The two sequences are aligned to produce a higher accuracy 2D read. The downside to using hairpin adapters is lower throughput. At any given moment, there is a single 5-mer in the pore that causes a specific electrical resistance of the channel that the pore creates in the membrane depending on which bases are contained in the 5-mer. As the DNA strand passes through the pore, the 5-mer changes and so does the electrical resistance of the channel which in turn changes the level of the ionic current passing through the pore. The produced signal has a roughly rectangular shape of varying levels. The details on how that captured data looks like are covered in chapter 3.



**Figure 1.1:** Nanopore sequencing illustration

## 1.2. The Problem - Basecalling

At  point, we know that a nanopore sequencing device outputs a signal of ionic current, but we still do not have actual an nucleotide sequence in the form of the four familiar bases, or in terms of computing - characters: A, C, G and T. This is where basecalling comes in. To get the nucleotide sequence we need to use some kind of algorithm that will process the signal and transform it into a sequence of bases, but the problem is not so straightforward.

Firstly, since DNA goes through the pore at an average of 450 bp/s (bp - base pairs) and the current is measured at a frequency of 4000 Hz, there are around 9 measurements/bp [28], but that is still only an average and the actual number varies, which complicates things. Secondly, the current levels don't stay at the same level for a single 5-mer, but they also vary themselves, which can introduce some errors. This makes it an N-to-1 sequence problem on the scale of a single base and an N-to-M sequence problem on the scale of the whole sequence, where the number of measurements N varies from case to case, with variation in the current levels for each base. Another thing to note here is the question of homopolymers, ~~it is~~ difficult to map a long sequence of measurements at roughly the same level within variation for a given base to

a specific homopolymer length due to the varying number of measurements per base, e.g. there are 63 measurements in one segment of a signal around a certain current level that corresponds to the AAAAA 5-mer, if we assume the average 9 measurements/bp we can conclude the segment maps to 11 A's ( $\frac{63}{9} = 7$  base movements and  $(5 - 1) = 4$  to complete the 5-mer), but if the average dropped to 7 measurements/bp at that moment we could conclude the segment maps to 13 A's. What we can gather from this is that the basecalling algorithm needs to be relatively complex considering how basic the data is.

## 2. Approaches to basecalling

### 2.1. Current & Past Approaches

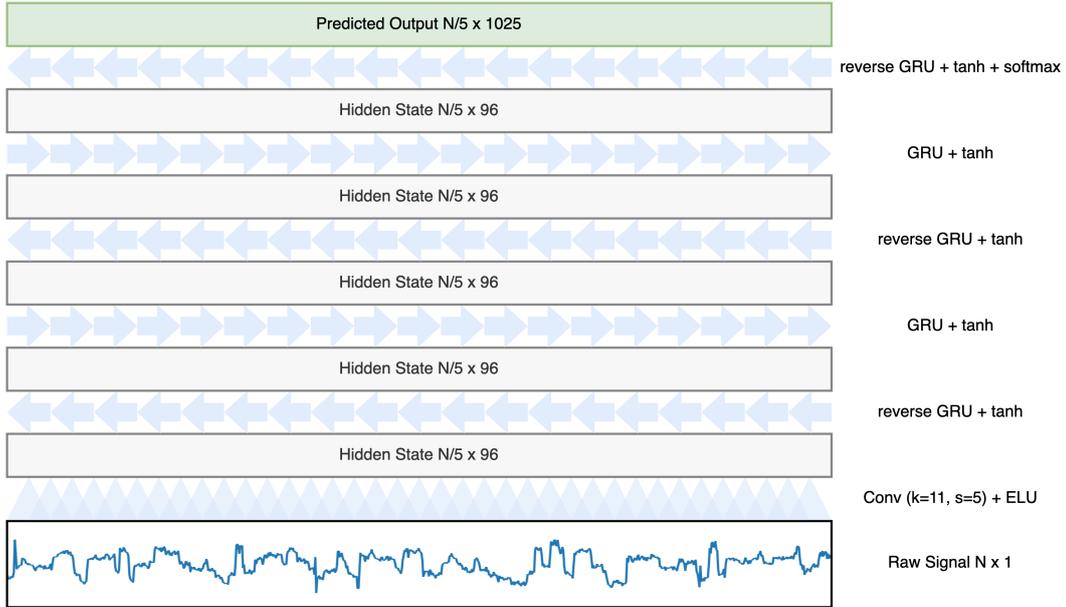
This section will provide a brief overview of current and past approaches in a level of detail that depends on how old they are, how good they are, and how close they are to our approach. The earliest version of ONT's basecaller was a Hidden Markov Model (HMM), after which they switched to RNNs which are still being used in some current models [30]. The latest type of used model is a convolutional neural network.

#### 2.1.1. Guppy Basecaller

ONT developed a data processing toolkit called Guppy that contains their basecalling algorithms. The exact algorithms and architectures they use are closed source and not publicly available, but the basic ideas are known and they will be covered in this subsection. The architecture that is used in Guppy is called RGRGR (Image 2.1), which represents the architecture itself: alternating reverse-GRU and GRU layers [35]. There are two versions of this architecture, the standard and the flip-flop version.

The standard version uses a convolutional layer as a feature encoder, followed by alternating reverse-GRU and GRU layers for a total of 5 layers (RGRGR). At the output, there is a decoder that can predict 1025 different values, 1024 for each possible 5-mer plus a special "*stay*" state.

The flip-flop version is similar, it also has a convolutional layer as a feature encoder, albeit with a wider kernel and smaller stride, and the same RGRGR component, but the output is calculated by a variation of the CTC decoder that replaces blanks with "*flip*" and "*flop*" variation for each base, so:  $\{A+, C+, G+, T+, A-, C-, G-, T-\}$ . This variation helps with calling homopolymers better [28]. Guppy achieves very good accuracy with high efficiency and speed.



**Figure 2.1:** RGRGR architecture

### 2.1.2. Bonito

Bonito [22] is part of a new generation of basecallers that moved away from RNNs. It is also open-source and publicly available. Bonito is a convolutional model based on the QuartzNet architecture [17] made for the Automatic Speech Recognition task.

QuartzNet is built upon the Jasper [19] architecture and replaces the standard 1D convolutions with time-channel separable convolutions, a kind of depthwise separable convolutions, to reduce the number of parameters and allow for much wider kernels. 1D time-channel separable convolution is a 1D depthwise separable convolution with kernel size  $K$  that is applied across  $K$  time frames and to each channel separately, followed by a pointwise convolution that is applied across all channels, but to each time frame separately. The architecture consists of repeating blocks of repeating convolutional modules, but first, it starts with a single 1D convolutional layer  $C_1$ . It is then followed by the aforementioned sequence of blocks. Each of those blocks  $B_i$  is repeated  $S_i$  times with residual connections between blocks and each of those blocks contains the same core modules that are repeated  $R_i$  and each of those modules consists of four layers in this order: depthwise convolution, pointwise convolution, batch normalization, and ReLU. After the sequence of repeating blocks, there are three convolutional layers  $C_2$ ,  $C_3$ , and  $C_4$ . At the end of the model, there is a standard CTC decoder and the model is trained with CTC loss.

Bonito adopts this architecture  for automatic speech recognition for solving the basecalling problem because it is a similar problem, we can see the analogy by

observing the data: speech is recorded in time as a signal and can be segmented into units that correspond to specific characters (depending on the approach it could also be phonemes or words) and those characters form sentences; DNA sequencing data is also recorded in time as a signal and can be segmented into units that correspond to specific bases and those bases make up DNA sequences. By experimenting with the model, they manage to achieve state-of-the-art accuracy in read and consensus-level accuracy.

## **2.2. Our Approach**

### **2.2.1. Motivation**

The advances in computational capabilities and efficient methods are allowing larger and more complex models to become feasible, boosting the capacity that the models have. There are also large amounts of data available, but much of it is not usable because it is not annotated or labeled.

#### **Amount of Data**

Currently, there are large amounts of many different types of data available to us, and more are being produced every day. Unfortunately, the vast majority of that data is not usable in the context of machine learning because it is not annotated or labeled. Having more training data available is almost always a positive thing, assuming the data is not too noisy and carries information, as it lowers the chances of overfitting by giving a better representation of the data found in practice. Most notable examples of the types of data that are available in large amounts are images, videos, and text, but it is also true for biological genomic data which comes from high-throughput sequencing and by itself can come in very large units of data — a human genome has around 3 Gbp, if you represent each base pair with a character or byte of data, you get 3 GB of data for a single human genome. Even simpler organisms, like bacteria, have genomes that range from 130 kbp [21] to 14 Mbp [11].

#### **Deep Learning Trends**

One way to utilize these large amounts of data, regardless of it being annotated or not, is an approach called self-supervised learning [34]. It combines the supervised and unsupervised approaches by training the models in a supervised manner using



unlabeled data from which task-specific "*labels*" are constructed. In this way, you can combine supervised learning which gives great results with unlabeled data which would be expensive and time-consuming to label. This approach has already proved successful in language modeling with the famous BERT [6] or in image classification with SimCLR [3].

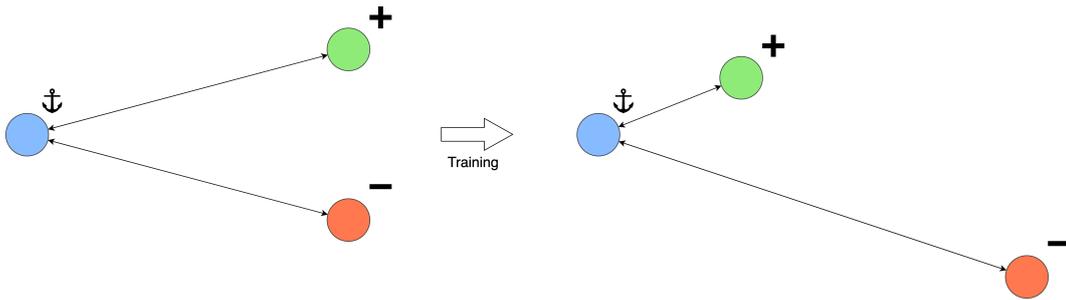
### **2.2.2. Self-supervised Pre-training**

The self-supervised learning approach on its own usually doesn't provide something very useful, but it can be used to learn better features and latent representations or embeddings which prove helpful in combination with other models, i.e. the self-supervised approach is used for pre-training models that are then fine-tuned for other downstream tasks.

One of the difficult parts of self-supervised learning is forming a pretext task for the model to solve that will make the model learn useful representations that describe the data in a general way, and not only in a way that will allow it to solve the pretext task.

For example, BERT [6] uses two pretext tasks, MLM (Masked Language Modeling) and NSP (Next Sentence Prediction). The MLM task consists of masking certain tokens of a sentence and making the model predict which token it was using the context from surrounding tokens, thus learning contextual representations of words. The NSP task consists of a simple classification task, whether a sentence follows another sentence or not, allowing the model to learn relationships between sentences, which is helpful in tasks like question answering.

