Refining Homopolymer Predictions in Nanopore Sequences

Nicklas Kirkegaard Breum, Martin Nyholm Jørgensen,
Jakob Stig Olesen

Software, MI107F20
June 2020

Master Thesis

Department of Computer Science
Aalborg University
https://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Refining Homopolymer Predictions in Nanopore Sequences

Theme:
Master Thesis

Project Period:
February 2020 – June 2020

Project Group:
MI107F20

Participant(s):
Nicklas Kirkegaard Breum
Martin Nyholm Jørgensen
Jakob Stig Olesen

Supervisor(s):
Katja Hose
Thomas Dyhre Nielsen

Page Numbers: 92

Date of Completion:
June 4th, 2020

Abstract:

In DNA basecalling one of the remaining problems is to reliable basecall homopolymers. A homopolymers is difficult to basecall because it consists of a region of unchanging signals. We also find that there is a significant overlap in the signal-lengths of homopolymers of varying length, exemplifying prior difficulties in homopolymer basecalling.
We propose a pipeline to support an existing basecaller where signals produced by ONT nanopores is segmented into homopolymer and non-homopolymer regions, such that homopolymers may be processed directly. To this end, we use a segmentation model based on the U-Net architecture and a homopolymer-length model based on the ResNet achitecture. When our models are combined with the default Bonito model we improve the identity-rate by 2% achieving an identity-rate of 83% compared with the standalone Bonito model with an identity-rate of 81%.

Preface

This project was done as part of our master thesis in Machine Intelligence and in cooperation with The Center of Microbial Communities of The Department of Chemistry and Bioscience, Aalborg University.

We would like to thank Mads Albertsen, Rasmus Kirkegaard, and their colleagues at The Center of Microbial Communities guidance and answering our questions regarding the biological aspects of basecalling.

Lastly, we extend our thanks to our supervisors Katja Hose and Thomas Dyhre Nielsen for their keen eye to detail and patience regarding our project.

Aalborg University, 4th June 2020

Nicklas Kirkegaard Breum
<nbreum15@student.aau.dk>

Martin Nyholm Jørgensen
<mnja15@student.aau.dk>

Jakob Stig Olesen
<jolese15@student.aau.dk>

Contents

Chapter 1

Introduction

DNA sequencing has become an accessible technology with the advent of readily available bioengineered nanopores by Oxford Nanopore Technologies (ONT). These nanopores allow DNA to be sequenced fast and cheaply compared to traditional sequencing methods. A current is observed as DNA, or RNA, translocates through the nanopore, and this signal can be analyzed and converted to DNA bases (A, T, C, and G) through basecalling. This is a non-trivial task, as several bases influence the observable current, therefore, a observed does not correspond to a singular base, but instead several bases, i.e. a k-mer, where k is the number of bases (or context) influencing the signal. The R9.4.3 cells from ONT have a context of 5, i.e. it produces signals corresponding to a 5-mer [32]. Additionally, the translocation speed may vary throughout a sequencing run, whereby the number of signals corresponding to a single base is non-uniform.

In the basecalling community, several basecallers perform exceedingly well, with ONT boasting $> 95\%$ accuracy in the latest iteration of their commercial basecaller Guppy [36]. However, the current generation of basecallers are still troubled by the occurrence of homopolymers (stretches of a repeating base) in the DNA. In our previous project [29], we encountered many such instances in the literature where homopolymers wrought problems.

This is unfortunate, as accurate basecalling is essential for certain biological analyses. For instance, the analysis of the poly(A) tail found in the majority of eukaryotic mRNAs[1], where the length of this tail can be used to determine the maturity of the mRNA [43].

Accurate basecalling of long homopolymers currently poses great challenges for all available basecallers. Therefore, in this project we will investigate possible solutions to the basecalling of homopolymers from nanopore sequences.

1.1 Basecalling

The following section has been copied (with minor modifications) from our previous report [29].

DNA sequencing is the general term for finding bases in a DNA, and a popular technique for doing so is basecalling: the process of translating electrical signals into DNA bases. To better understand basecalling, the following will give an introduction to the nature of DNA.

[1] Messenger RNA is used in the synthesis of proteins.

Deoxyribonucleic acid, or DNA, is the double-stranded molecule (see Figure 1.1) that makes life possible. An organism's genome contains all of its DNA and it encodes all the information that make us who we are. DNA is a double-stranded, helical polymer of nucleotides. A nucleotide consists of a phosphate group, a sugar, and a base [1]. In this project, we are only interested in finding the bases of a DNA string, and therefore we will not expound the specifics of the phosphate groups and sugars. In DNA the four primary, or canonical, bases are: adenine (A), cytosine (C), guanine (G), and thymine (T) — these can be thought of as the alphabet of the DNA. In the double-stranded DNA these form basepairs with a base on each strand, they pair as follows: A bonded to T and C to G, as illustrated in Figure 1.1.

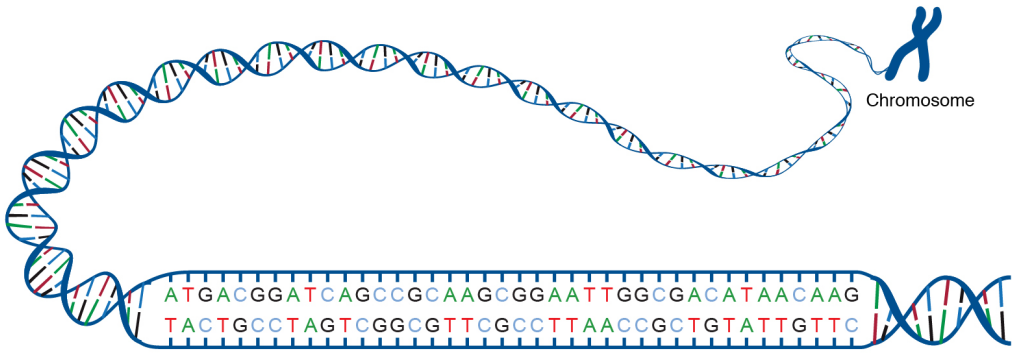


Figure 1.1: Illustration of the structure of DNA [15].

It is possible for other bases to be present in DNA, through modifications either organically or synthetically [1]. One can think of these modifications as expanding the alphabet of the DNA beyond the usual A, T, C, and G.

Returning to basecalling, a device that can measure electrical signals from DNA is the Nanopore R9.4 (henceforth referred to as the *nanopore*), the second most recent iteration from Oxford Nanopore Technologies (at the time of writing). A DNA strand is sequenced by being physically pulled through the nanopore and the bases present in the narrowest part changes the electrical resistance of the nanopore [42]. The R9.4 fits approximately 5 bases inside and the signal thus corresponds to a 5-mer, which refers to every possible sequence of 5 bases. The amount of possible k-mers can be calculated as n^k, where n is the size of the DNA alphabet.

1.1.1 Homopolymer Difficulties

One problem of basecalling is the appearance of homopolymers, i.e. a repeat of an identical base (e.g. AAAAAA). The problem with homopolymers is that the signal of the nanopore does not change significantly because the bases currently residing in the nanopore stay alike while the homopolymer passes through [33]. For this reason, homopolymers is a significant cause for inaccurate basecalling. This is made more difficult due to the fact that the DNA

may momentarily become stuck in the nanopore [40], this leads to widely varying lengths for identical parts of a DNA molecule.

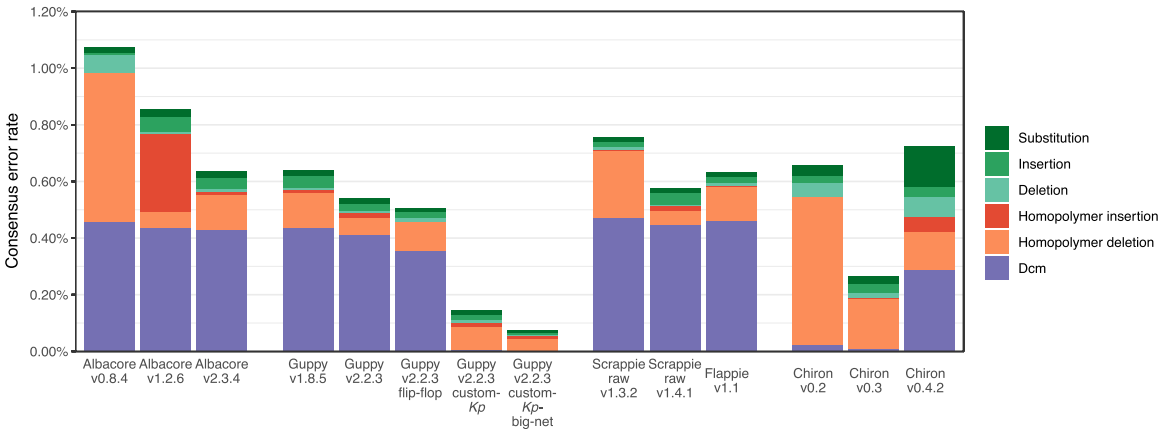


Figure 1.2: Errors per basecaller for the *K. pneumoniae* benchmarking set, broken down by type. [42]

'Performance of neural network basecalling tools for Oxford Nanopore sequencing' by Ryan R. Wick, Louise M. Judd and Kathryn E. Holt performed an analysis on error profiles of some contemporary basecallers. They quantified the error profiles for the *K. pneumoniae* benchmarking genome in terms of Dcm-methylation sites, homopolymers, and other sites (see Figure 1.2).

Most of the benchmarked basecallers have high rates of the Dcm-errors. Therefore concluding that the errors for Dcm-methylation should be classified is the most stressing error would be straight-forward. However, DNA methylation corresponds to expanding the alphabet of DNA bases beyond the usual A, C, T, and G. For Dcm-methylation this means that a methylation of the patterns CCAGG and CCTGG has occured in the second C [8]. The solution to the Dcm-errors is simple, one only needs to retrain a basecaller with training set that includes Dcm-methylations for the desired basecaller. Early versions of Chiron and the *custom-Kp* versions of Guppy were trained on datasets containing the Dcm-methylations and thus obtained lower error-rate on these.

Ignoring the Dcm errors, Figure 1.2 shows that homopolymers is the most frequent group of error rates. The homopolymer deletions means that a homopolymer was predicted to have shorter length than expected. Conversely, homopolymer insertions are predictions of longer homopolymers than expected.

Though ONT is working on new nanopores that allow for more accurately measuring the bases as they pass through [6], significant gain for basecalling can still be achieved if the problem of homopolymers can be addressed.

1.2 Problem Description

A major problem for basecalling of nanopore sequences is caused by the presence of homopolymers. The problem arises when the length of a homopolymer exceeds the context size of the nanopore. When this is the case, the electrical signal does not change while the bases comprising a homopolymer translocates through the pore. Only when a distinctly other base than those comprising the homopolymer enters the context does the signal change

observably.

Inaccurate basecalling of long homopolymers introduces systematic errors in a consensus sequence, and is thereby an inaccurate representation of the real DNA sequence. As such, the consensus sequence may obscure alterations of the real DNA sequence, and detection of mutations occurring in homopolymers are thus unreliable.

Oxford Nanopore Technologies have recently released the R10.3 nanopore [32]. This version increased the size of the context specifically in order to improve homopolymer basecalling. The context from R9.4.3 to R10.3 has been increased from ∼5bp to ∼9bp [32]. Still, this does not solve the problem entirely, it only slightly increases the length of homopolymers that can accurately be basecalled. If the homopolymer exceeds the size of the context, the same problem arises.

Therefore, increasing the size of the context of the nanopore is not a viable solution for homopolymer basecalling. As this is the case, we will attempt a software approach for inferring homopolymers.

To this end we formulate the following problem statement:

How can accurate inference of homopolymers
be achieved from nanopore sequences?

Chapter 2

Data Analysis

In order to construct a basecaller that can reliably basecall homopolymers, we need to understand how homopolymers are represented in the underlying data. To get this understanding we have examined nanopore data in homopolymeric regions and will showcase some statistics and visualizations of homopolymer signals. This exploration will highlight the difficulties of working with nanopore sequences. Additionally, we will examine if it is feasible to infer the translocation speed of the DNA strand as it passes through the nanopore.

We will be using the Loman Lab Mock Community[24] dataset to showcase homopolymers in nanopore sequences. The dataset consists of >240GB of Fast5 files extracted from ZymoBIOMICS Microbial Community Standard (Lot ZRC190633), which consists of 8 bacteria and 2 yeasts.

For convenience, we shall adopt the shorthand nX to denote a homopolymer of an arbitrary length and base. This should be read as *"homopolymers of X"*. E.g. $13A$ is an adenine homopolymer of length 13.

2.1 Taiyaki and Tombo

In subsequent sections either Tombo or Taiyaki is used as part of their data preparation, therefore will briefly give a common introduction to these. Both tools produce an alignment between a basecalled sequence and a reference, which maps the basecalled sequence, such that signal segments correspond to their reference base.

At first, we used the *resquiggle* algorithm of Tombo [37] to align signals to a reference genome. The resquiggle algorithm takes as input a basecalled fast5 file and a reference genome (fasta file) and maps the raw signals to the reference genome. This produces triples $(start\_pos, end\_pos, base)$ that specify the position of each base in the signals. [29]

We later became aware of the data preparation built into Taiyaki [38], ONT's tool for training basecaller models. Taiyaki uses a Guppy model to map raw signals to a reference. The output is the same as Tombo, triples $(start\_pos, end\_pos, base)$ that specify the position of each base in the signals.

2.2 Signal-Base Length Correlation

Our objective is a homopolymer basecaller, so in this we only consider homopolymeric regions. This section will examine if there exists a correlation between homopolymer length and homopolymer signal length in the nanopore sequences.

These were found using Tombo across the Loman Lab Mock Community[24], and we selected all homopolymers with a length greater than 5, as that is the context of the R9.4.3 nanopore that sequenced the data. Homopolymers were found by selecting repeating bases in the alignment.

To visualize the up-going trend for signals length, we used linear regression to estimate the average increase in signal length per extra base. Specifically, we used *ordinary least squares* to estimate the parameters of the linear equation that best fit the data.

This is shown in Figure 2.1. The equation for the trendline is $y = 12.122x + 52.699$, which means that the average additional signals of each base in a homopolymer is 12.122. The trendline has $R^2 = 0.284$, and indicates many outliers among the signals, which are very apparent in Figure 2.1. Appendix A.3 highlights these outliers well, as many values are seen outside all boxes for both Taiyaki and Tombo. However, the interception of the y-axis at 52.699 is a little odd, as it implies that a homopolymer of length 0 spans 52.699 signals. The low R^2 indicates that it would be difficult for a subsequent model to yield reliable results due to high variance in signal length of homopolymers.