Another very popular approach is contrastive representation learning which can be well illustrated on images. The core of contrastive representation learning focuses on creating embeddings in a way that the embeddings are similar for examples that are similar in the input space, and dissimilar embeddings for examples that are dissimilar in the input space. There are multiple loss functions developed for this purpose like Contrastive Loss [4], Triplet Loss [27], NCE Loss [10], and InfoNCE Loss [31]. They are all based on comparing positive samples and negative samples and measuring their distances. What the terms positive and negative denote, depends on the setting. In Contrastive Loss, positive samples are simply ones that are similar in the input space and negative samples are ones that are dissimilar in the input space, so the loss is based on minimizing the distance between latent representations of positive samples and maximizing the distance between latent representations of negative samples. In



**Figure 2.3:** Triplet Loss illustration

Triplet Loss there are three samples taken into consideration, one is an anchor, one is a positive sample and one is a negative sample. The positive sample is similar to the anchor and the negative sample is dissimilar, and so the loss is based on minimizing the distance between the anchor and the positive sample while maximizing the distance between the anchor and the negative sample, see Figure 2.3. On the other hand, InfoNCE Loss takes a context vector and a set of samples where only one is a positive sample from the target distribution, while the other samples are noise, and the task is to identify the positive sample that corresponds to the context vector among the negative samples that are unrelated to the context vector, i.e. noise.

### 2.2.3. Architecture Style

When using the self-supervised pre-training approach the architecture can be split into two core components: the encoder and the decoder. The encoder is the component that is trained during pre-training and outputs new feature embeddings. The decoder is attached to the encoder and is used to decode the features computed by the encoder.

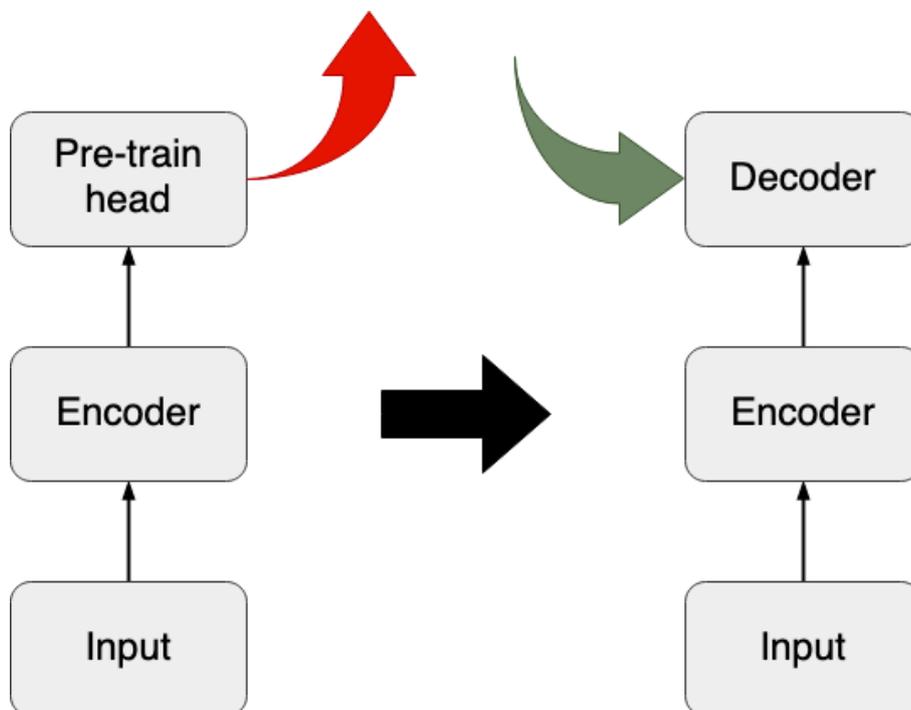
#### Encoder

During pre-training, the encoder is used as a backbone and there is a head component at the output that transforms the embeddings into a form suitable for computing the loss function, and the head is later removed for downstream tasks as it is only useful for pretext tasks. The encoder can create embeddings on different levels of the input. In images, it can be used to represent fragments of images, whole images, or a sequence of images. The level used is based on the needs in the downstream task. Different goals can be achieved with the created embeddings, it can be used to infuse them with context information of surrounding data or to compress the amount of data that is used to describe the same information. The encoder component can be frozen during fine-

tuning.

## Decoder

The decoder uses the embeddings created by the encoder to produce predictions for the downstream task. It can be as simple as a single fully-connected feedforward layer or something more complex, depending on how effective the encoder is in creating valuable features for the downstream task.



**Figure 2.4:** Left — pre-training setup, right — fine-tuning setup

## 3. Data

Nanopore sequencing DNA data consists of a large number of reads that each span some region of the genome. The lengths of the reads start from around 500 bp and can reach up to 2.2 Mbp [25], but are typically in the 10 kbp order of magnitude. The chemistry of the data used in this thesis is R9.4.1.

### 3.1. Raw Data

The format that is used for storing nanopore reads is called FAST5, which is based on the HDF5 format. The HDF5 format is a generic format for storing large amounts of data in a hierarchical structure (HDF - Hierarchical Data Format). The data is structured using two types of objects: groups and datasets. Groups are container objects that can hold datasets and other groups. Datasets are multidimensional arrays of a defined type. There are also attributes that contain metadata and can be assigned to groups and datasets. The format can be compared to a file-system organization where groups function as folders and datasets function as files.

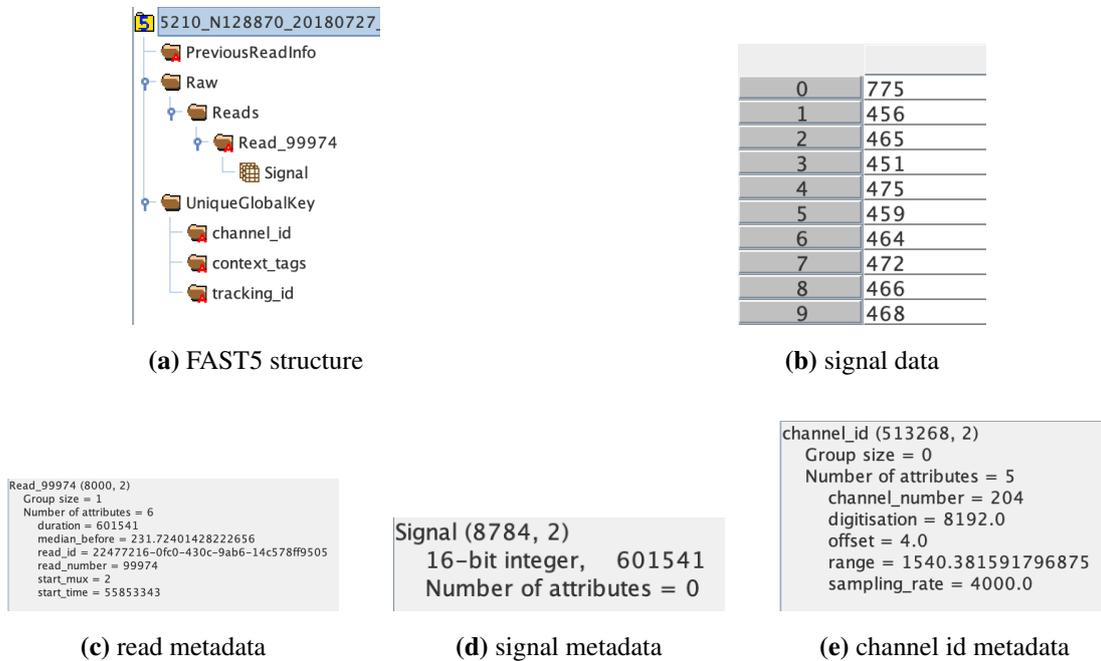
FAST5 takes these concepts and limits the format to a defined, fixed structure. There are two basic types of FAST5 files: single-read files that contain only a single read per file and multi-read files that can contain multiple reads per file. The FAST5 files can contain raw sequencing data stored in the *Raw* group, event-level data, and base-level data stored in the *Analyses* group. For basecalling uses, the *Raw* group contains the input data for basecalling and the base-level data can be added after basecalling. The raw signal data consists of electric current measurements in pA sampled at a certain rate (usually 4000 Hz), which are scaled and discretized before being stored as 16-bit integers under the *Signal* key. The reads also come with useful attributes like the duration of sequencing expressed in the number of samples, electric current median before the read, unique read ID, read number, multiplexor start settings, sequencing start time expressed in the number of samples. For each read, there are additional groups that contain more metadata about the read. The *channel\_id* group contains the follow-

ing set of attributes: *channel\_number*, *digitization*, *range*, *offset*, *sampling\_rate*. This group is important as it contains information needed to transform the signal from the discretized representation back to the continuous representation: *digitization*, *range*, and *offset*. The transformation is calculated with the following formula:

$$scale = range/digitization$$

$$current = scale \cdot (Signal + offset)[pA]$$

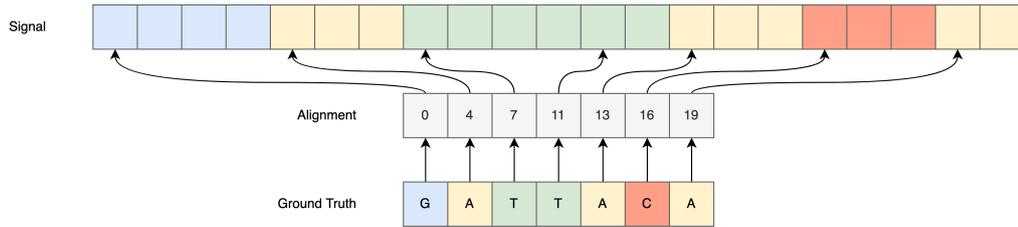
HDF5 data can be viewed with HDFView and an example of a single-read FAST5 file can be seen in Figure 3.1.



**Figure 3.1:** FAST5 structure and contents

## 3.2. Training Data

To train a basecalling model we need labels alongside the raw signal data, and to get those labels we need to basecall the raw signal because it's completely impractical to manually label the data. This means we need a basecaller to train a basecaller, making this a kind of bootstrapping process. The way to do this, as described on ONT's Taiyaki GitHub repository [23], is to use some available basecaller (they propose Guppy) to basecall the raw reads, map the basecalled reads to the reference genome and then



**Figure 3.2:** Training data structure

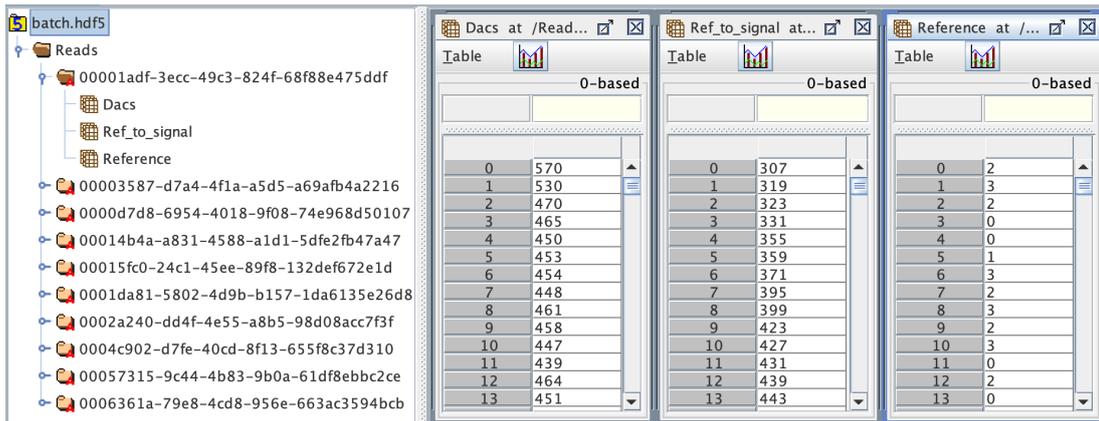
extract the ground truth for each sequence by taking the segment of the reference that the read was mapped to. This will give us labels that are consistent across all reads. The assumption here, I believe, is that either the reference was generated using those reads, or the ground truth is obtained by doing consensus with the available reads after mapping the reads to the reference genome, or maybe they decide to ignore the possible small differences between the reads and the reference. Once we have obtained the ground truth we need to align the signal and the ground truth, which is done by using a pre-trained basecaller model. Alternatively, the alignment can be obtained with the re-squiggle algorithm provided by Tombo [24].

Although the alignment is technically not needed for training basecaller models that use CTC, practically, there is a need for this because the length of the signal and the ground truth makes computations necessary for CTC more difficult due to the quadratic complexity of those computations, and by aligning the signal and the ground truth we can divide them into smaller chunks of data we can make those computations more practical to execute.

There is an already prepared HDF5 dataset available from the authors of Bonito which was used for training the model presented in this thesis. The dataset contains around 12 GB of reads, ground truths, and their alignments. The alignment is represented as an array of integers where each base from the ground truth array maps to the starting position of the alignment in the signal array indicated by the value stored at the same index in the alignment array as the index of the base in question in the ground truth array. See Figure 3.2 for a visualization of the described structure and Figure 3.3 for an example.

### 3.3. Used data

The data from Bonito was split into a training, validation, and test set, in a ratio of 80:10:10, but in the end this test set was not used. Instead, raw FAST5's with a reference genome from Ryan Wick's basecaller benchmark [35] were used for testing



(a) Batch sample from Bonito HDF5 dataset example

```

00001adf-3ecc-49c3-824f-68f88e475ddf (6592, 2)
  Group size = 3
  Number of attributes = 8
    alphabet = ACGT
    collapse_alphabet = ACGT
    digitisation = 8192.0
    offset = 4.0
    range = 1476.4886474609375
    read_id = 00001adf-3ecc-49c3-824f-68f88e475ddf
    scale_frompA = 16.0330232076
    shift_frompA = 82.367592

```

(b) Bonito dataset read metadata example

**Figure 3.3:** Training data example

purposes. Three different sets of reads were chosen from three different species: *Acinetobacter pittii*, *Serratia marcescens*, and *Haemophilus haemolyticus*. The raw FAST5s are available at [https://bridges.monash.edu/articles/dataset/Raw\\_fast5s/7676174](https://bridges.monash.edu/articles/dataset/Raw_fast5s/7676174), and the reference genomes are available at [bridges.monash.edu/articles/dataset/Reference\\_genomes/7676135](https://bridges.monash.edu/articles/dataset/Reference_genomes/7676135). Before being fed to the model, the data is preprocessed by normalizing it with the median and median absolute deviation (MAD) as seen in Bonito.

$$\begin{aligned} \text{med}_{\text{signal}} &= \text{median}(\text{signal}) \\ \text{MAD}_{\text{signal}} &= 1.4826 \cdot \text{median}(|\text{signal} - \text{med}_{\text{signal}}|) \\ \text{signal}_{\text{medMAD}} &= \frac{\text{signal} - \text{med}_{\text{signal}}}{\text{MAD}_{\text{signal}}} \end{aligned}$$

## 4. Methods

### 4.1. Model Architecture & Optimization

This section consists of a description of the entire model architecture across its components and the optimization technique used to train the model with available hyperparameters in mind. The model architecture is designed around utilizing a pre-trained encoder and using the CTC decoder and loss for fine-tuning.

#### 4.1.1. Connectionist Temporal Classification

Connectionist Temporal Classification (CTC) [9] is an algorithm for producing neural network outputs, and the algorithm is combined with a corresponding loss function for training. The goal of the method is to solve the problem of modeling sequences with a variable time component. An example of this would be ASR mentioned in subsection 2.1.2. With a fixed sampling rate, the number of data points in an audio signal that corresponds to the same phoneme may vary depending on how fast it is spoken, which is, as mentioned in that subsection, similar to basecalling data. For this reason, we decided to use CTC in our model.

CTC gives us a way to map an input sequence  $X$  to an output sequence  $Y$  where the lengths of both those sequences vary in length without aligning the sequences. It does so by extracting a probability distribution over all possible sequences  $Y$  for a given sequence  $X$ . This can then be used for inference by choosing the most likely sequence, or it can be used for scoring the model given a known target  $Y_{gt}$ .

#### Alignment

CTC works by implicitly creating alignments between the input and output sequences without having that information at the input. The best way to describe this alignment process is by visualizing these alignments. For our example, let our input be some sequence of latent variables  $X = \{x_1, x_2, x_3, \dots, x_{12}\}$ , or in a general case simply



**Figure 4.1:** CTC alignment examples

inputs, and the output a sequence of bases  $GAT$ , or for the general case tokens. To create a valid alignment we can assign a base to each element of the input and then collapse the repeating bases into a single base so that the result is a valid alignment, e.g.  $2 \times G, 7 \times A, 3 \times T$ . The big problem here is that we lack a way to represent sequences where bases actually are repeated due to the collapsing step. To counteract this, CTC introduces a new special token *blank* or  $\epsilon$  which can be used to represent no output at a time step or to separate repeating tokens that should not be collapsed. After adding  $\epsilon$ , one more step is added to decoding the output sequence from an alignment and that is removing any remaining  $\epsilon$  tokens. To summarise, an output token is assigned to each input, repeating tokens are collapsed and  $\epsilon$  tokens are removed. So if we have again 12 inputs and a sequence of bases  $GAA$  then we would need to have blanks in the alignment. For a visual representation of these examples check Figure 4.1. There are a few things to note about these alignments. They are monotonic, moving one step in the input sequence means either staying on the same step or moving one step in the output sequence. The alignment relationship between  $X$  and  $Y$  is many-to-one, multiple elements of the input sequence can be aligned to one element of the output sequence, but not the other way around. This means that the length of the input sequence  $X$  must be greater than or equal to the length of the output sequence  $Y$ .

## Loss function

The objective of CTC is to produce an output sequence  $Y$  given an input sequence  $X$  by using probabilities at each time step to compute the probability of the entire output sequence. More formally:

$$p(Y|X) = \sum_{A \in V(X,Y)} \prod_{t=1}^T p(A_t|X)$$

In other words, the conditional probability of  $Y$  given  $X$  is a marginalization over alignments  $A$  from a set of valid alignments  $V(X, Y)$  whose probabilities are computed from the product of probabilities at each time step  $t$  from the first time step  $t = 1$  to the final time step  $T$ . Therefore, the model that uses CTC needs to compute the probabilities for each output class at every time step. The output classes in basecalling are  $y \in \{\epsilon, A, C, G, T\}$ . To arrive at a loss function for CTC, the probability estimate  $p(Y|X)$  is combined with the negative log-likelihood loss. When applied to the entire dataset  $\mathcal{D}$ :

$$\mathcal{L}_{CTC} = \sum_{(X,Y) \in \mathcal{D}} -\log p(Y|X)$$

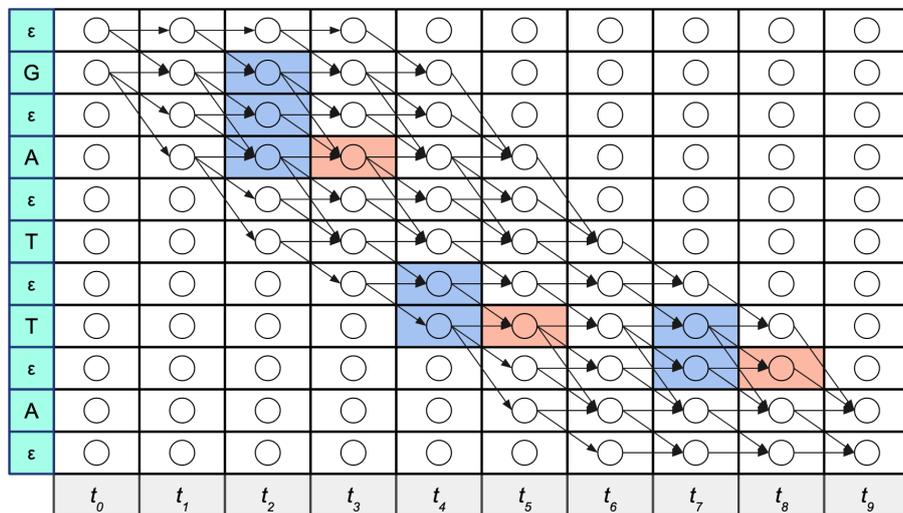
Computing the loss naively by going through all the alignments and computing the probability of each of them and then summing them up can be very expensive to compute with longer sequences due to the number of possible alignments. There is an efficient solution using dynamic programming that makes use of computations made for alignments that share the same prefix and merging the alignments that lead to the same output at the same time step. To achieve this, the output sequence  $Y = \{y_1, y_2, y_3, \dots\}$  is interspersed with  $\epsilon$ :  $Y = \{\epsilon, y_1, \epsilon, y_2, \epsilon, y_3, \epsilon, \dots, \epsilon\}$ . This is possible because the sequences are equivalent since the  $\epsilon$  are dropped after collapsing. Using this, the dynamic programming solution can be formed as a matrix of probabilities where each row corresponds to the element of  $Y$  expanded with  $\epsilon$  and each column represents a single time step like depicted in Figure 4.2, and the allowed transitions are moving monotonically right and down-right with some additional rules.

In order to generate only valid alignments some cases need to be handled. Let  $Y = \{\dots, \epsilon_{i-1}, y_i, \epsilon_{i+1}, y_{i+2}, \epsilon_{i+3}, \dots\}$ . Transitions occur one time step at a time, from  $t_j$  to  $t_{j+1}$ . Case 1: when  $y_i \neq y_{i+2}$ , it is possible to transition to state  $y_{i+2}$  from states  $y_i, \epsilon_{i+1}$  and  $y_{i+2}$ . Case 2: when  $y_i = y_{i+2}$ , it is possible to transition to state  $y_{i+2}$  from

$\epsilon$	○	○	○	○	○	○	○	○	○	○
G	○	○	○	○	○	○	○	○	○	○
$\epsilon$	○	○	○	○	○	○	○	○	○	○
A	○	○	○	○	○	○	○	○	○	○
$\epsilon$	○	○	○	○	○	○	○	○	○	○
T	○	○	○	○	○	○	○	○	○	○
$\epsilon$	○	○	○	○	○	○	○	○	○	○
T	○	○	○	○	○	○	○	○	○	○
$\epsilon$	○	○	○	○	○	○	○	○	○	○
A	○	○	○	○	○	○	○	○	○	○
$\epsilon$	○	○	○	○	○	○	○	○	○	○
	$t_0$	$t_1$	$t_2$	$t_3$	$t_4$	$t_5$	$t_6$	$t_7$	$t_8$	$t_9$

**Figure 4.2:** DP matrix for an example with the expected output GATTA and an input with 9 time steps.

states  $\epsilon_{i+1}$  and  $y_{i+2}$ , but not  $y_i$  because it would mean that one token denoted by  $y_i$  and  $y_{i+2}$  would be lost when collapsing repeats. Case 3: when transitioning to an  $\epsilon$ , like  $\epsilon_{i+1}$  here, it is possible to transition to it from  $y_i$  or  $\epsilon_{i+1}$ , but not  $\epsilon_{i-1}$  because then  $y_i$  would be skipped in the alignment. A visual of this example is in Figure 4.3.