2.2.1 Homopolymer Signals

Next, we will examine some of the peculiarities of the signals themselves. Figure 2.2 shows the normalized signals corresponding to 25 different 18As. Interestingly, it seems that the theory that a homopolymer tail is unchanging is not universal. There are some clear and considerable changes in the signals that vary significantly from the mean of the homopolymer tail at roughly $y = -0.3$. Still, the tail of all homopolymers start after $\sim$50 signals, indicated by the transition from distinct signal levels in the beginning to more or less the same signal levels in the tail. This makes sense, because the R9.4.1 nanopore considers about 5 bases during sequencing [32], and from the trendline previously mentioned we found each additional base increases the signal by about 12 signals. I.e. $4 \cdot 12.122 \approx 48$ is about the number of signals before the last base not part of the homopolymer leaves the nanopore.

The wide range of signal lengths and signal level variations in the homopolymer tails is indicative of why homopolymers are basecalled with such difficulty.

2.3 Comparison of Homopolymeric Regions

In the previous section we found a significant overlap of homopolymer signal lengths occur when using Tombo. And as mentioned, we initially only used Tombo, but we later became aware of ONT's similar tool, Taiyaki. In this section, we compare Tombo and Taiyaki to see if Taiyaki produces a more precise signal-to-reference alignment.

We compared the segmentation of homopolymeric regions from Taiyaki and Tombo. We picked a subset of reads in the following way. Over 2 million reads were first processed with Tombo, (Appendix A.4 shows the distribution of HPs in these reads). From these 2 million reads, 18,491 reads were selected s.t. each contained multiple homopolymers between 4X

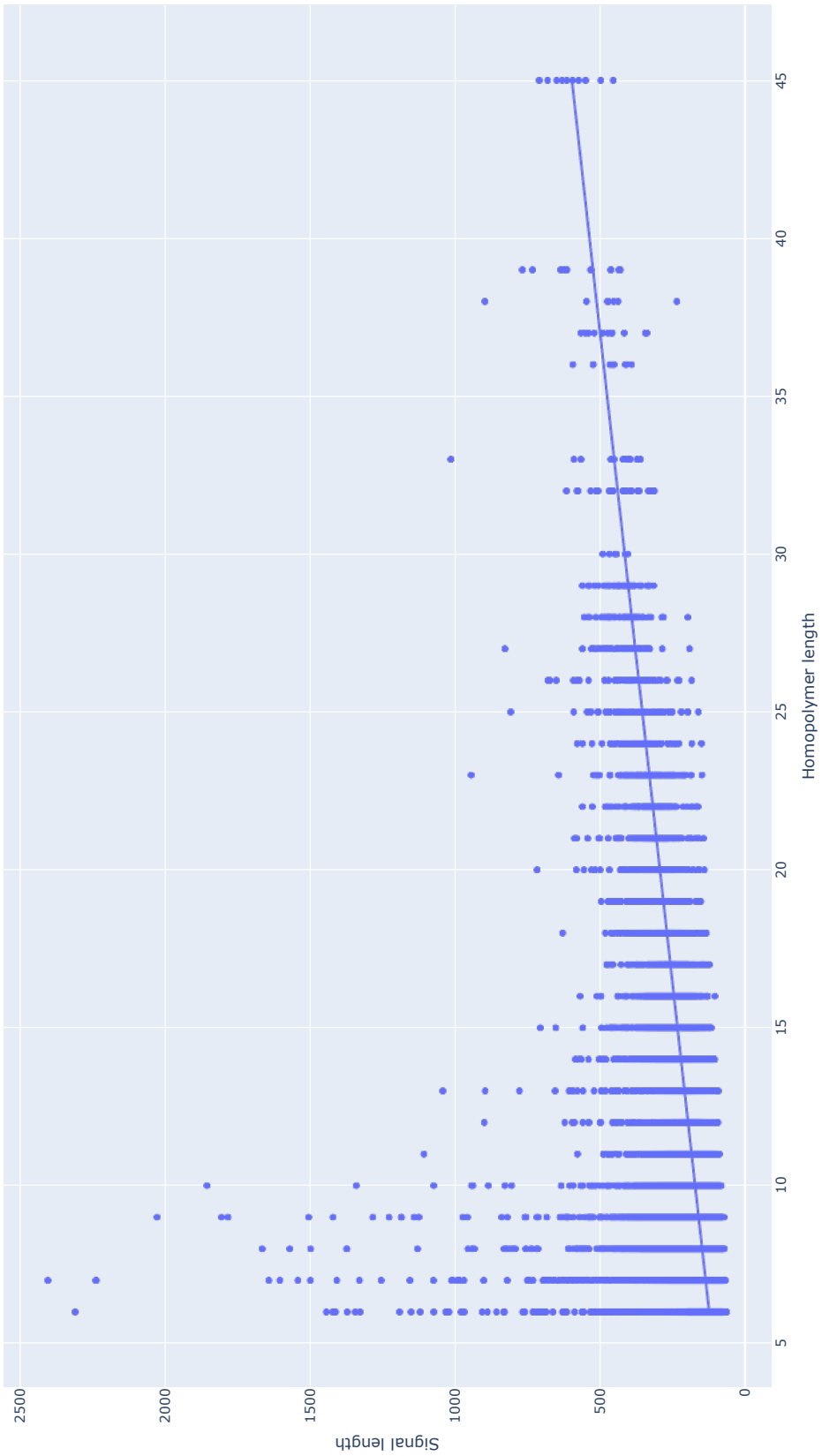


Figure 2.1: Signal length of homopolymers in Loman Lab Mock Community[24] with trendline.

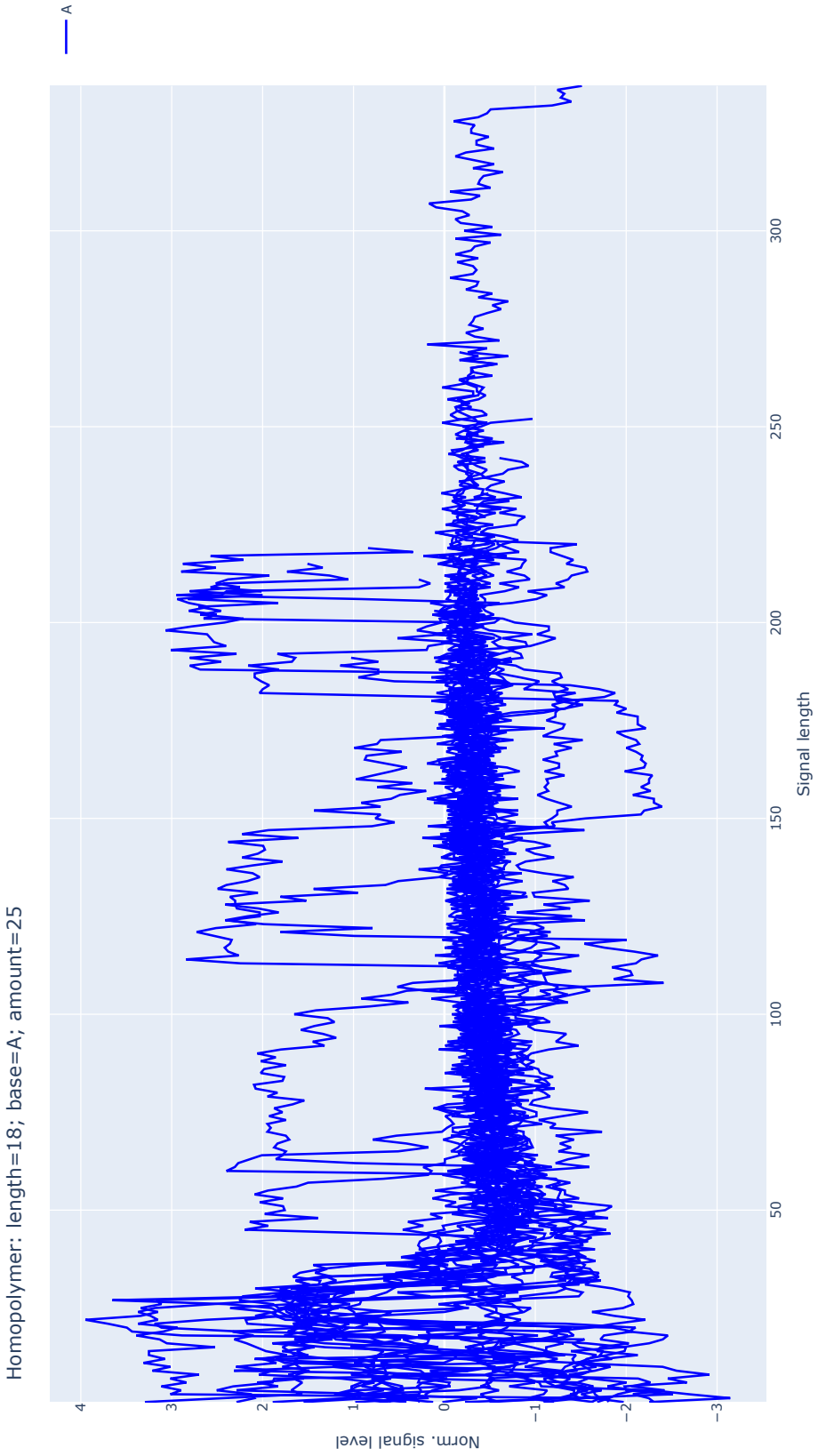


Figure 2.2: Normalized signal segments of 25 18*A*s in the Loman Lab Mock Community[24] dataset.

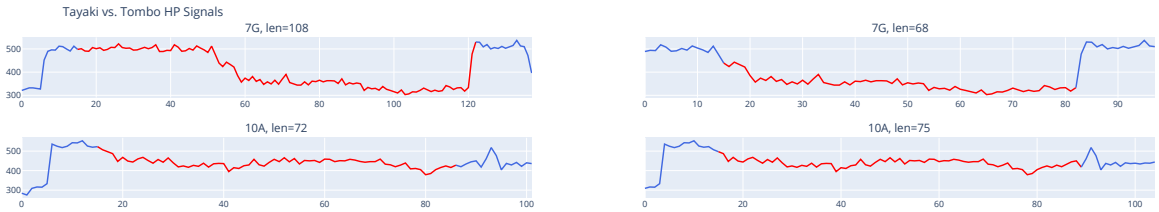


Figure 2.3: Homopolymer signal from Taiyaki and Tombo. The 7G spans 40 signals more according to Taiyaki than Tombo. On the 10A there is more agreement.

and 10X according to Tombo, while excluding reads with longer homopolymers. The reason to exclude homopolymers longer than 10, is that our bioscience contacts told us that even the ground truth references are unreliable for these these lengths [2].

Performance-wise, Taiyaki's data preparation is very slow: processing these 18,491 reads took around 3 days, whereas Tombo could easily process over 2 million reads in the same time. For this reason we first used Tombo to select reads for preprocessing with Taiyaki, as processing over 2 million reads with Taiyaki would be infeasible.

We found that there were minor differences between Taiyaki and Tombo, when comparing the homopolymers found in the 18,491 reads:

- Taiyaki and Tombo find roughly the same amount of homopolymers (Appendix A.1).
- They produce similar homopolymer boundaries. It is hard to tell which one is more precise (Appendix A.2). In some cases there is disagreement between Taiyaki and Tombo on homopolymer region of the signal (Figure 2.3).
- Mean signal lengths of homopolymers between the two differ by about 5 signals in most cases. For 10C's and 10G's the mean signal difference is 20 (Appendix A.3). This may be due to Taiyaki finding 65 (10C) and 61 (10G) compared to 18 (10C) and 16 (10G) found by Tombo.

2.3.1 Signal Length Distributions

Appendix A.3 shows that there is a wide range of possible signal lengths for each homopolymer. For example, a 4A can have the same signal length as a 10A.

Appendix A.3 also reveals some extreme outliers. For instance, the signal length of $6X$ ranges from 66–2309 signals, though with a median of 122. With varying signal lengths without significant change in signal level, there is not much useful information for a basecaller to make its call. Considering thousands of nX, outliers are bound to happen and they do not seem to significantly influence the distribution. Indeed, there is a noticeable upward trend for the signal length as the homopolymer length grows. However, there is still a significant overlap in signal-lengths for adjacent nX.

2.4 Translocation Speed

In the previous sections we found that there was a significant overlap of signal-lengths for homopolymers. This might be due to a varying translocation speed throughout the read, s.t.

there are periods where the DNA translocates faster through the nanopore, whereby events of the same base may vary in length. This section will investigate, if any patterns of the translocation speed is measurable.

There are multiple reasons for why varying translocation speed, such as the temperature of the solution, number of times the solution is reused, and the stochastic motion the DNA itself [5]. Additionally, the translocation speed of a single read may also change throughout the sequencing, such that sections of the DNA sequence is sequenced faster or slower than other sections. A common reason for DNA slowing down and speeding up throughout a single read is that the DNA sequence might get entangled with nearby molecules or itself. Such an entanglement slows down the translocation speed, possibly either through friction or from having to break the bonds of the entanglement [40].

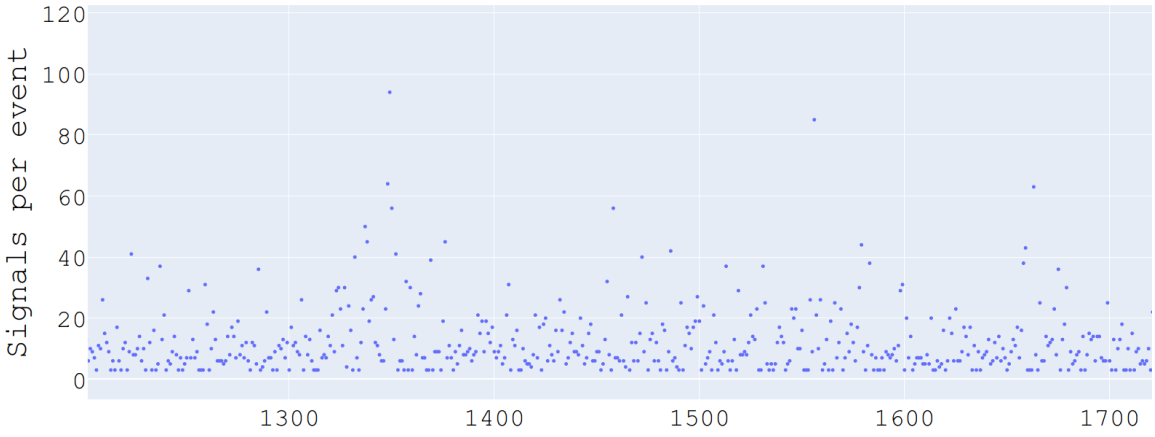


Figure 2.4: Number of signals per event produced by Tombo for 400 events

To analyze the translocation speed of a DNA sequence (a read), we will use the resquiggle algorithm of Tombo [37] to align signals to the reference genome. To reiterate, the resquiggle algorithm produces events where each event consists of $(start_{pos}, end_{pos}, base)$. This enables us to examine how the number of signals per event (SPE) changes throughout a read.