**Figure 4.3:** DP matrix with indicated transitions and transitions of interest for mentioned cases. Case 1) ( $A, t_3$ ), Case 2) ( $T, t_5$ ), Case 3) ( $\epsilon, t_8$ )

The algorithm performed on this matrix to compute the posterior probabilities of each state at each time step in the valid alignments for the given label is the forward-backward algorithm. The algorithm uses two passes, forward and backward, to com-

pute the total probability of a token at a time step. The forward pass computes the total probability  $\alpha_t(s)$  from the first time step to time step  $t$  and token  $s$ . The backward pass computes the total probability  $\beta_t(s)$  from time step  $t + 1$  and token  $s$  to the last time step. A single path probability is the product of elements in the path and all paths are then summed for the total probability. See example in Figure 4.4 and following equations:

$$\alpha_4(1) = p(\epsilon\epsilon\epsilon\epsilon G) + p(\epsilon\epsilon\epsilon GG) + p(\epsilon\epsilon GGG) + p(\epsilon GGGG) + p(GGGGG)$$

$$\alpha_4(1) = y_0^\epsilon y_1^\epsilon y_2^\epsilon y_3^\epsilon y_4^G + y_0^\epsilon y_1^\epsilon y_2^\epsilon y_3^G y_4^G + y_0^\epsilon y_1^\epsilon y_2^G y_3^G y_4^G + y_0^\epsilon y_1^G y_2^G y_3^G y_4^G + y_0^G y_1^G y_2^G y_3^G y_4^G$$

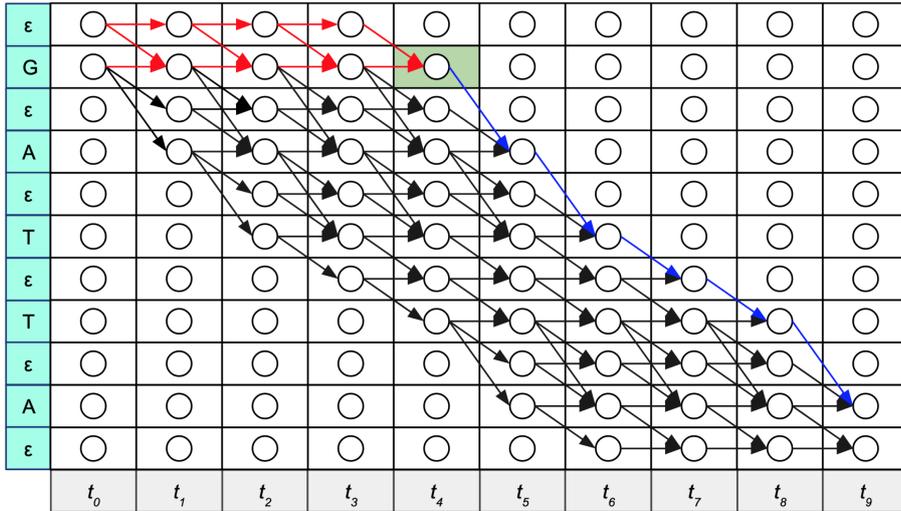
$$\beta_4(1) = p(AT\epsilon TA)$$

$$\beta_4(1) = y_5^A y_6^T y_7^\epsilon y_8^T y_9^A$$

$$\begin{aligned} \alpha_4(1)\beta_4(1) = & y_0^\epsilon y_1^\epsilon y_2^\epsilon y_3^\epsilon y_4^G y_5^A y_6^T y_7^\epsilon y_8^T y_9^A \\ & + y_0^\epsilon y_1^\epsilon y_2^\epsilon y_3^G y_4^G y_5^A y_6^T y_7^\epsilon y_8^T y_9^A \\ & + y_0^\epsilon y_1^\epsilon y_2^G y_3^G y_4^G y_5^A y_6^T y_7^\epsilon y_8^T y_9^A \\ & + y_0^\epsilon y_1^G y_2^G y_3^G y_4^G y_5^A y_6^T y_7^\epsilon y_8^T y_9^A \\ & + y_0^G y_1^G y_2^G y_3^G y_4^G y_5^A y_6^T y_7^\epsilon y_8^T y_9^A \end{aligned}$$

$$\begin{aligned} \alpha_4(1)\beta_4(1) = & p(\epsilon\epsilon\epsilon\epsilon GAT\epsilon TA) \\ & + p(\epsilon\epsilon\epsilon GGAT\epsilon TA) \\ & + p(\epsilon\epsilon GGGAT\epsilon TA) \\ & + p(\epsilon GGGGAT\epsilon TA) \\ & + p(GGGGGAT\epsilon TA) \end{aligned}$$

When doing inference, two options are available: Viterbi search and beam search. Viterbi search in this context is a greedy algorithm that simply picks the output with the highest probability at each time step, i.e.  $\text{argmax}$ . Beam search offers a prediction that is closer to the most likely output sequence by branching and keeping at most  $K$  (beam size) output sequences with the highest probability up to that point. Increasing the beam size produces a more likely output sequence, but it takes exponentially more computation with the increase of the beam size.



**Figure 4.4:** Forward-backward example for token at state  $s = 1$  which is  $G$ , and time step  $t_4$ , or  $t = 4$ , giving the total probability of all paths going through  $G$  at  $t = 4$ .

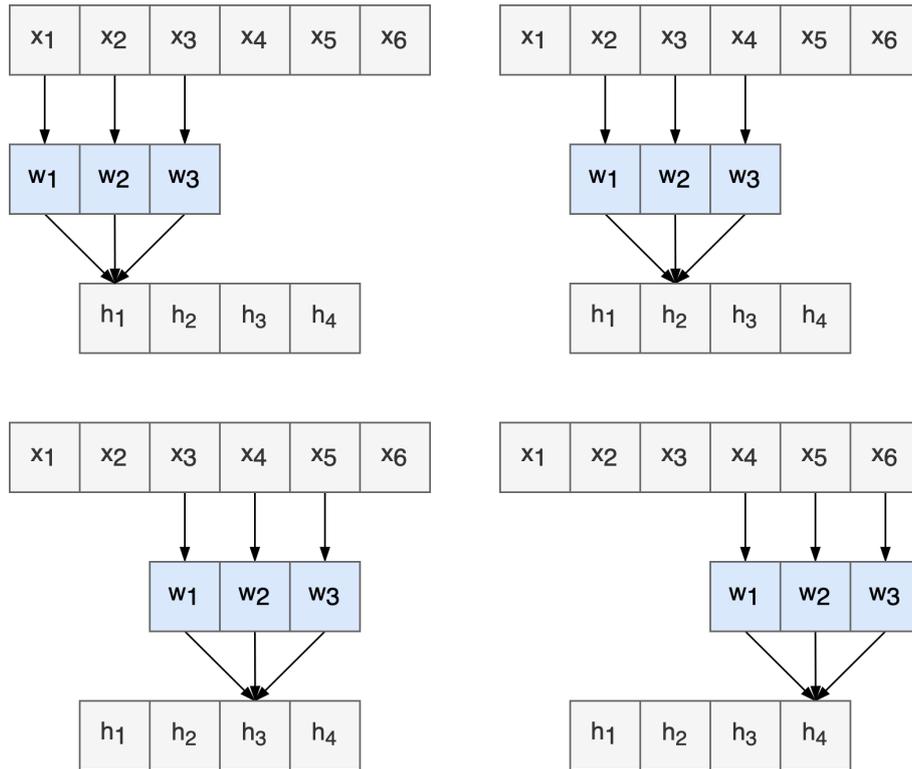
### 4.1.2. Encoder

Before starting work on this basecaller model, we worked on a self-supervised model for nanopore sequencing data with the idea that it will be used for different use-cases that operate on nanopore data, among which is basecalling. The encoder component architecture of the basecaller, therefore, follows what we did then, but can be swapped out for a different one since we did not finish the self-supervised mode before starting work on this basecaller.

#### Convolution

Since the input data is a 1D array of electric current values, we need to encode this data into vectors. For this, we decided to use the 1D convolutional layers which are a standard solution for encoding time-series data.

Convolutional layers typically operate on 2-dimensional data and are made up of learnable parameters grouped into kernels or filters, 4-dimensional tensors whose dimensions correspond to the number of output channels  $C_{in}$ , kernel width  $K_W$ , kernel height  $K_H$ , and the number of input channels  $C_{out}$ . In 1D convolutional layers the input is 1-dimensional (the dimension often being called the time or temporal dimension) and so the kernels are 3-dimensional tensors with dimension  $C_{in}$ ,  $K$  and  $C_{out}$ . The layer is applied on a part of the input in the time dimension with a receptive field  $K$ , but through all the input channels, thus acting locally in time. When applied, the kernel is convolved (actually cross-correlated since the kernel is not flipped) with the

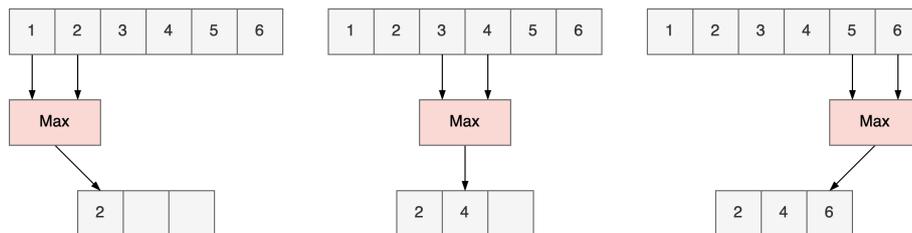


**Figure 4.5:** 1D convolution with  $C_{in} = 1$ ,  $K = 3$ ,  $S = 1$ ,  $C_{out} = 1$ . The elements of  $H$  are the result of convolving the input  $X$  and the kernel  $W$ . Each element of  $H$  is computed as a dot product,  $h_1 = x_1w_1 + x_2w_2 + x_3w_3$ .

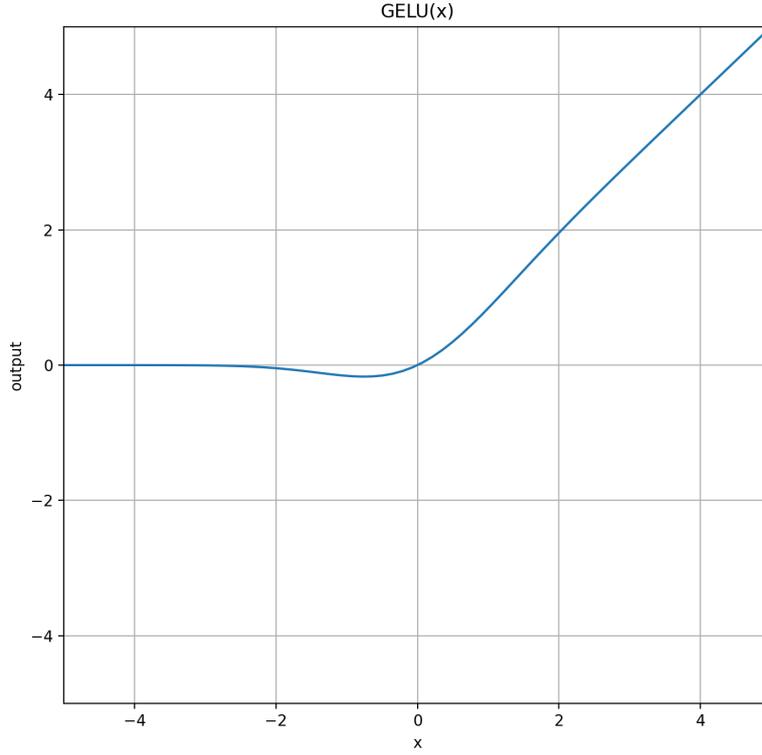
input by computing the dot product between the kernel and the input as it slides with stride  $S$  over the input's temporal dimension to cover all the positions. When training, the network can learn kernels that activate when they detect some pattern that is useful for producing desirable outputs.

The pooling layer is a layer without parameters that summarises local data by using some reduction function like the maximum or the average and sliding over the input just like the convolutional kernel.

The convolutional encoder is here referred to as the feature encoder. The feature encoder architecture is designed to model increasingly complex features by increasing



**Figure 4.6:** 1D max pool with  $K = 2$ ,  $S = 2$  on  $C_{in} = 1$ ,  $C_{out} = 1$ .

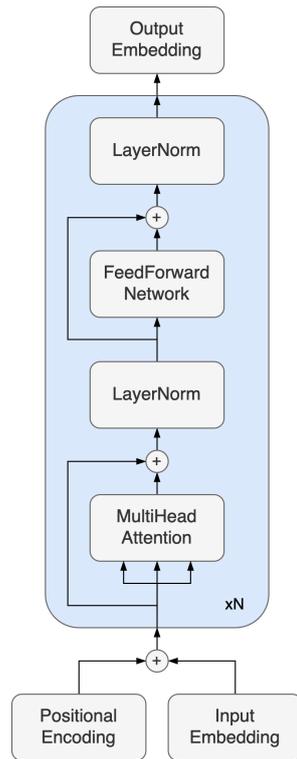


**Figure 4.7:** GELU activation function  $\text{GELU}(x) = x \cdot \Phi(x)$  where  $\Phi(x)$  is a Gaussian Cumulative Distribution Function (CDF).

the channel size  $C_{out}$  and downsample the time domain using pooling layers to decrease the complexity of the next steps while controlling the receptive field with kernel size  $K$  and stride  $S$  hyperparameters. The feature encoder consists of convolutional layers and pooling layers with padding, batch normalization [13], and GELU activation [12], and is described in Table 4.1.

**Table 4.1:** Convolutional feature encoder architecture

Layer #	$K_{conv}$	$S_{conv}$	$C_{out}$	$K_{maxpool}$	$S_{maxpool}$	BatchNorm	Activation
1	3	1	64	2	2	Yes	GELU
2	3	1	128	2	2	Yes	GELU
3	3	1	256	2	2	Yes	GELU
4	3	1	512	1	2	Yes	GELU



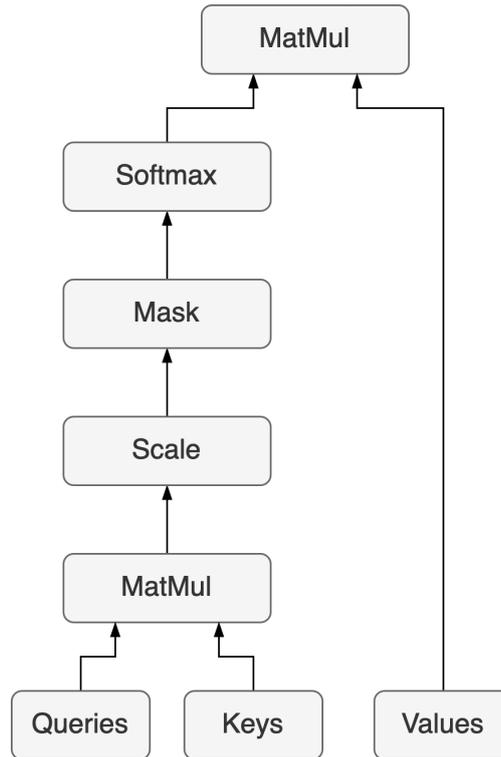
**Figure 4.8:** Transformer encoder architecture

## Transformer

The main component of the encoder and the whole basecaller model is the Transformer [32]. The Transformer architecture is based only on the attention mechanism and replaced the recurrent architectures for sequential models because it is simple and is easily parallelized as it does not rely on iteratively going through the input sequence and holding a state. Transformers were quickly employed in Natural Language Processing (NLP) tasks and Computer Vision (CV) tasks, replacing or aiding recurrent and convolutional models, and have shown state-of-the-art performance in various tasks. Originally, the Transformer consists of an encoder and decoder part, but in our case, only the encoder part is used.

A single Transformer encoder layer consists of two sub-layers: the multi-head self-attention and the position-wise feed-forward network. Each of those sub-layers has residual connections and has layer normalization applied afterward. The encoder layer is then repeated  $N$  times. The Transformer encoder architecture diagram is visualized in Figure 4.8.

The attention mechanism works by taking in queries, matching them with keys, and using corresponding values for outputs. Queries, keys, and values are represented with vectors. The output of attention is a weighted sum of values whose weights



**Figure 4.9:** Scaled dot-product attention diagram

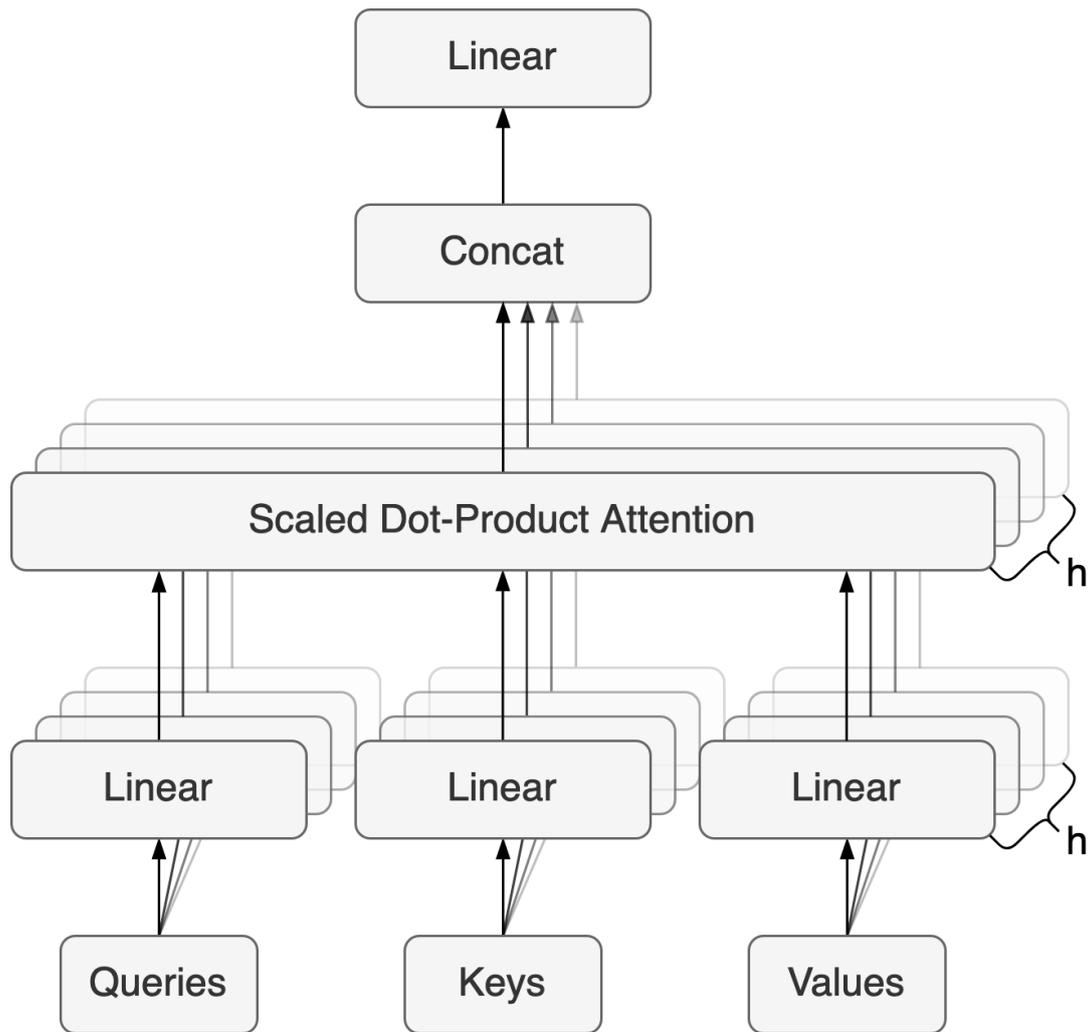
are determined by the compatibility of the query and the key that corresponds to that specific value. Concretely, the Transformer architecture computes the attention as the scaled dot-product attention. The input to scaled dot-product attention are queries and keys of dimension  $d_k$ , and values of dimension  $d_v$ . The dot product is computed between all queries and keys and then scaled by  $\sqrt{d_k}$ . To obtain the weights for the weighted sum of values, a softmax function is applied. Masking the attention is also possible if needed for padding or autoregressive models.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

To allow the model to learn different representations for the same data, the attention is split into multiple heads of attention  $h$  by taking the input vectors for queries, keys, and values, and projecting them into  $h$  linear projections of dimension  $d_k$ ,  $d_k$ , and  $d_v$  with projection matrices  $W$ . Each of these heads then computes the attention function independently and in parallel on the received projections. The attention outputs are then concatenated and projected back to the desired dimension.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O$$

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$



**Figure 4.10:** Multi-head attention diagram

This formulation applies in general, but the differences can come from what is considered as  $Q$ ,  $K$ , and  $V$  in a particular case. For the encoder part, the attention is also called self-attention because both the queries and key-value pairs are taken from the same place, i.e. queries are made on the same sequence that key-value pairs are taken from, which means that the keys and queries are the same vectors.

The second sub-layer, position-wise feed-forward network (FFN), consists of two standard linear transformations with an activation function applied between the two transformations. The first transformation changes the dimensionality to a specified value and the second transformation changes the dimensionality back to the dimensionality the model uses. While the original Transformer uses a ReLU activation function, we decided to use a GELU activation function.

Since the Transformer does not have ordering built into the architecture, the inputs need to be modified to somehow contain information about their relative or global

position in the sequence. This is done with positional encoding (PE) which can be done in a multitude of ways, but the authors chose sine and cosine function positional encoding and we decided to follow their decision here.

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

For our model, we used 10 transformer encoder layers with 8 heads, FFN dimension of 2048, 0.05 dropout, and GELU activation function.

### 4.1.3. Decoder

The decoder is attached to the encoder that can be pre-trained and can arbitrarily decrease the sequence length by downsampling the time domain. The bottleneck of CTC requires that the length of the input sequence be greater than the length of the target sequence, if the downsampling in the encoder breaks that requirement, the decoder needs to upsample back to a level that is acceptable for CTC. Since downsampling is beneficial for the performance of the model due to the complexity of the transformer that depends quadratically on the length of the input sequence, we decided to downsample below the requirement and then upsample back.