A visualization of the number of SPEs is shown in Figure 2.4, (a larger version of this figure is shown in Figure B.1) where the y-axis denotes the number of signals per event, and the x-axis is index of the events in chronological order.

The average number of SPE is 11.62, and ∼63% of all events are 11 SPE or less, while the mode SPE is 3 with ∼16%, and the variance of SPE was 92.19. A plausible reason for why 3 signals per base is the most common in the events, is that the default minimum number of signals per base is set to three in Tombo.

When examining the visualization an upwards trend can be seen from around event 1310 to event 1360, where multiple events occupy a significantly higher number of signals than the rest, though shorter events are interspersed that break the trend. Figure 2.4 does not indicate a global trend, nor any significant local trends, therefore we will abandon this idea of translocation speed inference for now.

However, it may be still be a worthwhile endeavor to further investigate the translocation speed, as we have only briefly examined the translocation speed of a few reads with discouraging results. It is possible that developing a method for such may very well be an entire project in itself.

2.5 Summary

In this section we investigated (1) if the length of homopolymers and the length of the corresponding signals correlate, (2) compared homopolymeric regions of Tombo and Taiyaki, and (3) studied the translocation speed of Tombo events. To summarise the findings of this chapter, the following lists the most important findings:

- Overrepresentation of nA and nT, may result in underwhelming performance in prediction of nC and nG.
- Extreme outliers of signal lengths is subject of great concern.
- Overlap of signal-lengths for adjacent nX is worrisome.
- An upward trend for the signal length as the homopolymer length grows is evident, with $\sim$12 signals added per base.
- Changes in the supposed unchanging homopolymer tail occasionally occur.
- Tombo and Taiyaki showcase minor differences their production of events, and it is not clear which selects the best homopolymer boundaries.
- No apparent pattern in variations of translocation speed were found from investigating Tombo events in isolation.

In Section 2.2 we found that the average signal-length increase per additional base in a homopolymer was 12.122 signals. Similarly, in Section 2.4 we found that the average signals per event (in Tombo) was 11.62. These values are nearly equal, and indicates a trend. However, as these are averages they hide the underlying variance in the data, thus, a naïve statistical approach for inferring homopolymer lengths from signal lengths would probably produce unreliable results. Therefore, we will not pursue such an end.

Chapter 3

Preliminaries

In this chapter will we explore some of the preliminaries needed to understand the different machine learning modules used in the following chapters.

The following sections were copied from our previous report [29] (some with minor modifications): Linear Layer, Rectified Linear Unit, and Convolution.

3.1 Training a Neural Network

This section is based on chapter 5 and 6 of Deeplearning by I. Goodfellow [11].

This section will detail the training of a typical feed forward neural network. We will explore the formal mathematics for the forward and backwards pass for training a neural network, as well as examine the gradient based learning method called backpropagation that is used throughout the deep learning field. Finally, we will go through a simple example of how a feed forward neural network would update the weights for minimizing an objective function $\mathcal{L}$.

Training of a neural network is generally separated into two different phases. The first phase is the forward pass where the neural network outputs a prediction based on an input. The second phase updates the parameters of the network based on an error calculated from the output of the forward pass and the corresponding ground truth.

We can think of a training a neural network as an optimization problem, whereby minimizing the objective function maximizes a performance measure P. Here optimization refers to the task of either minimizing or maximizing some objective function $\mathcal{L}$. The function to be optimized is called the objective function and it describes the relationship between the label for a training example and the prediction produced by the neural network.

3.1.1 Forward Pass

The first phase is a simple process of matrix multiplication and matrix addition. We define the inputs to the network as a column vector $\mathbf{X} = [x_1, \cdots, x_n]^T$, and associated with each layer a weight matrix $W^{(l)}$ and a bias column vector $b^{(l)}$.

We can define the output of a layer as $Z^{(l)}$ shown in Equation 3.1

$$\mathbf{Z}^{(l)} = \mathbf{a}^{(l-1)} \cdot W^{(l)} + \mathbf{b}^{(l)} \tag{3.1}$$

where $\mathbf{a}$ is the the activations from the previous layer, or $\mathbf{X}$ for the first layer, defined as:

$$\begin{aligned} \mathbf{a}^{(l)} &= \sigma^{(l)}(\mathbf{Z}^{(l)}), \\ \mathbf{a}^{(1)} &= \mathbf{X}. \end{aligned} \tag{3.2}$$

Here $\sigma^{(l)}$ is an arbitrary activation function for layer l. The output of the neural network would then be the activations of the last layer $\mathbf{a}^{(L)}$.

3.1.2 Backward Pass

The method used for updating the parameters of a neural network is called backpropagation, and it aims to minimize the objective function by adjusting the weights and biases in the network by backpropagating an error backwards through the network. The backward pass needs an objective function to describe the error produced by the network by comparing the target for the training example and the output of the network. A simple loss function is the mean squared error:

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2, \tag{3.3}$$

where $\hat{\mathbf{y}}$ is the prediction produced by the network for a training example and y is the label for that example.

Traditionally when updating the weights of a neural network, the error calculated by the objective function is the average error over all training examples. The calculation of the gradients for the entire dataset is computationally expensive, because it requires evaluating the model on every example in the entire dataset. A typical optimization used to speed up the training of the network is to define batches where a batch contains N unique random sampled elements from the training set. We would then feed forward each training example from the batch and finally calculate the average error for the batch. The approach of batching the training examples makes convergence faster (in terms of total computation time, not in terms of number of model updates), because we are able to update the weights of the network more frequently [11, p.278].

Updating the Parameters

To update the weights and biases of the network, backpropagation uses gradients that show the direction of the error, thus, a parameter x needs to change in the opposite direction of its gradient to minimize the objective function $\mathcal{L}$.

The equation for backpropagation is shown in Equation 3.4 where $\mathcal{L}$ is the objective function, e.g. mean squared error.

$$\frac{\partial \mathcal{L}}{\partial W^{(L)}} = \frac{\partial \mathcal{L}}{\partial \alpha^{(L)}} \cdot \frac{\partial \alpha^{(L)}}{\partial Z^{(L)}} \cdot \frac{\partial Z^{(L)}}{\partial W^{(L)}}, \tag{3.4}$$

where the subequation are defined as follows:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \alpha^{(L)}} &= 2(y - \hat{y}),\\ \frac{\partial \alpha^{(L)}}{\partial Z^{(L)}} &= \sigma'(Z^{(L)}),\\ \frac{\partial Z^{(L)}}{\partial W^{(L)}} &= \alpha^{(L-1)}.\end{aligned} \tag{3.5}$$

To see from where these equations are derived, we draw the computational graph shown in Figure 3.1, where the arrows describe the parameters used for calculating the function at the arrowhead. So, the objective function $\mathcal{L}$ takes as parameters the target y and the output of the network's last layer: $\alpha^{(L)}$, where L is the last layer of the network.

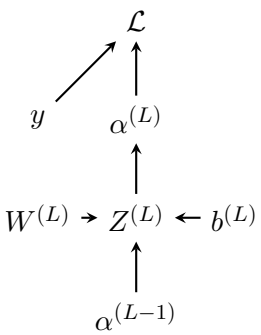


Figure 3.1: An example of a computation graph, where L is the objective function and $a^{(L)}$ is the output of the network.

The computational graph is only shown for the last layer in the network, but its preceding layer $\alpha^{(L-1)}$ follows the definition for $\alpha^{(L)}$ but one layer deeper, i.e.

$$\alpha^{(L-1)} = \sigma'(Z^{(L-1)} \cdot W^{(L-1)} + b^{(L-1)}).$$

3.2 Optimization by Gradient Descent

Machine learning are optimization problems, and though the specific task to learn is only limited by human ingenuity, the learning is generally done with gradient descent.

Gradient descent is an iterative optimization algorithm, and it aims to minimize an objective function $f(\theta)$ w.r.t the parameters $\theta \in \Theta$ of a given model. This is done by updating the parameters in the opposite direction of the gradient of the objective function w.r.t to the parameters, i.e. $\nabla_\theta f(\theta)$[1] [35]. The task of gradient descent can be formulated as $\theta^* = \arg\min_{\theta \in \Theta} f(\theta)$. This minimizer is found through a sequence of updates w.r.t. the entire training dataset by

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta f(\theta). \tag{3.6}$$

[1] $\nabla_\theta f$ is a vector of all first-order partial derivatives of f w.r.t. to the parameters θ, and describes the gradient surface of the function.

However, performing gradient descent across the whole dataset is slow as it must calculate the gradient of the entire dataset exactly. Instead one can use Stochastic Gradient Descent (SGD), which approximates the gradient on the basis of a single training example [4, 35], i.e.

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta f(\theta, x^{(i)}, y^{(i)}). \tag{3.7}$$

This is usually much faster, though the loss may fluctuate until a minimum is reached. This fluctuation of SGD also allows the exploration of several local minima, whereas vanilla gradient descent may get stuck in a local minimum. Though this also makes it possible for the optimizer to overshoot, so it is customary to decrease the learning rate to minimize the fluctuations as SGD (hopefully) approaches the global minimum [35].

Mini-bath gradient descent is an amalgamation of both batch gradient descent[2] and stochastic gradient descent, whereby it performs an update every n training examples:

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta f(\theta, x^{(i:i+n)}, y^{(i:+n)}). \tag{3.8}$$

This has the nice properties it is less fluctuating than SGD — which leads to more stable convergence — and remains faster than batch gradient descent [35]. Often the term SGD is used, when in fact, mini-batch gradient descent is the algorithm employed. Following this widely adopted practice, we will also use SGD to refer to mini-batch gradient descent.

Still, several other gradient based optimizers exists, and we will briefly cover two widely used alternatives in the following.

3.2.1 Adadelta

Adadelta [45] is an extension of another optimization algorithm called Adagrad [9]. In short, Adagrad adapts the learning rate of the parameters by performing large updates for infrequent parameters and smaller updates for frequent parameters. This is done by accumulating all past squared gradients per parameter. The update rule for Adagrad is as follows:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\sum_{\tau=1}^{t} g_{\tau,i}^2}} g_{t,i}, \tag{3.9}$$

where $g_{t,i}$ is a shorthand for the gradient of parameter i at time step t [9], i.e.

$$g_{t,y} = \nabla_{\theta_t} f(\theta_{t,i}). \tag{3.10}$$

Adagrad has the drawback of continual decay of learning rates throughout training. This is evident by the denominator in Equation 3.9, which becomes infinitesimally small after many iterations.

Adadelta improves on Adagrad in two major ways: 1) gradients are accumulated over a fixed window, and 2) removes the need for a manually defined learning rate [35, 45]. In Adadelta, the sum of gradients is defined as an exponentially decaying average of the past squared gradients in the previous w time steps [45]. If we assume at time t this running average is $E[g^2]_t$ then we compute

[2] Vanilla gradient descent is also known as batch gradient descent.

$$\begin{aligned} E[g^2]_t &= \rho E[g^2]_{t-1} + (1-\rho) g_t^2, \\ E[\Delta g^2]_0 &= 0 \end{aligned} \tag{3.11}$$

where ρ is a decay constant similar to the one used in momentum. The square root of this is required in the parameter update rule, and effectively becomes the Root Mean Squared error (RMS) of the previous squared gradients up to time t:

$$RMS[g]_t = \sqrt{E[g^2]_t + \epsilon}, \tag{3.12}$$

where ϵ will be used to later eliminate the possibility of division by zero. The resulting parameter update becomes:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t. \tag{3.13}$$

[45] notes that *"if the parameter had some hypothetical units, the changes to the parameter should be changes in those units as well."* However, this is not the case with this equation, so [45] also defines another exponentially decaying average, this time of the squared parameter updates:

$$\begin{aligned} E[\Delta\theta^2]_t &= \rho E[\Delta\theta^2]_{t-1} + (1-\rho)\Delta\theta_t^2, \\ E[\Delta\theta^2]_0 &= 0, \end{aligned} \tag{3.14}$$

likewise due to the square root, this effectively becomes an RMS:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}. \tag{3.15}$$

$RMS[\Delta\theta]_t$ is unknown, so it is approximated with the RMS of parameter updates until the previous time step. By replacing the learning rate in Equation 3.13 with $RMS[\Delta\theta]_{t-1}$ we get the update rule of Adadelta [35, 45]:

$$\begin{aligned} \Delta\theta_t &= -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \tag{3.16}$$

which has eliminated the learning rate. Adadelta is easily computable and provides a learning rate per dimension, and even eliminates the need for a manually selected learning rate.

3.2.2 Adam

Adaptive Moment Estimation (Adam) [19] also computes individual adaptive learning rates for different parameters. Like Adadelta, Adam stores an exponentially decaying average of past gradients m_t and the squared gradient v_t, and uses two parameters $\beta_1, \beta_2 \in [0, 1)$ to control decay rates of the averages.

$$\begin{aligned} m_t &= \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t \\ v_t &= \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2 \end{aligned} \tag{3.17}$$

m_t and v_t are estimates of the first moment (mean) and the second moment (variance) of the gradient respectively. These are initialized as vectors of zeros, and the authors note that this results in a bias towards zero in initial time steps, especially when the decay rates are small (i.e. when the βs are close to). This initialization bias is corrected by calculating bias-corrected estimates from the biased moment estimates [19]:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{3.18}$$

With this, the update rule of Adam is given by:

$$\theta_t = \theta_{t-1} - \frac{\eta \cdot \hat{m}_t}{\sqrt{v_t} + \epsilon} \tag{3.19}$$

Adam is computationally efficient and requires little memory, and is aimed at machine learning problems with large datasets and/or high-dimensional parameter spaces.