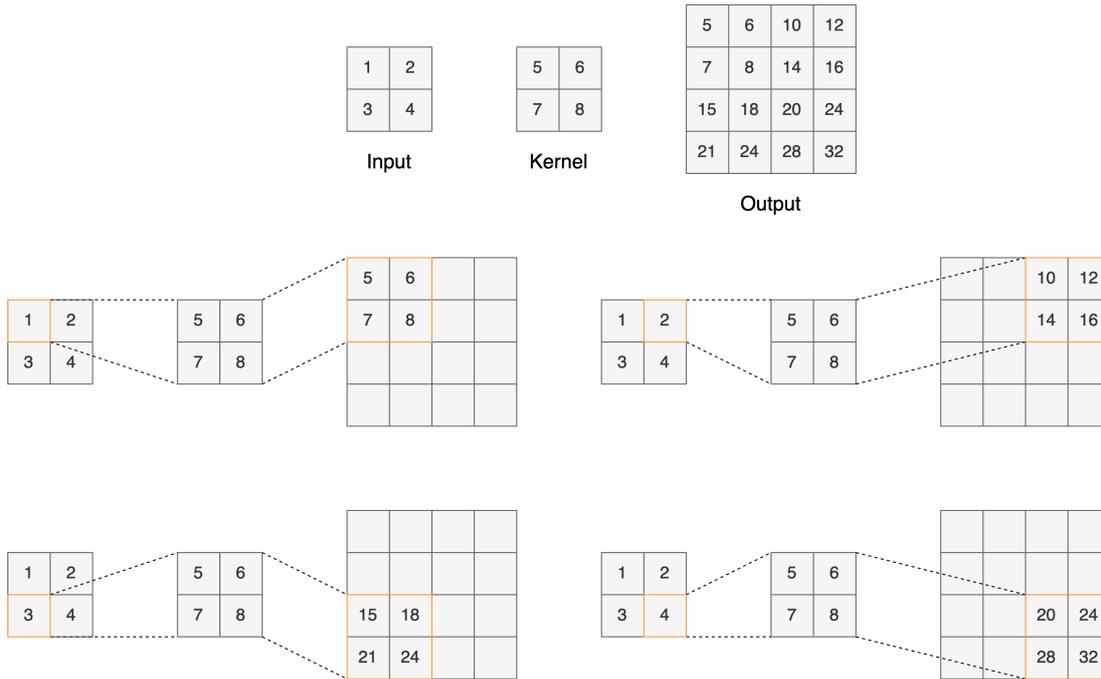
### Transposed Convolution

The method of choice for upsampling in the time domain was transposed convolution. Transposed convolution offers a way to upsample with learnable parameters. It is an operation that is similar to convolution but acts in the opposite direction. It can be compared to the backward pass of the convolution. The output is calculated by taking each element of the input and multiplying the kernel with it to produce a local output. The stride acts in the output instead of the input. Padding can be applied to both the input and the output.

The architecture used consisted of convolution and transposed convolution, batch normalization, and GELU activation function and can be seen in Table 4.2.

### Classification

To produce a classification a simple fully-connected feedforward layer is applied to set the dimension to the number of classes in the output. The number of classes in the



**Figure 4.11:** Transposed convolution example with  $2 \times 2$  input,  $2 \times 2$  kernel with stride 2 producing a  $4 \times 4$  output.

**Table 4.2:** Convolutional feature decoder architecture

Layer #	$K$	$S$	$C_{out}$	Transposed	BatchNorm	Activation
1	3	1	512	No	Yes	GELU
2	3	2	256	Yes	Yes	GELU
3	3	1	256	No	Yes	GELU

output for this basecaller model is 5:  $\{\epsilon, A, C, G, T\}$ . During training, the CTC Loss is applied to the log softmax of these outputs, and during inference, a decoding process uses the softmax of these outputs to produce a likely output sequence.

#### 4.1.4. Optimization

For training the model, we used the Adam [16] optimizer with decoupled weight decay regularization [20]. Adam stands for Adaptive Moment Estimation and is one of many iterative improvements done on previous optimization algorithms. It uses moving averages of gradients  $g$  as first moments  $m$  and squares of gradients  $g$  as second moments  $v$  with decay factors  $\beta_1, \beta_2 \in [0, 1)$ , step size  $\alpha$ , and a small scalar value  $\epsilon$  for numerical stability to optimize parameters  $\theta$ .

$$\begin{aligned}
m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \\
v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \\
\hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \\
\theta_t &= \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned}$$

Adam is a usual choice for initial experiments and can be replaced with Stochastic Gradient Descent (SGD) with a tuned learning rate and scheduling later on if necessary. For our experiments, we used an initial learning rate of  $\alpha = 1e^{-4}$  with exponential decay  $\gamma = 0.95$ , weight decay  $\lambda = 1e^{-5}$ , moment decay factors  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$ , and numerical stability constant  $\epsilon = 1e^{-8}$ .

## 4.2. Evaluation

For validation and testing purposes an accuracy metric is defined using a pairwise sequence alignment tool. The tool I decided to use for sequence alignment for validation during training is Edlib [29] and for testing, I used Minimap2 [18] since Edlib does not have the read mapping functionality. The alignment is provided in the CIGAR format as defined in [26] and specified in Table 4.3. The format specifies a string that consists of a sequence of pairs of operations that need to be applied to transform one sequence into the other and the number of times it is repeated consecutively, e.g.  $2=1\ I\ 3=2\ X\ 5=3\ D\ 1=$  would mean that to align two sequences, a query and a reference, with the defined CIGAR string, we need to match the first two elements, insert an element into the reference, match three elements, mismatch or change 2 elements, match 5 elements, delete 3 elements from the reference sequence and finally match 1 element.

```

Query:  GATTTAGTGATTA---A
        ||  |||  |||||  |
Reference: GA-TTACAGATTACAGA

```

The operations and their quantities are extracted from the CIGAR string provided by the aligner, aggregated by summation through the entire sequence and the accuracy metric is computed as follows:

$$Accuracy = \frac{=_{sum}}{=_{sum} + I_{sum} + X_{sum} + D_{sum}}$$

**Table 4.3:** Relevant CIGAR string operations specification

Op	Description	Consumes query	Consumes reference
I	insertion to the reference	Yes	No
D	deletion from the reference	No	Yes
S	soft clipping	Yes	No
=	sequence match	Yes	Yes
X	sequence mismatch	Yes	Yes

By using Minimap2 for testing the model in inference, we also had the information of what percentage of the basecalled reads Minimap2 managed to map to the reference sequence.

## 4.3. Experiments

### Blocks of Convolutions

We tried to use deeper convolutions in the feature encoder and transposed convolutions in the feature decoder, but we did not find that having many convolutional layers helped with the model accuracy.

### Separable Convolutions

During the early stages, we used more convolutional layers and separable convolutions provided a way to decrease the number of parameters, but as we moved the weight from convolutions to transformers there was no longer a need for separable convolutions and they no longer provided much use.

### Residual Connections in Convolutions

Inspired by the success of Bonito [22], we tried using blocks of convolutional layers with wide kernels with residual connections. As already mentioned in the first subsection, the model does not rely much on convolutions, but on the transformer part of the model.

### Classical Upsampling

After we settled on having most of the load on the transformer, we wanted to increase the performance by reducing the sequence length at the transformer input, so we needed a way to upsample due to the CTC input and target length requirements mentioned in subsection 4.1.3. The first choice was a classical linear upsampling algorithm which seemed to work, but was later replaced with transposed convolution to provide more capacity for the model.

### Nyströmformer

Another way to alleviate the problem with the transformer’s computational complexity is to try and use approximations that reduce the complexity. A method that does that successfully and reduces the complexity to linear complexity in the length of the input sequence is the Nyströmformer [37] which uses a Nyström-based algorithm to approximate self-attention. Unfortunately, the publicly available implementation I found re-

quired too much GPU memory for some reason and I did not manage to find a solution for that issue so we proceeded with a standard transformer.

### **ReZero**

To help the model converge faster, we tried using ReZero [1], a variation of a transformer architecture with gated residual connections using a single zero-initialized parameter. This is supposed to give faster convergence when training deep models, but we found no improvement in convergence speed when applied to our architecture.

### **Chunk Size**

Since we can't train on full-length reads because they are too long when considering the computational complexity and inefficient batching due to varying lengths of reads, we divided the data into chunks. Chunk size could only be increased to a limited size due to the restricted cuDNN CTC loss implementation that limits the length of the target sequences to 256. Taking the average 9 measurements/bp we see that the chunk size limit should be around 2304, but since the ratio between the signal and the actual sequence varies, the lower bound is in reality significantly lower, e.g. when using a chunk size of 2048, the implementation raised an error. For this reason, we stuck with a chunk size of 1024.

### **Hyperparameter Optimization**

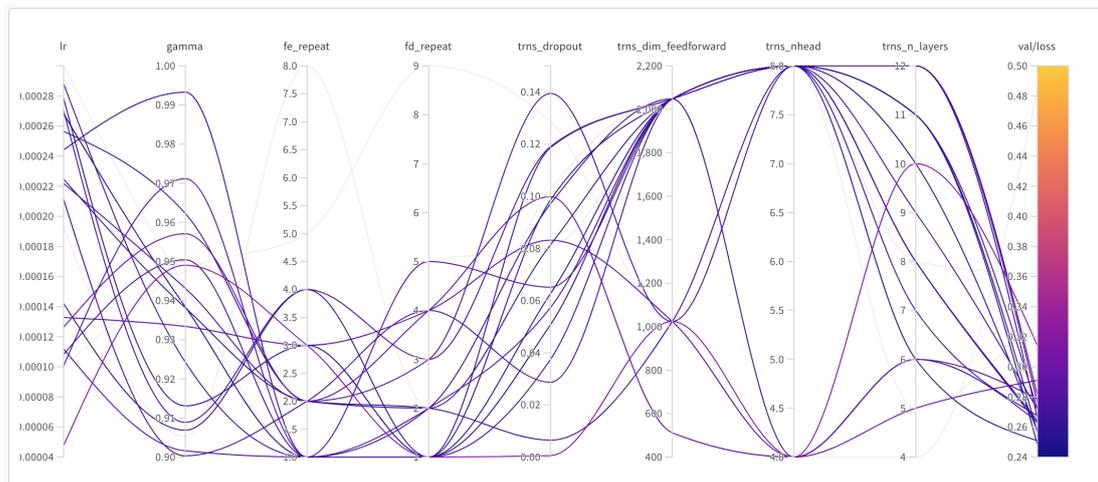
After settling on the architecture, I ran a hyperparameter optimization using Bayesian optimization on a pre-defined ranges and sets of values. Bayesian optimization constructs a probabilistic model that iteratively evaluates hyperparameter configurations using an objective function or metric and updates the probabilistic model to maximize the set objective [36]. The optimization balances exploring the hyperparameter space and finding the optimum hyperparameter configuration. The optimization was done using the Sweep functionality from Weights & Biases. It also provides a nice visualization of the whole search shown in Figures 4.12, 4.13, 4.14.

## Decoding

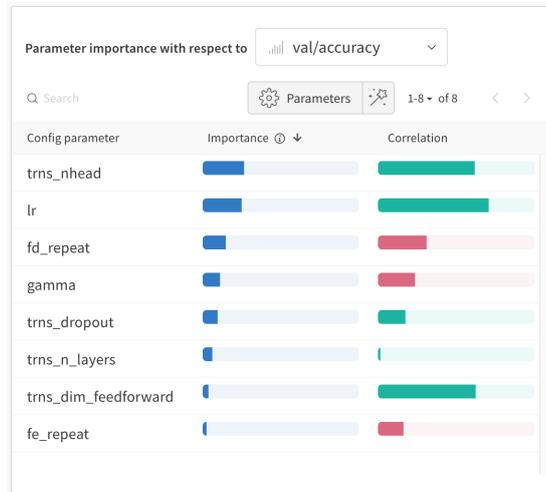
There are two methods for decoding CTC outputs: Viterbi search and beam search as described in subsection 4.1.1. Both options were tested and beam search showed no significant improvement from Viterbi search while being much slower, so Viterbi is the method of choice for decoding.

## Stochastic Weight Averaging

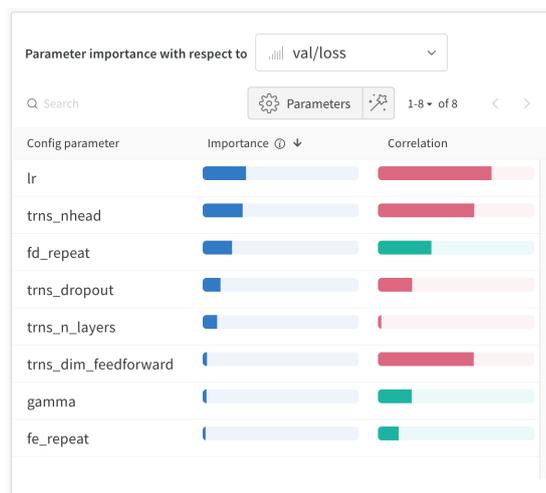
The used framework offers the Stochastic Weight Averaging (SWA) [14] functionality that can improve generalization with a negligible cost. No detailed testing was done to see if this was true in our case, but since it does not affect performance, it was used during training.



**Figure 4.12:** Sweep chart showing the used hyperparameter configurations and their respective validation metrics.



**Figure 4.13:** Computed correlations and importances of individual hyperparameters on the validation accuracy metric.



**Figure 4.14:** Computed correlations and importances of individual hyperparameters on the validation loss metric.

# 5. Implementation

## 5.1. AttentionCall

Using the methods described in this thesis, a command-line implementation called AttentionCall was developed. The implementation consists of dataset definitions, data preprocessing, module and architecture definitions, model optimization, model checkpointing, logging, and basecalling inference. AttentionCall offers a range of configurable options and hyperparameters for training. It also supports multi-GPU training. The implementation is available on GitHub on the following repository URL: <https://github.com/StanislavPavlic/attentioncall>.

Before starting work on this thesis, I was in a group project with my colleagues Sanja Deur and Rafael Josip Penić as part of our university's master's programme course "Project" where we worked on a self-supervised model that would create embeddings for nanopore sequencing data which would be used in multiple downstream tasks, including basecalling. Throughout the project, we tried different approaches and tasks to train the model inspired by other successful methods such as wav2vec 2.0 [2], BERT [6], and O3N [8] but struggled to find a method that works for our data. After the project ended, we each went on to work on our assigned tasks. Rafael continued working on the self-supervised model, Sanja started working on modified base detection, and I started working on the first downstream task for our self-supervised model. Unfortunately, I didn't get to utilize a pre-trained self-supervised model but instead trained directly on the downstream task from scratch.

## 5.2. Libraries, Frameworks, Dependencies

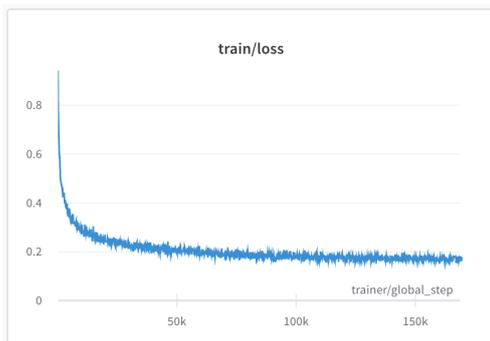
AttentionCall was implemented in Python using PyTorch, an open-source machine learning library with automatic differentiation. PyTorch offers implementations of many layers, loss functions, optimizers, and most things commonly used in machine learning. In addition to PyTorch, we also used PyTorch Lightning, an open-source library that offers a high-level interface for PyTorch and ready-made implementations for useful things like training and validation loops, device management, logging, debugging, and checkpointing. Other used software:

- Weights & Biases ([wandb.ai](https://wandb.ai))
  - experiment tracking
- Edlib ([github.com/Martinsos/edlib](https://github.com/Martinsos/edlib))
  - sequence alignment
- Minimap2 ([github.com/lh3/minimap2](https://github.com/lh3/minimap2))
  - sequence mapping and alignment
- fast-ctc-decode ([github.com/nanoporetech/fast-ctc-decode](https://github.com/nanoporetech/fast-ctc-decode))
  - CTC decoding
- ont-fast5-api ([github.com/nanoporetech/ont\\_fast5\\_api](https://github.com/nanoporetech/ont_fast5_api))
  - FAST5 file interface

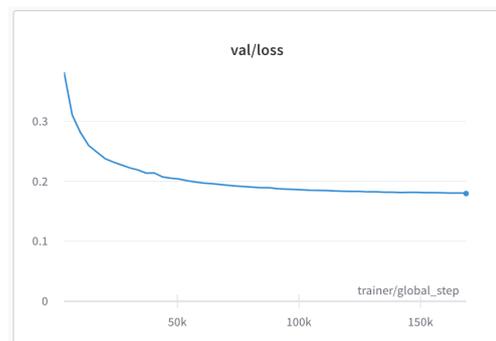
# 6. Results

## 6.1. Training

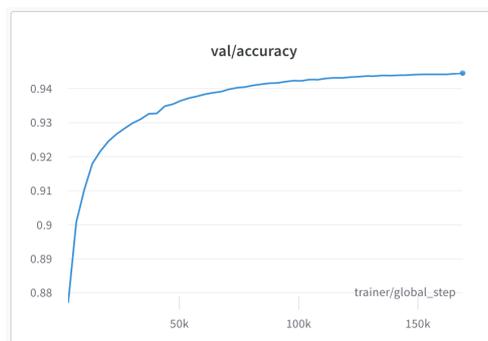
The model was trained for 50 epochs on 4 Tesla V100 GPUs. It achieved 94.46% accuracy on the validation set and the model seemed to have converged. The training loss, validation loss, and validation accuracy throughout training can be seen in Figure 6.1. The used model checkpoint is available at [api.wandb.ai/artifactsV2/gcp-us/spavlic/QXJ0aWZzhY3Q6MTE3MjUyMzk=/84f78a7c45e6395fee7c997d7a44204a](https://api.wandb.ai/artifactsV2/gcp-us/spavlic/QXJ0aWZzhY3Q6MTE3MjUyMzk=/84f78a7c45e6395fee7c997d7a44204a).



(a) Training loss



(b) Validation loss



(c) Validation accuracy

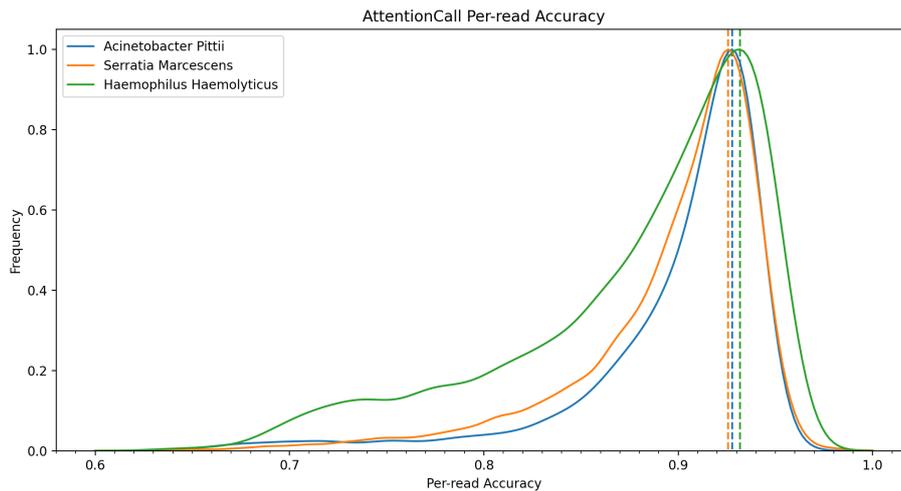
**Figure 6.1:** Metrics tracked through training

## 6.2. Comparison

To evaluate the model, two other basecallers were chosen to compare them: Guppy v4.4.2 and Bonito v0.2.0. A thing to note, this is an older version of Bonito that used the modified QuartzNet [17] architecture, but there are newer versions that use RNNs and Conditional Random Fields (CRFs) that have much better accuracy and were not used in this comparison. This was because I had issues with either dependencies or resource limits with newer versions. Unfortunately, Guppy and Bonito were not retrained on the same dataset that AttentionCall was trained on, so it is an unequal comparison of their performances. To test the basecallers, they were run on three different datasets from Ryan Wick’s benchmark [35] data collection that has raw FAST5 files and their corresponding references: *Acinetobacter pittii*, *Serratia marcescens*, and *Haemophilus haemolyticus*. Testing was done on a single RTX 2080 with 11 GB of memory.

**Table 6.1:** Basecaller performance comparison for *Acinetobacter Pittii*

Basecaller	Time [s]	RAM [GB]	GPU Mem [GB]	Chunk Size
Guppy	93.43	2.1	9.6	2,000
Bonito	1,297.24	2.3	10.9	4,000
AttentionCall	1,448.81	7.8	2.8	1,024



**Figure 6.2:** Per-read accuracy distribution for AttentionCall.

When looking at computational performance in Table 6.1, Guppy is the obvious winner. This is because it is well optimized and parallelized. Bonito and Attention-

Call are relatively close when it comes to speed. AttentionCall was not optimized for inference, it handles each read individually which means there are a lot of incomplete batches being used and it is not parallelized. This can be seen by the amount of unused GPU memory that can be utilized. The input data is read into RAM at once and held there for the duration of basecalling. This is visible in the relatively large amount of RAM that it uses.

**Table 6.2:** Read accuracy comparison for different organisms

Basecaller	<i>A. pittii</i>	<i>S. mercescens</i>	<i>H. haemolyticus</i>
Guppy	93.03	92.16	91.02
Bonito	92.58	92.07	91.22
AttentionCall	90.09	89.48	88.18

**Table 6.3:** Consensus accuracy comparison for different organisms

Basecaller	<i>A. pittii</i>	<i>S. mercescens</i>	<i>H. haemolyticus</i>
Guppy	99.97	99.93	99.85
Bonito	99.96	99.94	99.91
AttentionCall	99.85	99.83	-

When looking at read accuracy in Table 6.2, Guppy and Bonito are close with Guppy being better on *A. pittii*, slightly better on *S. mercescens*, and Bonito being slightly better on *H. haemolyticus*. AttentionCall is not far behind, but the other two are significantly better. Again, it is good to note that recent evaluations show that Bonito has improved performance in the newer versions and is better than Guppy when it comes to accuracy [28]. The consensus accuracy for AttentionCall on *H. haemolyticus* is missing because Rebaler failed to generate a consensus due to an unknown issue.

Another measure that can be looked at to evaluate a basecaller is majority-rule consensus accuracy shown in Table 6.3. This is done by creating a consensus sequence from overlapping reads at the same genomic location and measuring the identity as described in [35]. Consensus was created using Rebaler also mentioned and used in [35].

## 7. Conclusion

The goal of this thesis was to develop a basecaller using deep learning with self-supervised pre-training in mind. The developed basecaller shows solid performance but lags behind the competition. The basecaller uses a simple architecture that includes a convolutional feature encoder, a vanilla transformer, and a transposed convolution feature decoder.

With modifications to the architecture with the input data structure in mind, the model might be able to achieve better accuracy. CRFs have become popular recently and have been shown to improve performance (which is what newer versions of Bonito use). Label smoothing can be applied to improve accuracy. Another thing that could perhaps be explored is using the detailed alignment of the signal and the ground truth instead of just using it to chunk the data.