3.3 Linear Layer

A feedforward network is the simplest form of neural network. It takes an input and feeds it forward without any complex transformations. In its most common form, every node is connected to every node at the previous layer. The activation of a single node depends only on the previous layer and its connections to the current node. Figure 3.2 shows an illustration of a simple network and how a node's activation is computed.

The activation of a node is computed as a weighted sum of the output of the previous layer along with a *bias* in the form of b_0 (usually implemented as an always-active weight). The weights are the parameters that the network learns during training. The resulting weighted sum and bias are supplied to an activation function σ, and this is then the activation of the current node. An activation function can be either linear or non-linear depending on which function it represents [28].

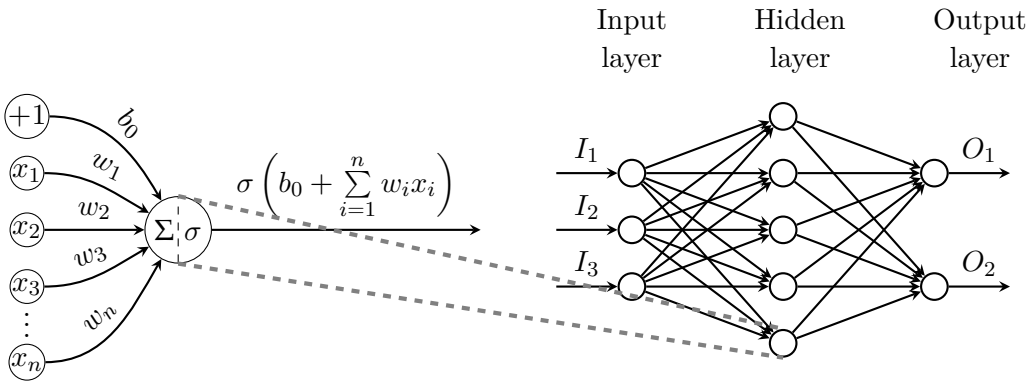


Figure 3.2: Activation of a node in a feedforward neural network.

A linear layer is one of the simplest layers for a neural network. Such a layer uses an activation of the form

$$\mathbf{y} = W * \mathbf{x} + \mathbf{b},$$

where W is the weight matrix and $\mathbf{b}$ is the bias for the layer. This is simply the weighted sum plus the bias as seen in Figure 3.2, and the plot for this is seen in Figure 3.3.

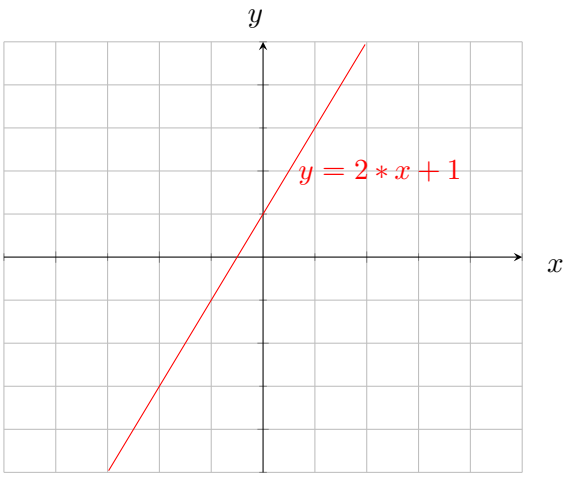


Figure 3.3: Plot of the linear function $y = 2 * x + 1$

3.4 Rectified Linear Unit

Rectified Linear Unit [12], or ReLU, is an activation function often used in neural networks. It is defined as $y = max(0, x)$ and its plot is illustrated in Figure 3.4. With ReLU units are effectively "dead" when they output 0. This was inspired by biological neurons where many connections may be "dead" at a given time. ReLU has proven better than sigmoidal activation functions, both being easier to train and easier to compute [10].

However, if $x \leq 0$ then the gradient is 0, meaning backpropagation can be difficult. Several variants of ReLU exists to combat this, but the simplest is leaky ReLU and it is defined as

$$y = \begin{cases} x & \text{if } x \geq 0 \\ \frac{x}{a} & \text{if } x \leq 0 \end{cases}$$

where a is a fixed parameter in the range $(1, -\infty)$ [44]. This allows a small positive gradient and "dead" units are avoided.

3.5 Convolution

In Section 2.4 we wrote that the DNA translocates through the nanopore at an inconsistent speed and that five bases influence the signal observed. That is, a single signal value is actually influenced by the five bases that reside in the nanopore at the time of observation. Due to the inconsistent speed and the fact that a signal is influenced by surrounding bases,

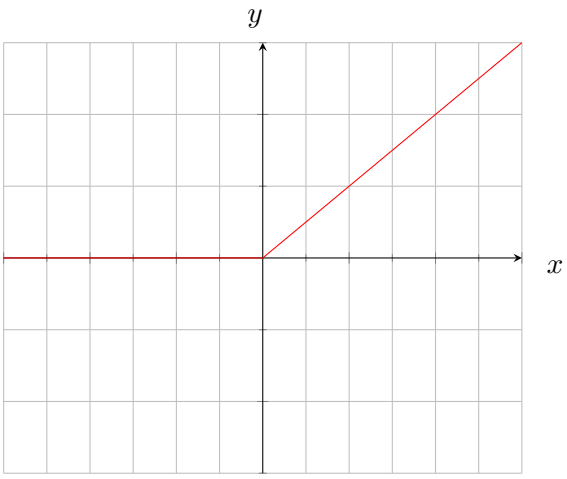


Figure 3.4: Plot of the ReLU activation function.

it may be beneficial to consider a local group of signals. One way to learn features of a group of signals, is by convolving the signals with a kernel large enough to capture the signals that constitute a base, and may also account for varying speeds. The theory of convolutions is the subject of this section.

A 1D convolution over an input $\mathbf{x}$ and a *kernel* $\mathbf{k}$ is defined as,

$$\mathbf{y}_j = \sum_{m=0}^{|\mathbf{k}|-1} \mathbf{x}_{j+m}\mathbf{k}_m, \tag{3.20}$$

where $\mathbf{y}_j$ is the j^{th} output, $\mathbf{x}_{j+m}$ is the input value at position $j+m$, and $\mathbf{k}_m$ is the kernel value at position m (from [11, p. 342]). That is, for each position in the input, j, we sum over the product of each kernel value multiplied by the input value in increasing order. An example is provided in Figure 3.5.

$$\left(\boxed{\underset{x1}{0}\ \underset{x0}{1}\ \underset{x1}{1}}\ 1\ 0\ 0\ 0\right) * \left(\boxed{1\ 0\ 1}\right) = \left(\boxed{1}\ 2\ 1\ 1\ 0\right)$$

$$\left(0\ \boxed{\underset{x1}{1}\ \underset{x0}{1}\ \underset{x1}{1}}\ 0\ 0\ 0\right) * \left(\boxed{1\ 0\ 1}\right) = \left(1\ \boxed{2}\ 1\ 1\ 0\right)$$

Figure 3.5: Two steps of a 1D convolution with a kernel of size 3.

3.5.1 Stride

A minor modification to convolutions is to modulate its *stride*, i.e. how much the kernel is moved at every step A large stride will skip over more of the input and reduce the dimensionality, effectively downsampling the input. Here we multiply a constant, s, with the position j, obtaining

$$\mathbf{y}_j = \sum_{m=0}^{|\mathbf{k}|-1} \mathbf{x}_{j \cdot s+m} \mathbf{k}_m, \tag{3.21}$$

where s is the stride [11, p. 343]. An example is provided in Figure 3.6.

$$\begin{pmatrix} \underset{x1}{0} & \underset{x0}{1} & \underset{x1}{1} & 1 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 1 & \underset{x1}{1} & \underset{x0}{1} & \underset{x1}{0} & 0 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \end{pmatrix}$$

Figure 3.6: Two steps of a strided 1D convolution with a stride of 2.

3.5.2 Input and output channels

It is also possible to define multiple kernels for a convolution. Each kernel convolves the input and produces a *channel* each. One can think of kernel as representing different feature-extractors, and channels as representing different features same object. For example, the same base may be predicted from signals that look somewhat similar. Thus being able to learn different features from the same signal can be beneficial because a kernel may be able to learn base-specific features encoded in multiple channels.

A 1D convolution with input channel l, and output channel i is computed by

$$\mathbf{y}_{i,j} = \sum_{l=0}^{|L|-1} \sum_{m=0}^{|\mathbf{k}|-1} \mathbf{x}_{l,j+m} \mathbf{k}_{i,l,m}, \tag{3.22}$$

where $\mathbf{y}_{i,j}$ is the j^{th} position in output channel i; $\mathbf{k}_{i,l,m}$ is the kernel value at output channel i, input channel l, and kernel position m; $\mathbf{x}_{l,j+m}$ is the input value at position $j+m$ in input channel l. Equation 3.22 is from [11, p. 342].

Equation 3.22 states that every output channel must have a kernel for each input channel, i.e. $i * l$ is the number of kernels. For instance, if we have two input channels and two output channels, we need two kernels for each output channel, making it four kernels in total. See Figure 3.7 for an example on a 1D input.

3.5.3 Padding

When an input is convolved, the output dimensionality will be reduced by $kernel\_size - 1$ (assuming a stride of 1). Increasing the stride of the convolution further reduces the dimensionality. Naïvely using many convolutional layers will ultimately reduce the dimensionality to the point where only a single value remains as output. One could use a smaller kernel size and fewer layers, though this will lower the expressiveness of the network [11, p. 343]. Also, note that not all kernels element will be multiplied to the boundary elements without padding. Vital features may therefore not be learned if the kernel does not cover the whole input.

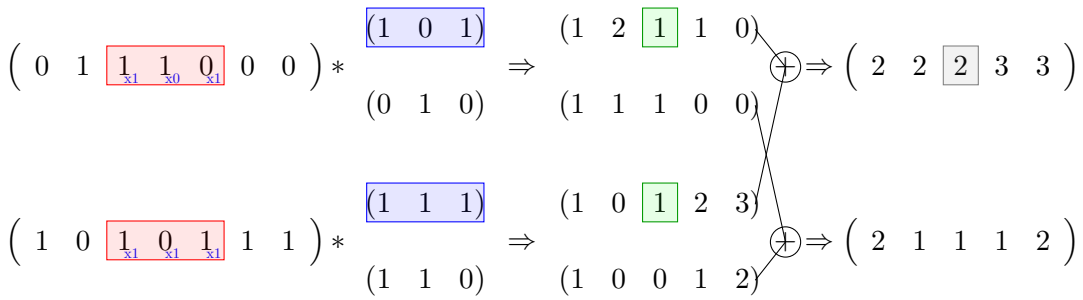


Figure 3.7: 1D convolution with two output channels and two input channels.

To alleviate such issues, one can apply zero-padding to the edges of the input. Padding simply adds extra elements at each side of the input for the convolution such that the kernel may cover a large space, we shown an example of zero-padding in Figure 3.8.

$$\left(\boxed{\mathbf{0}_{x1}\ \mathbf{0}_{x0}\ 2_{x1}}\ 3\ 2\ 5\ \mathbf{0}\ \mathbf{0}\right) * \left(\boxed{1\ 0\ 1}\right) = \left(\boxed{2}\ 3\ 4\ 8\ 2\ 5\right)$$

$$\left(\mathbf{0}\ \boxed{\mathbf{0}_{x1}\ 2_{x0}\ 3_{x1}}\ 2\ 5\ \mathbf{0}\ \mathbf{0}\right) * \left(\boxed{1\ 0\ 1}\right) = \left(2\ \boxed{3}\ 4\ 8\ 2\ 5\right)$$

Figure 3.8: 1D convolution with zero-padded input.

3.5.4 Transposed Convolution

In an encoder-decoder architecture it is often desirable to upsample the encoded latent space representation back to the original input size, e.g. in the case of an autoencoder. There exists typical algorithm approaches for such an upsampling, such as bilinear interpolation which computes each output $y_{i,j}$ from the nearest four inputs by a linear map that depends only on the relative positions of the input and output cells [30, p. 123-124]. An obvious disadvantage of bilinear interpolation is that the output is based entirely on the cell values of the encoded sequence, which might introduce artifacts that are undesirable.

A more elaborate method for upsampling an encoded sequence is to use a traditional convolution operation, but in reverse, where given the output and a kernel construct the input sequence. This operation is often called transposed convolution [25] and is useful because it allows a neural network to construct a non-linear upsampling.

An example of a transposed convolution can be seen in Figure 3.9, where each value in the input is multiplied with the kernel and outputs a vector of size 5.

$$\begin{pmatrix} 1 & 1 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$
$$= \begin{pmatrix} 1 & 1 & 2 & 1 & 1 \end{pmatrix}$$

Figure 3.9: Tranposed 1D convolution upsampling 3 inputs to 5 using a kernel size of 3 and stride of 1.

3.6 Pooling layer

Like the convolution operation, pooling consists of a fixed sized window that slides across the entire input field according to the specified stride parameter. A pooling function replaces the outputs of a layer in a neural network at a certain location with a *summary* of the nearby output. For example the widely used max pooling operation outputs the maximum output within a rectangle, thereby downsampling the input [11, p. 335]. An example of the max pooling operation with size 3 and stride 1 is shown in Figure 3.10.

$$\begin{pmatrix} 1 & 1 & 2 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 2 & 2 \end{pmatrix}$$

Figure 3.10: 1D max pooling layer with size 3 and stride 1.

3.7 Batch Normalization

Batch Normalization is a module often used in neural network architectures to accelerate deep network training by reducing the internal covariate shift. Bath normalization builds on the premise that the output distribution of neural network layers changes drastically in initial epochs. In the paper [16] they call this change the "Covariate shift" and they define it as the change in the distribution of the network's activations made by changes in the network parameters.

To reduce the internal covariate shift [16] uses a simple normalization operation known as Z-score normalization that produces zero mean and unit variance. After the Z-score normalization, two learnable parameters are added γ and β. These enable the network to shift and scale the normalized output to any desired distribution.

The batch normalization algorithm is shown in Algorithm 1, where the mini-batch is used to calculate the mean output activations, as is shown in step 1. The mean is then used to calculate the variance for each output in the mini-batch. When the mean and variance are calculated we can normalize the entire mini-batch using typical Z-score normalization.

Algorithm 1: The batch normalization algorithm from [16]

Input: Values of x over the mini-batch: $B = x_{i...m}$ Parameters to be learned γ, β
Output: $y_i = BN_{\gamma,\beta}(x_i)$

$$\mu B \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i)$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu B)^2$$

$$\hat{x_i} \leftarrow \frac{x_i - \mu B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i \leftarrow \gamma \hat{x_i} + \beta = BN_{\gamma,\beta}(x_i)$$

The paper [16] shows that by introducing the batch normalization module in a neural network architecture, they are able to utilize a significantly higher learning rate (10 times higher than the original training of the architecture) and still produce similar and sometimes better testing accuracies with fewer training epochs.

3.8 Autoencoder

Generally, one employs an autoencoder when an efficient encoding (typically by dimensionality reduction) of the input is desired. This is done by the autoencoder network learning to extract and preserve only the most relevant parts of the input while ignoring "noise".

An autoencoder consists of two main parts: an encoder that maps the input into a small encoding, and a decoder that maps the encoding into a reconstruction of the original input [11, ch. 14]. The training of the autoencoder then aims to minimize the reconstruction error between the input and its reconstructed output. Mathematically an autoencoder can be described as the composition of two functions, such that

$$\begin{aligned} E &: \mathcal{X} \to \mathcal{F} \\ D &: \mathcal{F} \to \mathcal{X} \\ E, D &= \arg\min_{E,D} L(X, (D \circ E)(X)), \end{aligned}$$

where L is the loss function for the reconstruction error.

Chapter 4

Development

Contemporary basecallers do not accurately basecall the length of homopolymers, they can, however, basecall non-homopolymers with very low error-rates [42]. We therefore propose a solution where the signal is segmented into HP and non-HP signals. The HP signals are handled separately by our solution while the non-HP signals are accurately basecalled with a contemporary basecaller, an approach similar to [39].

Figure 4.1 shows our proposed pipeline, where HP signals are detected and basecalled separately from non-HP signals. Input to the pipeline is a signal, which then follows the arrows in the diagram to the different components:

1. HP signals are detected and segmented from the signal among other non-HP signals.
2. HP signals are assigned a base and a length.
3. Non-HP signals are basecalled using a contemporary basecaller.
4. Basecalled non-HPs and HPs are combined to form a full base sequence.

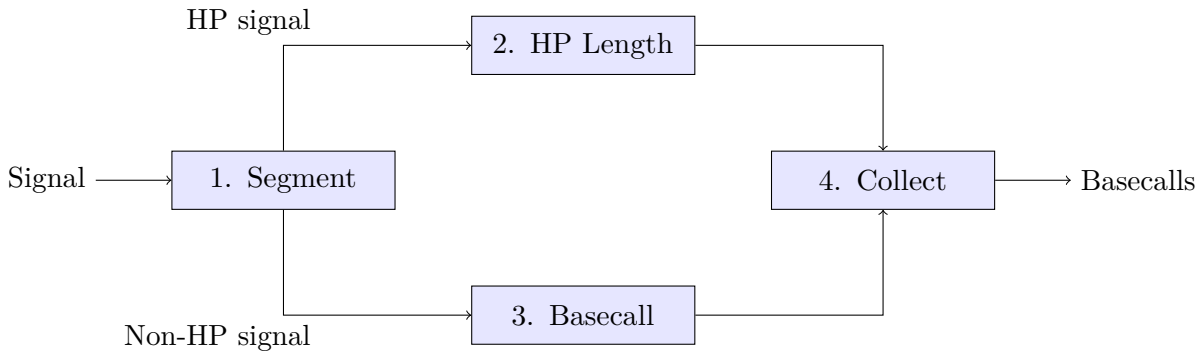


Figure 4.1: Pipeline for calling HP signals.

The following chapter will detail development in pursuit of our proposed solution. We have tried different methods for the parts of the pipeline, each of which is explained in the following, starting with the segmentation of signals into homopolymer and non-homopolymer segments.

4.1 Segmentation of Homopolymer Signals

The segmentation model takes as input signals produced by the nanopore sequencer and produces a binary mask for each signal. The binary mask corresponds to whether a signal is a part of a homopolymer or not.

4.1.1 Architecture

The architecture is based on an image segmentation architecture called U-Net [34], which has produced state-of-the-art results in image segmentation tasks. U-Net operates on 2D data, however, our data is 1-dimensional, so we will adopt the U-Net architecture to 1D. The architecture is shown in Figure 4.2 and is given the name U-Seg to avoid confusion with the U-Net model. The idea behind the architecture is to downsample the input sequence multiple times to create a dense feature representation and then upsample the representation back to the input length to produce a prediction for each element in the sequence. A similar approach has been used by [46] which showed promising results, but instead of producing a binary mask for each element in the input sequence, they produce a probability for each base at each time stamp.

4.1.2 U-Seg

The U-Seg architecture consists of an encoding path (left) and an decoding path (right) as shown in Figure 4.2. The number inside each rectangle is the length of the output sequence and the number under each rectangle is the number of output channels.

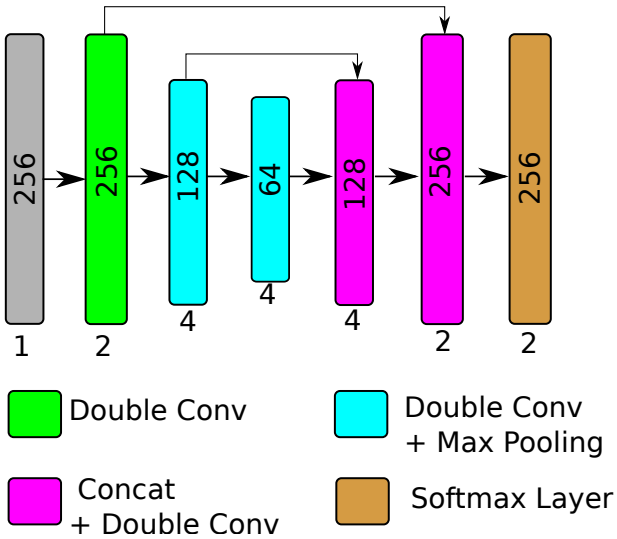


Figure 4.2: The U-Seg architecture. The number in each block corresponds to the sequence length and the number under each rectangle corresponds to the number of output channels.

The encoding path consists of multiple downsampling blocks, where the sequence length is halved for each block while the number of channels are doubled. The intuition behind downsampling the sequence multiple times is to construct a dense feature representation.

After downsampling the sequence an additional downsampling block (described in Section 4.1.3) is added, which does not construct additional channels, but only halves the

sequence length. The U-Net paper describes this block as a "bottle neck" that is meant to refine the downsampled features. We can define the "bottle neck" layer as

$$BN : \mathbb{R}^{C_{in} \times S} \rightarrow \mathbb{R}^{C_{out} \times S/2}$$

Where C_{in} and C_{out} are equal, and S is the segment length. After the downsampling procedure the encoded sequence is upsampled using the upsampling blocks (described in Section 4.1.3).

Lastly, for converting the output logits of the final upsampling layer to class probabilities we add a LogSoftmax layer, which outputs two log probabilities for each element in the sequence. LogSoftmax is defined as:

$$LogSoftmax(o_i) = log\left(\frac{exp(o_i)}{\sum_j exp(o_j)}\right),$$

we can define the last layer as $R^{C_{out} \times S} \rightarrow R^{C_{label} \times S}$, where $C_{label} = 2$ for the two classes, non-HP and HP.

4.1.3 Double Convolutional Block

The double convolutional block (Figure 4.3) is used in all layers throughout the network except the last, which is the LogSoftmax layer. The double convolution block, as the name suggests, applies a convolution operation twice. Once for the input to the layer and once for the output of the first convolution. The double convolution block is used for constructing additional channels, where a sequence length of S with C_{in} channels constructs a new sequence still of length S with C_{out} channels. Formally, let $DC : \mathbb{R}^{C_{in} \times S} \rightarrow \mathbb{R}^{C_{out} \times S}$ be the function to encode the signals of a sequence with C_{in} channels into a new sequencewith C_{out} channels.

DC is defined as

$$DC = Conv_2(Conv_1(S))$$

where S is the input sequence for the DC module, $Conv_1$ is the first convolution operation and $Conv_2$ is the second convolution operation.

The U-Net paper [34] uses a block similar to the double convolution block. Since their architecture is based on image segmentation they use a 2D convolution operation, whereas we use a 1D convolution operation as we are working with one-dimensional sequential data. Additionally, the U-Net paper uses the REctified Linear Unit (ReLU) activation function, whereas we instead opted to use the leaky ReLU activation function (both are explained in Section 3.4). We decided to use the leaky ReLU activation based on the paper [44], which shows that the leaky ReLU activation function either outperforms or performs similar to the typical ReLU activation function in all of their experiments.

To accelerate training we added a batch normalization layer (see Section 3.7) after each activation function (two batch normalization layers in total). These batch normalization layers should allow a slightly higher learning rate without compromising training accuracy. The paper [26] shows that adding the batch normalization layer after a non-linear activation function slightly improves test accuracy for a number of different convolutional neural networks.

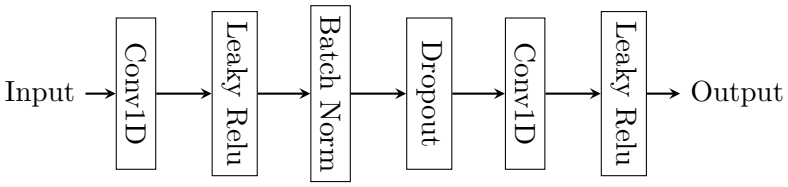


Figure 4.3: Double convolution block

Downsampling

To downsample the input sequence, we use the double convolution block previously described, where a max pooling layer (see Section 3.6), with a kernel size equal to 2, is applied to the output of the double convolution block. A figure of the downsampling module is shown in Figure 4.4.

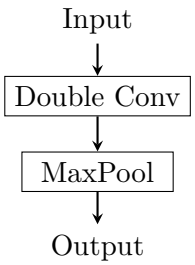


Figure 4.4: Downsample block

The max pooling layer reduces the sequence length S by half. We can define the downsampling modules as:

$$DS = \mathbb{R}^{C_{in} \times S} \rightarrow \mathbb{R}^{C_{out} \times S/2}$$

When multiple downsampled blocks are stacked we can encode the input sequence into a yet denser feature representation.

Upsampling

As previously mentioned the output of the segmentation module is not just a single prediction for the entire input sequence, but rather a prediction for each element in the input sequence. Thus, we must upsample the latent space produced by the downsampling blocks back to the input sequence length. We can define the upsampling module as

$$US = \mathbb{R}^{C_{in} \times S/2} \rightarrow \mathbb{R}^{C_{out} \times S},$$

where US is defined as:

$$US = DC(DS_{in} + trans_{conv}(US_{in})).$$

Here DS_{in} is the output of the previous downsampling block and US_{in} is the output of the previous upsampling block. If it is the first iteration of upsampling, the US_{in} is the output of the "bottle neck" layer. The symbol $+$ is the concatenation operation where the channels of the two sequences are concatenated, s.t. the result of the concatenation operation outputs

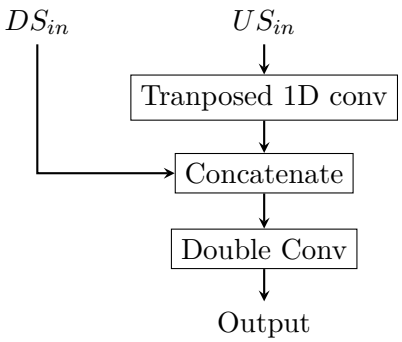


Figure 4.5: The upsampling block. US_{in} is the output of the previous upsampling module and DS_{in} is the output of the downsampling layer at the corresponding depth.

a sequence of length S with double the number of channels. The upsampling procedure is shown in Figure 4.5.

The number of channels in US_{in} and DS_{in} is always equal but the sequence length S of DS_{in} is twice the length of US_{in}, so we utilize the transposed 1D convolution operation (described in Section 3.5.4) with a stride of 2 to upsample the sequence such that the sequence length k of US_{in} and DS_{in} are of equal length.

4.1.4 Dataset Construction

When preparing a training dataset for the U-Seg model, we need to consider the sparsity of homopolymers in each read from the [24] dataset. Otherwise, the U-Seg model might only predict non-homopolymers simply due to the abundance of those. To combat the sparsity of homopolymers we prepared a balanced dataset from a larger training dataset with an equal number of homopolymer and non-homopolymer segments Each segment corresponds to 256 signals with their corresponding 256 labels. Additionally, we remove homopolymers longer than 10 bases (see Section 2.3).

4.1.5 Hyperparameter Tuning

The U-Seg architecture has several hyperparameters. In this study we will investigate which hyperparameters yields the highest training and testing accuracy.

- Down Kernel size: The kernel size used by the double convolutional blocks in the downsampling layers.
- Up Kernel size: The kernel size used by the transposed convolutional layers in the upsampling layers.
- Number of Channels: Used in the downsampling of the U-Seg. We only control the number of channels in the downsampling part of the architecture, because the upsampling has to use the same number of channels, otherwise a dimensionality mismatch occurs.
- Number of downsampling + upsampling layers: The number of times the sequence is downsampled and upsampled.

All these hyperparameters should influence the accuracy of the model, but in the following study we will show that the hyperparameters does not influence the accuracy of the model substantially. Initial experiments showed that the kernel size used for the transposed 1D operation in the upsampling layers did not influence the accuracy of the model, so for the following experiments we will keep the upsampling kernel size constant with a value of 7.

The notation we will use for the downsampling layers and upsampling is $[in_{channels}, out_{channels}]$. An example of this could be

$$\begin{aligned} Downsampling &= [2,4],[4,8] \\ Upsampling &= [16,4],[8,2] \end{aligned}$$

The $in_{channels}$ in the upsampling is always twice the size of the previous $out_{channels}$ due to the upsampling procedure, where two sequences are concatenated together which doubles the number of channels.

4.1.6 Results

To train and test the different U-Seg models we will use a total of 1000 reads in the training set, with the sampling method described in Section 4.1.4. The testing set consists of 500 reads but without an equal number homopolymer and non-homopolymer segments. Likewise, the evaluation dataset consists of 500 reads. It should be noted, that the segments in the training and testing set are shuffled, so all segments in a batch might not originate from the same read, but in the evaluation dataset the accuracies are calculated per read.