There is a lot of space for improvement in computational performance during inference. It can be parallelized to increase the speed. It can be generalized to a group of reads instead of individual reads when basecalling to decrease the number of incomplete batches and increase GPU utilization and efficient memory usage. The data can be loaded from disk instead of holding all of it in RAM during basecalling. The vanilla transformer could be replaced with a more efficient variation like Nyströmformer [37], Linformer [33], or Performer [5], which would also help during training by decreasing the training time.

To better understand how to improve the basecaller, it would be beneficial to analyze the mistakes it makes by checking the distribution of errors by type (mismatch, insertion, deletion). It would also make sense to see the accuracy for homopolymers.

With more time and resources, the data can be explored further to obtain a better understanding of the underlying structure, and more methods can be experimented with to improve the accuracy and performance of the model.

# BIBLIOGRAPHY

- [1] Thomas Bachlechner, Bodhisattwa Prasad Majumder, Huanru Henry Mao, Garrison W. Cottrell, i Julian McAuley. Rezero is all you need: Fast convergence at large depth, 2020.
- [2] Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, i Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations, 2020.
- [3] Ting Chen, Simon Kornblith, Mohammad Norouzi, i Geoffrey Hinton. A simple framework for contrastive learning of visual representations, 2020.
- [4] S. Chopra, R. Hadsell, i Y. LeCun. Learning a similarity metric discriminatively, with application to face verification. U *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, svezak 1, stranice 539–546 vol. 1, 2005. doi: 10.1109/CVPR.2005.202.
- [5] Krzysztof Choromanski, Valerii Likhoshesterov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, i Adrian Weller. Rethinking attention with performers, 2021.
- [6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, i Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.
- [7] John Eid, Adrian Fehr, Jeremy Gray, Khai Luong, John Lyle, Geoff Otto, Paul Peluso, David Rank, Primo Baybayan, Brad Bettman, Arkadiusz Bibillo, Keith Bjornson, Bidhan Chaudhuri, Frederick Christians, Ronald Cicero, Sonya Clark, Ravindra Dalal, Alex deWinter, John Dixon, Mathieu Foquet, Alfred Gaertner, Paul Hardenbol, Cheryl Heiner, Kevin Hester, David Holden, Gregory Kearns, Xiangxu Kong, Ronald Kuse, Yves Lacroix, Steven Lin, Paul Lundquist, Congcong Ma, Patrick Marks, Mark Maxham, Devon Murphy, Insil Park, Thang Pham, Michael Phillips, Joy Roy, Robert Sebra, Gene Shen, Jon Sorenson,

- Austin Tomaney, Kevin Travers, Mark Trulson, John Vieceli, Jeffrey Wegener, Dawn Wu, Alicia Yang, Denis Zaccarin, Peter Zhao, Frank Zhong, Jonas Krolach, i Stephen Turner. Real-time dna sequencing from single polymerase molecules. *Science*, 323(5910):133–138, 2009. ISSN 0036-8075. doi: 10.1126/science.1162986. URL <https://science.sciencemag.org/content/323/5910/133>.
- [8] Basura Fernando, Hakan Bilen, Efstratios Gavves, i Stephen Gould. Self-supervised video representation learning with odd-one-out networks, 2017.
- [9] Alex Graves, Santiago Fernández, i Faustino Gomez. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. U *In Proceedings of the International Conference on Machine Learning, ICML 2006*, stranice 369–376, 2006.
- [10] Michael Gutmann i Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. *Journal of Machine Learning Research - Proceedings Track*, 9:297–304, 01 2010.
- [11] Kui Han, Zhi-feng Li, Ran Peng, Li-ping Zhu, Tao Zhou, Lu-guang Wang, Shu-guang Li, Xiao-bo Zhang, Wei Hu, Zhi-hong Wu, Nan Qin, i Yue-zhong Li. Extraordinary expansion of a sorangium cellulosum genome from an alkaline milieu. *Scientific reports*, 3:2101–2101, 2013. ISSN 2045-2322. doi: 10.1038/srep02101. URL <https://pubmed.ncbi.nlm.nih.gov/23812535>. 23812535[pmid].
- [12] Dan Hendrycks i Kevin Gimpel. Gaussian error linear units (gelus), 2020.
- [13] Sergey Ioffe i Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.
- [14] Pavel Izmailov, Dmitrii Podoprikin, Timur Garipov, Dmitry Vetrov, i Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization, 2019.
- [15] Miten Jain, Hugh E. Olsen, Benedict Paten, i Mark Akeson. The oxford nanopore minion: delivery of nanopore sequencing to the genomics community. *Genome Biology*, 17(1):239, Nov 2016. ISSN 1474-760X. doi: 10.1186/s13059-016-1103-0. URL <https://doi.org/10.1186/s13059-016-1103-0>.

- [16] Diederik P. Kingma i Jimmy Ba. Adam: A method for stochastic optimization, 2017.
- [17] Samuel Krivan, Stanislav Beliaev, Boris Ginsburg, Jocelyn Huang, Oleksii Kuchaiev, Vitaly Lavrukhin, Ryan Leary, Jason Li, i Yang Zhang. Quartznet: Deep automatic speech recognition with 1d time-channel separable convolutions, 2019.
- [18] Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 05 2018. ISSN 1367-4803. doi: 10.1093/bioinformatics/bty191. URL <https://doi.org/10.1093/bioinformatics/bty191>.
- [19] Jason Li, Vitaly Lavrukhin, Boris Ginsburg, Ryan Leary, Oleksii Kuchaiev, Jonathan M. Cohen, Huyen Nguyen, i Ravi Teja Gadde. Jasper: An end-to-end convolutional neural acoustic model, 2019.
- [20] Ilya Loshchilov i Frank Hutter. Decoupled weight decay regularization, 2019.
- [21] John P. McCutcheon i Carol D. von Dohlen. An interdependent metabolic patchwork in the nested symbiosis of mealybugs. *Current biology : CB*, 21(16): 1366–1372, Aug 2011. ISSN 1879-0445. doi: 10.1016/j.cub.2011.06.051. URL <https://pubmed.ncbi.nlm.nih.gov/21835622>. 21835622[pmid].
- [22] Nanoporetech. Bonito, 2021. URL <https://github.com/nanoporetech/bonito>.
- [23] Nanoporetech. Taiyaki, 2021. URL <https://github.com/nanoporetech/taiyaki>.
- [24] Nanoporetech. Re-squiggle algorithm - tombo, 2021. URL <https://nanoporetech.github.io/tombo/resquiggle.html>.
- [25] Alexander Payne, Nadine Holmes, Vardhman Rakyan, i Matthew Loose. Whale watching with bulkvis: A graphical viewer for oxford nanopore bulk fast5 files. *bioRxiv*, 2018. doi: 10.1101/312256. URL <https://www.biorxiv.org/content/early/2018/05/03/312256>.
- [26] Samtools. Sam/bam and related specifications, 2021. URL <https://github.com/samtools/hts-specs#sambam-and-related-specifications>.

- [27] Florian Schroff, Dmitry Kalenichenko, i James Philbin. Facenet: A unified embedding for face recognition and clustering. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun 2015. doi: 10.1109/cvpr.2015.7298682. URL <http://dx.doi.org/10.1109/CVPR.2015.7298682>.
- [28] Jordi Silvestre-Ryan i Ian Holmes. Pair consensus decoding improves accuracy of neural network basecallers for nanopore sequencing. *bioRxiv*, 2020. doi: 10.1101/2020.02.25.956771. URL <https://www.biorxiv.org/content/early/2020/02/25/2020.02.25.956771>.
- [29] Martin Šošić i Mile Šikić. Edlib: a c/c++ library for fast, exact sequence alignment using edit distance. *bioRxiv*, 2016. doi: 10.1101/070649. URL <https://www.biorxiv.org/content/early/2016/08/23/070649>.
- [30] Oxford Nanopore Technologies. New research algorithms yield accuracy gains for nanopore sequencing, May 2021. URL <https://nanoporetech.com/about-us/news/new-research-algorithms-yield-accuracy-gains-nanopore-sequencing>.
- [31] Aaron van den Oord, Yazhe Li, i Oriol Vinyals. Representation learning with contrastive predictive coding, 2019.
- [32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, i Illia Polosukhin. Attention is all you need, 2017.
- [33] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, i Hao Ma. Linformer: Self-attention with linear complexity, 2020.
- [34] Lilian Weng. Self-supervised representation learning. *lilianweng.github.io/lil-log*, 2019. URL <https://lilianweng.github.io/lil-log/2019/11/10/self-supervised-learning.html>.
- [35] Ryan R. Wick, Louise M. Judd, i Kathryn E. Holt. Performance of neural network basecalling tools for oxford nanopore sequencing. *Genome Biology*, 20(1):129, Jun 2019. ISSN 1474-760X. doi: 10.1186/s13059-019-1727-y. URL <https://doi.org/10.1186/s13059-019-1727-y>.

- [36] Wikipedia contributors. Hyperparameter optimization — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Hyperparameter\\_optimization&oldid=1022309479](https://en.wikipedia.org/w/index.php?title=Hyperparameter_optimization&oldid=1022309479), 2021. [Online; accessed 21-June-2021].
- [37] Yunyang Xiong, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, i Vikas Singh. Nyströmformer: A nyström-based algorithm for approximating self-attention, 2021.

# DNA Nanopore Sequencing Basecaller

## Abstract

Nanopore sequencing is one of the state-of-the-art sequencing technologies. It passes a DNA sample through a pore which changes the ionic current in the pore. Due to the size of the pore, there are usually five nucleotides (5-mer) present in the pore influencing the measured signal. Each of the 1024 possible 5-mers produces a different signal, and this information is used for basecalling (converting the raw signal to a sequence of nucleotides). The signal is approximately rectangular because the 5-mer changes one nucleotide at a time, but there is a lot of noise present. The goal of this thesis was to develop a DNA nanopore sequencing basecaller using modern deep learning architectures with self-supervised learning in mind. The architecture is mainly based on transformers. The basecaller was evaluated on publicly available datasets. The solution called AttentionCall was implemented in Python and the PyTorch library. The source code is available on GitHub at [github.com/StanislawPavlic/attentioncall](https://github.com/StanislawPavlic/attentioncall).

**Keywords:** bioinformatics, basecalling, nanopore sequencing, deep learning, transformers, CTC.

## **Metoda za pretvaranje signala dobivenog sekvenciranjem nanoporama u niz nukleotida**

### **Sažetak**

Sekvenciranje nanoporama je jedna od vodećih tehnologija sekvenciranja danas. Prolaskom DNA kroz poru mijenja se ionska struja koja teče porom. Uslijed veličine pore, obilno se u pori nalazi 5 nukleotida (5-torka) koji utječu na mjereni signal. Svaka od 1024 moguće 5-torke uzrokuje različit signal i ta informacija se koristi za određivanje baza (pretvaranje sirovog signala u slijed nukleotida). Oblik signala je pravokutan jer se 5-torke mijenjaju po jedan nukleotid, ali je prisutan šum. Cilj rada je razvoj nove metode za pretvaranje signala dobivenog sekvenciranjem nanoporama korištenjem suvremenih arhitektura dubokog učenja, pritom imajući na umu mogućnost korištenja samo-nadziranog učenja. Arhitektura se temelji na transformerima. Za evaluaciju su korišteni javno dostupni skupovi podataka. Rješenje je implementirano u programskom jeziku Python uz korištenje PyTorch biblioteke. Izvorni kod je dostupan na Github repozitoriju na [github.com/StanislavPavlic/attentioncall](https://github.com/StanislavPavlic/attentioncall).

**Ključne riječi:** bioinformatika, određivanje baza, sekvenciranje nanoporama, duboko učenje, transformeri, CTC.