The distribution of segments is shown in Table 4.1. Each model will be trained for a total of 15 epochs.

Dataset	Hp segments	Non-hp segments
Training	26846	26846
Test	15235	70433
Evaluation	1286	6532

Table 4.1: Number of training, testing, and evaluation segments containing homopolymers and non-homopolymers

When calculating the segmentation accuracy, we first have to convert the probabilities produced by the LogSoftmax layer to indices (0 or 1), for which we use the Argmax function. To calculate the accuracy, we compare the target sequence with the predicted sequence and count the correct predictions in each class and divide it with the total occurrence in each class. A small example of the accuracy calculation can be seen in Equation 4.1, where T is the target sequence and P is the prediction sequence produced by the model.

$$\begin{aligned} T &= [0,0,0,0,1,1,1] \\ P &= [0,0,1,1,1,1,1] \end{aligned} \tag{4.1}$$

The accuracies can be seen in Equation 4.2

$$\begin{aligned} \text{Acc}_{\neg HP} &= 2/4 = 0.5 \\ \text{Acc}_{HP} &= 3/3 = 1.0 \end{aligned} \tag{4.2}$$

To calculate the accuracy for evaluation, another method is used for converting the log probabilities of the network to class indices. This is due to a limitation of the argmax function in that it simply selects the best candidate for each time step in the sequence. Choosing the best candidate at each time step might be suitable for some of the time steps, but it may produce sub-optimal sequence predictions as a whole.

Instead, we use the beam search algorithm, which selects multiple candidates for a sequence at each time step based on the conditional probability. The number of alternative paths depend on the beam width B, so for each timestamp beam search selects the B candidates with the highest probability.

For the output sequence we select the predicted sequence with the highest probability found by beam search. After a probable path has been found, we change non-homopolymer predictions under a certain length to a homopolymer. This property is useful because non-homopolymer mistakes may occur inside a homopolymer region where a single element is wrongly predicted. For the final step we remove noticeable homopolymer mistakes where the length of a predicted homopolymer sequence is below a minimum homopolymer length threshold. I.e. the model might predict a homopolymer sequence of only 5 signals, which is highly unlikely, so we change the predicted homopolymer sequence to a non-HP sequence. For simplicity we define this process in three distinct steps.

1. Find most probable path.
2. Remove Zero mistakes.
3. Remove HP regions if below threshold.

A high-level overview of the post-processing algorithm can be seen in Algorithm 2.

Algorithm 2: Post processing of homopolymer mask

Input: *predictions*: Segmentation output for each segment of shape(n, 2)
HP_threshold: Minimum homopolymer length
zero_mistakes: Maximum number of non-HP predictions that should be changed to HP predictions.

Output: Most likely sequence: $seq \in \langle x_n | n \in \{0, 1\} \rangle$

```
propable_segmentation = beam_search(predictions)
// compressed_predictions is an array of tuples, where the first element is the
   prediction and the second is the number of consecutive like predictions
compressed_predictions = run_length_encode(propable_segmentation)
foreach (prediction, length) ∈ compressed_predictions do
    if prediction == 0 ∧ length ≤ zero_mistakes then
        prediction = 1
// the collapse functions collapses repeated homopolymer predictions to a single list
   i.e. [(1,5),(1,1),(1,5)] → [(1,11)]
compressed_predictions = collapse(compressed_predictions)
foreach (prediction, length) ∈ compressed_predictions do
    if prediction == 1 ∧ length ≤ HP_threshold then
        prediction = 0
seq = uncompress(compressed_predictions)
return seq
```

The *HP_threshold* was set to 15 based on the results of Section 2.4, which shows that even though the mean number of signals per event is 11.62 the minimum number of signals per event could be 3. So for a homopolymer of five bases, the minimum number signals should be at least 15 ($3 \cdot 5 = 15$). The *zero_mistakes* was 2, and the beam width was 10 for all experiments.

The results can be seen in Appendix D where Table D.1 shows the testing and training result with multiple different hyperparameters. The Table D.2 shows the evaluation of the different models. Table D.1 shows that the hyperparameters do not seem to influence the accuracy of the model significantly. This can be seen when comparing the smallest model (run #0) with the largest model (run #8) where the smaller model actually performs slightly better than the larger. A similar trend can be seen in the evaluation table Table D.2 where the majority of the runs are within a few percent of each other. Run (#2) performs slightly better than the rest with a non-HP accuracy of ∼80% and HP accuracy of ∼18%, thus, we will use this model for further analysis.

For the last experiment we evaluated *zero_mistakes* with varying sizes. The *zero_mistakes* parameter may influence the accuracy of both homopolymers and non-homopolymers because it may change the predicted sequence drastically. For evaluating the *zero_mistake* parameter we used the model from run #2 and tested it with 500 reads from the evaluation dataset. As shown in Table 4.2 the non-HP accuracy decreases when the *zero_mistakes*

Zero Mistakes	Non-HP Acc	HP Acc	Average Acc
0	0.969	0.018	0.4935
1	0.946	0.042	0.494
2	0.796	0.193	0.4945
5	0.663	0.329	0.496
10	0.553	0.443	0.498

Table 4.2: Evaluation of the zero mistake parameter using the model from #2 over 500 reads

parameter increases and as expected homopolymer accuracy increases. When none of the zero mistakes are corrected, the model predicts 97% of non-homopolymers correctly but only correctly identifies ∼2% of the homopolymers. Additionally, when zero mistakes up to 10 are corrected the model only identifies 55% of non-homopolymer but 44% of all homopolymers. We think that the optimal value of the *zero_mistakes* parameter is either 1 with ∼95% of non-homopolymers and ∼4% of homopolymers accuracy or *zero_mistakes* value of 2 with ∼80% of non-homopolymers and ∼19% of homopolymers accuracy.

4.1.7 Combination with Bonito

To combine the U-Seg model with Bonito, we had to make a slight modification to the output of the default Bonito model. The Bonito model outputs a tuple containing $(classes, lenghts)$, where the classes is the entire base prediction for a read, and the *lengths* running number of signals per base prediction, e.g.

$$(\text{"}ACTAT\text{"}, [0, 3, 6, 21, 35, 21])$$

Here the first predicted base is A with a length of 3 signals ($0 - 3 = 3$), and the second base is a C with a length of 3 ($6 - 3 = 3$) also. To convert the Bonito predictions into a binary

mask we use a method similar to when constructing the training and testing dataset for the U-Seg model. In this method, homopolymers over the length of 5 are converted to ones and all predictions under 5 to zero.

To combine the U-Seg model and Bonito we simply iterate the two sequences in parallel, and if either of the two sequences predicts a homopolymer, we select it as the truth. The segmentation accuracies are shown in Table 4.3. An interesting observation can be seen from the Bonito predictions which only finds 3.7% of the homopolymers in the sequence, but 95.4% of all non-homopolymers, whereas the U-Seg models correctly finds ∼19% of the homopolymers, but only 80% of non-homopolymers. A possible reason for the low homopolymer accuracy produced by Bonito might be that the sequence lengths of Bonito might not overlap with the label sequence due to mismatches between the event sizes produced by Tombo and the output of Bonito.

Model	Non-HP Accuracy	HP Accuracy
U-Seg Beam Search	0.796	0.193
U-Seg Argmax	0.901	0.086
Bonito	0.954	0.037
Bonito + U-Seg Beam	0.784	0.208
Bonito + U-Seg Argmax	0.880	0.109

Table 4.3: The combined accuracy of the U-Seg model and Bonito model from 500 reads

When combining our U-Seg model with Bonito we can see that the homopolymer accuracy has increased by 1.5% with beam search and 2.3% with argmax compared to the standard U-Seg model. However, as seen in Table 4.3, the Bonito + U-Seg model with beam search introduces almost 22% homopolymers mistakes where non-homopolymers are predicted as homopolymers. Whereas the Bonito + U-Seg with argmax "only" introduces 12% wrong predictions for the non-homopolymers. These homopolymer mistakes will most likely introduce problems for the length prediction model and therefore the combination of U-Seg and Bonito should be selected based on the highest non-HP accuracy.

4.1.8 Summary

In summary the U-Seg model seems promising on training and testing sets with accuracies of ∼97% for non-homopolymer and ∼47% for homopolymer (see Table D.1 for run #2). However, when applying the model on the evaluation dataset the results seem underwhelming with non-homopolymer accuracy of ∼80% and homopolymer accuracy of 20% (with beam search and the post processing algorithm to remove zero mistakes)

4.2 Segmentation by Breakpoint Detection

Another approach we tried for the segmentation is based on the paper *Time Series Segmentation through Automatic Feature Learning* [22]. The paper proposes an unsupervised approach for *breakpoint detection* that uses an autoencoder (see Section 3.8 for feature extraction.

Figure 4.6 shows [22]'s breakpoint detection pipeline. The idea is that an autoencoder can automatically and effectively extract the unique features of the input without assumptions on its generation. These unique features are an encoding of the input in a dimensionally lower representation. The breakpoint detection builds on an assumption for the unique features: similar inputs have similar features, and dissimilar inputs have dissimilar features. Based on this assumption one can compare consecutive segments to determine their similarity, and if this similarity score is above a certain threshold it signifies a breakpoint in the input, i.e. a change has happened in the input.

This approach is attractive due to it being a general changepoint detection scheme, does not require a priori knowledge of the generative nature of the input, and because it is simple and intuitive.

Our idea with the method is to predict breakpoints, and then find homopolymer regions in the input based on the duration between breakpoints as long homopolymers should contain periods of unchanging signals.

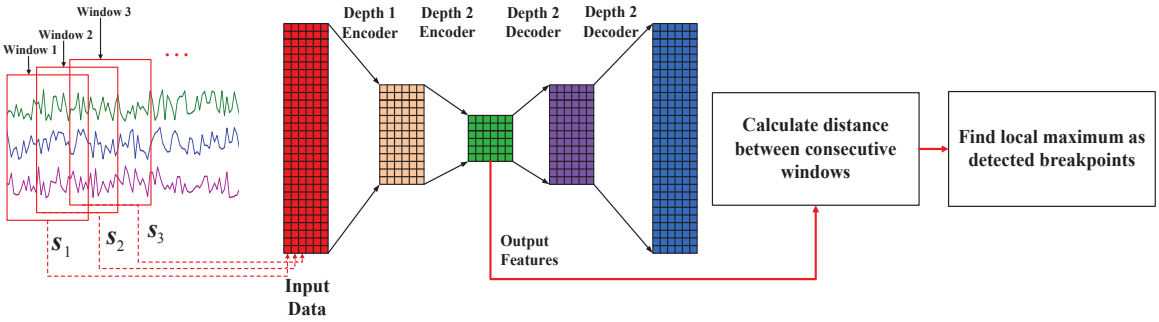


Figure 4.6: Breakpoint detection pipeline of [22]. Breakpoints are found from distance calculations between consecutive windows.

The method first splits the read into consecutive segments, that are then encoded and compared. Let R be a read that consists of a single channel across T timestamps. R is partitioned into windows of size N_w. Each window s_t $(t = 1, 2, \ldots, T/N_w)$ may overlap the following window s_{t+1} by O timestamps. The original read may be reformulated as $R = [s_1; s_2; \ldots]$, if the last O elements of each window s_t are removed, i.e. overlapped parts are removed from windows.

4.2.1 Architecture

[22] used a stacked autoencoder architecture, where each layer of the encoder finds increasingly lower-dimensional features of the input, conversely for the decoder. They found that the best results were achieved with two hidden layers, though this may be domain dependent, and we will test if this is the case for our data. The actual neural network architecture of the the autoencoder employed in [22] was a simple feedforward network, which can be seen in Figure 4.6. The window size varied per dataset, however, they used a constant ratio between the feature size and the input size of $dim(f_t)/dim(s_t) = 0.1$ (a factor 10 reduction), where f_t is the feature size of window s_t. This ratio may not be adequate for our dataset, hence, we will experiment with varying segment and feature sizes.

Like [22], we also used a stacked autoencoder layers, with mirrored encoder and decoder parts. However, we opted for a convolutional network instead of a feedforward network. The

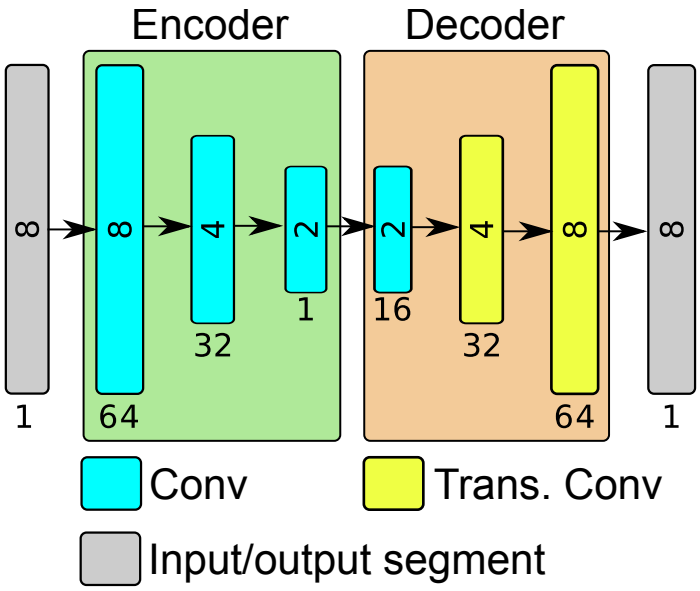


Figure 4.7: Architecture of the autoencoder. The number inside the rectangles are the segments size, and the number below is the number of channels

architecture of the autoencoder can be seen in Figure 4.7.

The encoder part uses convolutional layers with a kernel size of 3 and stride of 2 to halve the input size at each step, and a padding of 1. Conversely, the decoder uses transposed convolutional layers (which are covered in Section 3.5.4) for its upsampling, and these use a kernel size of 2 and a stride of 2. The first layer of the encoder has an input channel size of 1 to fit our 1D sequential data, and its last layer has an output channel size of 1. Likewise for the decoder. The architecture does not employ any activation functions after the convolutional layers. This choice was made as early experiments found that other activation functions inhibited convergence.

The architecture is adaptable, such that we can alter input size, number of layers, and number of channels per layer. However, this comes with some caveats. The encoder halves the segment at each layer, therefore the input size should preferably be a power of 2. Subsequently, the halving and kernel sizes of 3 with minimal padding, means that the smallest feature size is 2, and the number of layers to reach this size is equal to $log_2(segment\_size)$, which must not be exceeded due to the mentioned circumstances. The number of layers is indicated by a list of channel sizes, where each element indicates an additional layer with a desired channel size. For instance, the list $[64, 32, 16]$ would equal a network with 6 layers in total and a feature size of 2 given a segment size of 16. Note that the input, feature vector, and output all only have a single channel.

4.2.2 Breakpoint Detection

To find the supposed breakpoints in the signal, [22] uses a distance calculation of consecutive windows. We use the same distance calculation as [22], which calculates the distance between features f_t and f_{t-1} as

$$Dist_t = \frac{\|f_t - f_{t-1}\|_2}{\sqrt{\|f_t\|_2 \times \|f_{t-1}\|_2}}. \tag{4.3}$$

Our adaption of the approach used in [22] can be seen in Algorithm 3. The output of this algorithm is a set of indices indicating breakpoints.

Algorithm 3: Breakpoint detection algorithm adapted from [22].

Input: E: Encoder model
$segment\_size$: Size of segments
$overlap\_size$: Overlap size for segments
τ: Threshold for breakpoint detection.
Output: B: Set of detected breakpoints.

Calculate features $f_t = E(s_t)$ for all s_t.
Compute $Dist_t$ between consecutive features using Equation 4.3.
// Assemble distance curve s.t. a distance value spans its corresponding window ignoring overlaps. This is to correspond signal indices to distances (see Appendix C.1.1).
$span = segment\_size - overlap\_size$
foreach $Dist_t$ **do**
    $D = D \cup repeat(Dist_t, span)$
Calculate absolute difference along the distance curve, i.e. $diff[i] = \|D[i+1] - D[i]\|$.
foreach $diff[i]$ **do**
    if $diff[i] \geq \tau$ **then**
        Mark i as a breakpoint, i.e. $C = C \cup i$

Our approach differs from [22] in two major ways: 1) we use a convolutional network instead of a pure feedforward network, and 2) we use the absolute difference of consecutive distance values instead of finding local maxima on the distance curve. The first deviation was mostly made because convolutional networks have been widely used for segmentation tasks [3, 13, 17, 21] and for autoencoders [7, 41]. The second deviation was made in an effort to more efficiently find breakpoints, as the distances between the windows found in our data varies greatly, especially when the signal goes toward negative values.

Figure 4.8 shows an example of our breakpoint detection using the autoencoder approach, including events found by Tombo, which are used for comparison against found breakpoints. The model achieved a validation MSE loss of 491.6, and we can see that the reconstruction is not perfect, though it follows the original read very well. The breakpoints (vertical purple lines) were detected using a threshold of 0.5. Note that this threshold is a non-tuneable parameter, and should be tailored for each model to maximize the number of accurate breakpoints. We can see that for this example the model does not yield nearly as many breakpoints as events. This is in part because each signal does not have a unique distance value, which is because segments only overlap partially (see Appendix C.1.1 for more details). A unique distance value for each requires an overlap of $segment\_size - 1$.

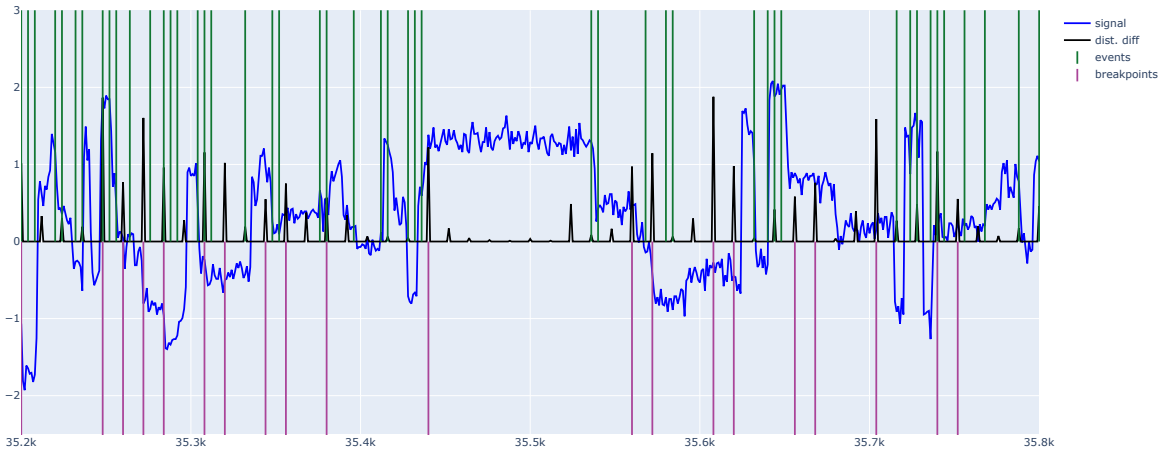


Figure 4.8: Example window of a read (blue) and the distance difference curve (black). The vertical lines denote events (green) and breakpoints (purple) that were found from a threshold of 0.5. The model used a segment size of 16, overlap of 4, and two channel sizes of 64 and 32 for a feature size of 4.

4.2.3 Experiments

Unfortunately, in our experimentation it was clear that the autoencoder approach was not as successful as we had hoped, as the breakpoints found did not align very well with the Tombo events. Additionally, it was not clear how to fairly evaluate found breakpoints, e.g. a distance-threshold between Tombo events and breakpoints would be necessary to decide if the breakpoint was a true positive or a false positive. Instead, we will give a more thorough investigation into the breakpoints detection of this approach to highlight the difficulties of the problem and shortcomings of the approach.

First, let us discuss hyperparameters that were varied between runs and why:

- **Segment size:** This is the size of the input and the output of the autoencoder. We do not think this will have a huge impact on the performance of the autoencoder, but might severely hinder the breakpoint detection. In Section 2.2 and Section 2.4 we found that a base spans ∼12 signals (but varies extremely), and if the segment size exceeds this value by too much, the model might have to encode signals that belong to three or more bases at a time. However, we are only interested in signals that span at most two bases, as breakpoints are at the border of a base-change.

- **Segment overlap:** The number of signals each segment overlaps the next by. The closer this number is to the segment size, the higher the resolution of the distance curve will be, i.e. if there is an overlap equal to $segment\_size - 1$ there will be a distance value for all but one timesteps, and every second timestep using overlap equal to $segment\_size - 2$. For convenience, we may use the term *non-overlap* to denote $segment\_size - overlap\_size$, so an overlap of $segment\_size - 2$ would be a non-overlap of 2. Also, if the overlap is too high, the distance between consecutive segments will probably change less than a smaller overlap, as the input changes less if consecutive segments overlap significantly.

- **Feature size:** Size of the encoder output. It may be the case that more dimensions provide smaller distances, as more hyperoctants enables better division of the latent

space. However, fewer dimensions would mean more distance between features, as they are more spread-out in the latent space. Because the segment size is constrained to values that are powers of two and each additional layer to the encoder halves the input, the feature size is also constrained to powers of two, a desired feature size thereby depends on the segment size and the number of layers in the encoder. For convenience we arbitrarily choose descending powers of two, starting from 64, for each additional layer.

The models were evaluated on 4 metrics:

- **Distances < 500:** Distance between breakpoint and nearest event. The purpose of this is to check how well the breakpoints align with the Tombo events. For why this statistic is split, see below. Higher count, lower mean, and lower standard deviation is better.
- **Distances ≥ 500**: The Distance measures are split in two, as, unfortunately, some reads have a last event that occurs long before the end of the read (expounded in see Appendix C.1.2). This is true if events have been found by either Taiyaki or Tombo, so it is inherent to the reads. This means that when finding nearest events for breakpoints that occur later than the last event in these reads, all the breakpoints gets matched to the last event, which are in rare cases more than 5000 signals prior to the breakpoint. 500 ended up being a suitable point of distinction, as most matches occurred well below 500, but including those up to 500 did not noticeably impact the statistics compared to 50, for instance. Lower count is better.
- **Breakpoint matches per event**: Depending on the distribution of breakpoints versus events, multiple breakpoints may share their individually closest event. This value should preferably be as close to 1 as possible, as that would indicate similar distributions of events and breakpoints. Mean should be close to 1, and lower standard deviation is better.
- **Value of breakpoint nearest to event**: This was meant to see if any insight into a usable threshold could be found. Preferably the value should have low variance, s.t. a suitable threshold is easily identifiable. This value is calculated by finding the nearest breakpoint candidate (a non-zero point on the distance difference curve) and recording the value. Higher mean and lower standard deviation is better.

All models were trained on 50 reads for 5 epochs, and evaluated on 50 reads. This was chosen as early experimentation revealed no improvement for either training loss or validation loss with neither increased number of reads or epochs. Training on more reads or for more epochs actually resulted in more jagged reconstructions that were judged visually. Therefore, this number of number of reads and epochs was a compromise between accuracy and time, as most models reached near-minimal losses after 2 epochs. Each read was segmented based on the segment size and overlap size, and the last segment was padded to match the desired segment size. The evaluation set contained a total of 3,910,867 signals and 342,599 Tombo events. Training and evaluation reads were from non-overlapping sets.

The loss function used to train the autoencoder was Mean Squared Error (MSE) loss, which is calculated as $MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$, where $\hat{Y}_i$ is the reconstruction of Y_i.

The threshold for breakpoint detection was set to 0.5 for all runs, and is the threshold used for evaluating the distance metrics and breakpoint matches per event. Preferably, this threshold should be tailored to each model to balance false positives and false negatives, but it was not clear exactly how to choose a proper value. 0.5 was chosen as it seemed to perform well in earlier experiments, however as subsequent experimentation has revealed, this value should be tailored to each model. "Breakpoints per event" and "nearest breakpoint value" do not use this threshold, as they only consider the nearest event-to-breakpoint-candidates and nearest breakpoint-candidate-to-events respectively. A breakpoint candidate is a distance difference value that is strictly positive and has not been filtered by any threshold.

Varying Segment Size

As mentioned the segment size is constrained to powers of two, therefore we tested on segment sizes of 4, 8, 16, and 32. We used a constant feature size of 2, which was achieved by adding the necessary number of stacked autoencoder layers. Likewise, the segments for each run used non-overlaps of 2. The interesting metrics for these runs are found in Table 4.4 (for complete metrics scores, see Table C.1).

model		distances<500	distances≥500	bps_pr_event	nearest_bp_val
4 : 2	count	503453.00	2441.00	240878.00	345299.00
	mean	4.53	3686.82	2.10	0.48
	std	13.38	2164.84	3.67	0.57
8 : 6	count	622141.00	3290.00	268392.00	345299.00
	mean	4.73	3636.11	2.33	0.59
	std	15.41	2138.80	4.66	0.59
16 : 14	count	366897.00	2087.00	195669.00	345299.00
	mean	5.03	3558.68	1.89	0.35
	std	15.26	2142.63	3.46	0.45
32 : 30	count	83638.00	424.00	50705.00	345299.00
	mean	4.34	3772.69	1.66	0.15
	std	14.13	2117.39	1.61	0.24

Table 4.4: Metrics for varying segment size for breakpoint detection. The values in the model column should be read as *segment_size* : *overlap_size*. For all metrics see Table C.1

In Table 4.4 we adopt the concise notation $S : O$, to denote segment size and overlap size, e.g. $8 : 6$ indicates a model with segment size of 8 and an overlap size of 6.

The $8 : 6$ model seems to be the best at distinguishing consecutive segments, as its *nearest_bp_val* is the highest. It also seems there is a near-steady decrease in *nearest_bp_val* as the segment size increases, though a slight increase is observed from the $4 : 2$ model to the $8 : 6$ model. This may be due to the fact that the non-overlap is kept constant at 2, which means as the segment size increases a smaller part of the whole segment is replaced at each timestep. This indicates that consecutive encoded segments become more alike as the segment size increases with a constant non-overlap size, which makes sense.

The $16 : 14$ model estimated almost the same number of breakpoints from distances under 500 (366,897) as the number of events (345,299), while retaining the second lowest

number of breakpoints above 500 (2,087). However, this model has the highest mean and second highest standard deviation for distances under 500.

The 32 : 30 model performed abysmal at breakpoint detection, only detecting 24.2% as many breakpoints as the number of events. Though this seems to be due to the constant threshold of 0.5 for breakpoint detection, as the *nearest_bp_val* column indicates a smaller value might be more appropriate for this model.

This experiment clearly shows that consecutive encoded segments become more alike as the segment size increases with a constant non-overlap size, however, it also becomes more difficult to discern small changes in consecutive segments (as indicated by the decreasing values in the *nearest_bp_val*). As mentioned, we believed that segments sizes exceeding 12 signals might decrease performance, and though the 16 : 14 model performs well, the 32 : 30 model clearly does not. In conclusion, the sweet spot for segment size seems to be around 8–16, and we will use 16 in the following experiments.

Varying Overlap Sizes

For this experiment, all models will use a constant segment size of 16 and channel sizes of 64 and 32 for a feature size of 4. The overlap sizes tested will be 0, 4, 8, and 14. some metrics for these runs can be seen in Table 4.5 (for complete metrics see Table C.2).

model		distances<500	distances≥500	bps_pr_event	nearest_bp_val
16 : 0	count	113776.00	388.00	105063.00	345299.00
	mean	5.92	3574.87	1.09	0.57
	std	13.27	2167.24	0.89	0.48
16 : 4	count	139513.00	457.00	122198.00	345299.00
	mean	5.81	3601.75	1.15	0.51
	std	12.62	2135.19	0.98	0.42
16 : 8	count	164462.00	511.00	135474.00	345299.00
	mean	5.45	3590.15	1.22	0.41
	std	12.43	2105.64	1.09	0.33
16 : 14	count	103189.00	546.00	67080.00	345299.00
	mean	5.26	3460.32	1.55	0.18
	std	16.34	2126.45	1.74	0.19

Table 4.5: Metrics for varying overlap size for breakpoint detection. The values in the model column should be read as *segment_size* : *overlap_size*. For all metrics see Table C.2

In this experiment it is more difficult to see the impact of changing the overlap size. The number of breakpoints found are very close. Still, we can see a decrease in *nearest_bp_val* from a mean of 0.57 to 0.18 and standard deviation from 0.48 to 0.19 as the overlap size increases. This means that not only do the values nearest to the breakpoint candidates' distribution become narrower, but there is also an increase in breakpoint candidates per event as the overlap size increases. This is probably due to breakpoint candidates only occurring every $segment\_size - overlap\_size$ signals (see Appendix C.1.1), and so if the overlap size increases, there should be more candidates. Again, this may also in part be impacted by the constant threshold of 0.5.

Varying Feature Size

The next experiment will test the impact of different feature sizes. The models will use a segment size of 16 and an overlap of 4. The feature sizes that will be tested are 8, 4, 2, and 1. As mentioned, each additional layer halves the input, therefore we need 1, 2, 3, and 4 layers respectively to reach the target feature sizes with the segment size of 16. The channel sizes for these layers will, as previously, be powers of two descending from 64. The interesting metrics for these runs are shown in Table 4.6 (see Table C.3 for a complete list of the metrics).

model		distances<500	distances≥500	bps_pr_event	nearest_bp_val
Feature: 8 Channels: 64	count	88389.00	109.00	80286.00	345299.00
	mean	5.57	3364.73	1.10	0.36
	std	11.03	2122.62	0.44	0.28
	50%	4.00	3296.00	1.00	0.30
	95%	16.00	6896.00	2.00	0.89
	max	492.00	7412.00	68.00	6.42
Feature: 4 Channels: 64 32	count	144626.00	471.00	126712.00	345299.00
	mean	5.70	3659.07	1.15	0.53
	std	12.13	2125.13	0.99	0.44
	50%	4.00	3716.00	1.00	0.42
	95%	16.00	7070.00	2.00	1.34
	max	484.00	7388.00	226.00	6.88
Feature: 2 Channels: 64 32 16	count	165401.00	610.00	144557.00	345299.00
	mean	5.59	3667.79	1.15	0.65
	std	13.43	2123.47	1.19	0.59
	50%	4.00	3758.00	1.00	0.51
	95%	16.00	7058.60	2.00	1.62
	max	496.00	7424.00	304.00	18.48
Feature: 1 Channels: 64 32 16 8	count	217326.00	865.00	189297.00	345299.00
	mean	5.05	3568.98	1.15	1.56
	std	12.51	2146.78	1.44	6.72
	50%	4.00	3428.00	1.00	1.10
	95%	16.00	7085.60	2.00	4.04
	max	492.00	7448.00	423.00	1912.35

Table 4.6: Metrics for varying feature size for breakpoint detection. The models all used a segment size of 16 and an overlap of 4.

Varying the feature size had the expected result of fewer dimensions showing more distance between features, as can be seen in the *nearest_bp_val* column. Here we see that not only does the mean increase when decreasing the number of feature dimensions, but the standard deviation also increases. Furthermore, we can see that the max *nearest_bp_val* more than doubles from 6.88 to 18.48 for a feature size of 4 to 2, however, from feature size 2 to 1 we see a drastic increase from 18.48 to 1912.35 This indicates that very few dimensions are not ideal, as a good threshold would be difficult to pinpoint.

The standard deviation and max in the *bps_pr_event* column also increases as we decrease the feature size, this however indicates more non-zero values in the distance curve as more breakpoint candidates are matched with the same event. The standard deviation should preferably be as low as possible, but we can also see that in this setup the 95th percentile is only at 2 across all runs.

4.2.4 Discussion

The outcome of this experiment can be summarized in three points:

- The segment size should not span too many signals, it should instead be close to the average span of an event.
- The overlap size impacts the number of breakpoint candidates. A larger overlap means more candidates, and vice versa. However, a large overlap means that consecutive segments are more alike and the distance between said segments is thus small. Though more candidates might also mean more false positives.
- Varying the feature size drastically impacts the distances of neighbouring segments, and a feature size of one should be avoided.

In conclusion, the approach of [22] was too attractive to pass by, but it does not seem to be a good match for breakpoint detection on nanopore reads. As we see it, this is because the data that [22] used was more stationary and uniform than Nanopore reads, which changes rapidly and only plateaus briefly. The plateaus of nanopore reads are the signals that correspond to the k-mer present in the sequencer, of which there are k^n possible values (5^4 for the R9.4.1 nanopore[1]), where n is the number of bases.

Secondly, Tombo and Taiyaki find events based on k-mer transitions (one plateau to another), whereas this breakpoint algorithm simply checks for large changes in the signal. However, the k-mer transitions may be slight as some k-mers do appear very similar, and these slight changes are apparantly difficult to detect with this approach.

The large number of possible values seems to interfere with the assumption *similar inputs have similar features, and dissimilar inputs have dissimilar features*, as the inputs of the reads seem to be too similar to reliably discern.

We did not reach a point where we thought segmentation of HP and non-HP segments was worthwhile, as the breakpoint detection was too unreliable. Therefore, we will use the U-Seg model for segmentation.

4.3 Homopolymer Length Classification

After the signals from a nanopore read has been segmented into homopolymer segment candidates by the U-Seg model (Section 4.1), the next step is to accurately classify the homopolymer candidates' respective lengths. This task may seem trivial as it resembles a simple supervised learning problem: train a neural network to predict the length of a homopolymer signal.

[1] 5^4 is actually a lower bound, as the sequencer happily sequences modified bases.

In practice, however, this is a difficult task, and in this section we will explore which model was trained to predict homopolymer lengths, tune the hyperparameters of the model, and examine and analyse the results from the model.

4.3.1 Architecture

The architecture consists of an encoder that encodes the input sequence into a dense latent representation, and a decoder that projects the latent representation into the length classes 1–10. In the following the encoder and decoder are formalized and tested for homopolymer length classification.

Encoder and Decoder

The encoder part of the architecture is a residual convolution network (ResNet) that encodes a signal into a latent representation. We use ResNet because it adds a simple yet powerful mechanism to convolutional networks: skip connections. Skip connections have been shown to speed-up or even enable convergence for some networks [14]. A bidirectional recurrent neural network was also tested as an encoder, but did not converge . A combination of convolutional layers and recurrent layers, however, was not tested.

Let $E : \mathbb{R}^S \to \mathbb{R}^{C_{out} \times S}$ be the function to encode a signal of length S into a new sequence with C_{out} features for each signal element. Let $e_i \in \mathbb{R}^{C_{out}}$ (for $i = 0, \cdots, S$) be the i^{th} encoded element by the encoder such that $(e_1, e_2, \cdots, e_S) \in \mathbb{R}^{C_{out} \times S}$. Here E is defined as

$$E = Res_1 \circ Res_2 \circ \cdots \circ Res_R$$

where each Res_i is a residual block (Figure 4.9). The first residual block Res_R has one input channel and C_{out} output channels, while the rest have C_{out} input and output channels. We set the number of input and output channels to be equal to avoid dimension mismatch with the element-wise addition between residual blocks. Otherwise the outputs of one block has to be projected to match the dimensionality of the following residual block.

Let $D : \mathbb{R}^{C_{out}} \to \mathbb{R}^{C_{label}}$ be the decoder that maps the encoder output to logits[2]. The decoder only takes the S^{th} encoded element, i.e the vector e_S, that forms a representation of the whole input signal. This avoids having to project all encoded elements $e_i, \cdots, e_S$ to $\mathbb{R}^{C_{label}}$. As a result, this simplifies the architecture. Early experiments were conducted where all encoded elements were projected with several linear layers, however this did not improve the validation error.

The decoder D is defined as a convolutional layer with a kernel size 1, C_{out} input channels, and C_{label} output channels that matches the number of classes from 1–10. Hence, D projects e_{out} into logits $d_{out} \in \mathbb{R}^{C_{label}}$. These logits are then passed to softmax to get probabilities for each class,

$$\text{Softmax}(\mathbf{o})_i = \frac{\exp(o_i)}{\sum_j^n \exp(o_j)}.$$

In practice, special care must be taken when taking the last vector from batched outputs. Signal lengths varies greatly, and signals are therefore padded to avoid dimension mismatch

[2]Logits are the outputs of the neural network before passing it to softmax/logsoftmax or argmax.

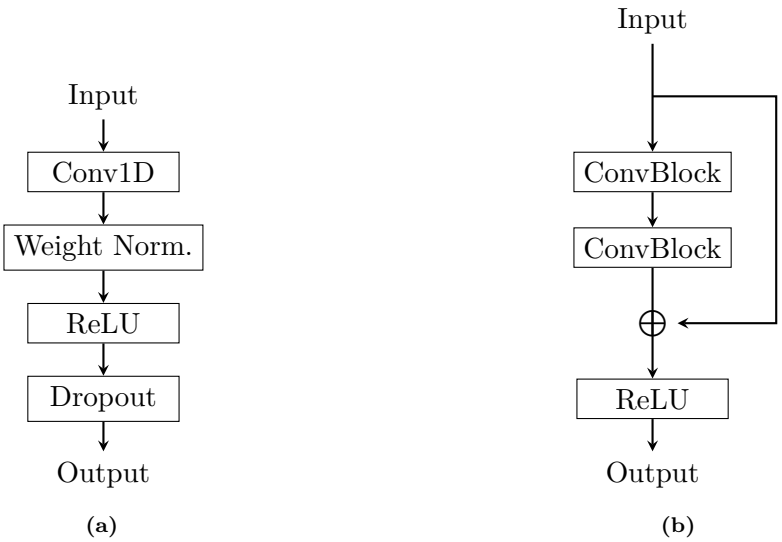


Figure 4.9: Residual block (b) composed of Convnet blocks (a). $\bigoplus$ is elementwise addition.

in a batch. Given a homopolymer segment, the encoder will output a feature vector with C_out entries for each signal element. This feature vector is an encoding of the signall segment. Algorithm 4 shows how the feature vector corresponding to the last signal element in a segment is selected from encoder outputs. Initial experiments showed that indexing this way gives better accuracy, in contrast to taking the very last output of every batch sequence.

Algorithm 4: Selecting the last sequence element in a batch of outputs.

Input: O_i: Outputs from decoder D (for $i = 1, \cdots, Batch_{size}$). The shape of O is $(Batch_{size}, max_{seq}, C_{out})$ where max_{seq} is the homopolymer segment in the training set with the maximum length.,
L_i: Lengths for homopolymer segments (for $i = 1, \cdots, Batch_{size}$).

Output: P: Last sequence element in each O_i.

```
P := []
foreach O_i, L_i do
    P.append(O_{i,L_i})
return P
```

Encoder and Decoder Parameters

It is worth mentioning how the different hyperparameters affect the total learnable parameter count. The majority of learnable parameters are located in the encoder. Whereas the decoder only consists of a projection layer to output classes. The amount of parameters for the encoder and decoder is given by:

$$P_{enc} = \underbrace{C_{out} \cdot K \cdot 2 \cdot R + C_{out}}_{enc} \underbrace{+C_{label}}_{dec} \tag{4.4}$$

Where:

C_{out}: is the number of output channels

K: is the kernel size

R: is the number of residual blocks

C_{label}: is the number of classes.

The factor of 2 is for two convolutional layers in a residual block. The additional parameters C_{out} is for projecting the input to the network from C_{in} channels (1 in this case) to C_{out}. This is due to the elementwise addition between the input and output of each residual block. Otherwise, the elementwise addition would result in dimension mismatch.

4.3.2 Objective Function

The objective for the network is to classify each homopolymer length. To that end, we used cross-entropy [31] to measure the similarity between logits $\hat{Y}$ and target classes $Y \in \{1, \cdots, 10\}^N$:

$$\begin{aligned} \mathrm{CE}(\hat{Y}, Y, \mathbf{w}) &= -\frac{1}{N}\sum_{i=1}^{N} w_{Y_i} \cdot \log\left(\mathrm{softmax}\left(\hat{Y}_i\right)_{Y_i}\right) \\ &= -\frac{1}{N}\sum_{i=1}^{N} w_{Y_i} \cdot \left(-\hat{Y}_{Y_i} + \log\sum_j \exp\left(\hat{Y}_j\right)\right) \end{aligned}$$

Where $\mathbf{w}$ are weights used to scale the loss for each class, in case of class-imbalanced data. For balanced data, these weights are set to 1 for all classes and $\mathbf{w}$ can be removed from the equation.

4.3.3 Sampling Strategies

The homopolymer dataset forms a long tail of sparse classes as the HP length increases (see Appendix A.4). Note that, homopolymers > 10 are removed from the dataset (see Section 4.4). These experiments were done early where signal segments for all 32 length classes (see Appendix A.4) were included.

To avoid giving frequent classes more importance than sparse classes, different avenues were pursued to train the network:

1. instance-based sampling: equal probability for each sample.
2. class-aware sampling (inspired by [18]): equal probability for each class.
3. loss weighting: a sample with class c is weighed by

$$min\left(\frac{most\ frequent\ class}{occurence\ of\ c}, 16\right).$$

Each weight is capped at 16 to avoid very large scaling factors.

These sampling strategies were used to train the network described in Section 4.3.1 on page 43.

4.3.4 Data Split

A 60/40 split for training and validation datasets is made over all HP data. The ratio is chosen to not miss samples from sparse classes in the dataset, and to have the validation set contain at least a few samples from the sparse classes.

In Section 4.4 a balanced sampler is used to split the data so that only a specified proportion of samples is taken from each class. This avoids holding out too many samples from the training set, as they would instead be in the validation set.

4.3.5 Training and Validation Results

With validation accuracy alone, instance-based sampling is the best (Table 4.7) with 31%. However, since we have 32 classes, the accuracy alone hardly tells us how well the models are able to classify the long tail of sparse classes.

Appendix E shows the confusion matrix on the validation data for each model. Class-aware sampling and loss weighting performs similarly for short homopolymers, with slight gaps in the predictions for loss weighting. For the homopolymer lengths 28–45, the loss weighting model predicts these as $6X$s possibly because the signals for these HPs can be the same length (Appendix A.4).

Name	Train Accuracy	Validation Accuracy
Instance Sampling	0.15	0.31
Class-aware Sampling	0.48	0.25
Log Weighting	0.32	0.25

Table 4.7: Training and validation accuracy on HP length dataset.

With class-aware sampling, the long HPs are actually predicted as being long, but they still do not lie on the diagonal. Moreover, it is seen that many predictions are off-by-one predictions, which makes sense since the signals for e.g. $6X$s can be the same length as $7X$s. Because there are only few samples for long HPs, these HPs gets oversampled making the model overfit, which is seen at the corner of the confusion matrix in Appendix E.

In summary, the class-aware sampling and loss weighting methods improves the accuracy of sparse classes and is simple in practice to employ. Although some shortcomings still apply. With loss weighting the model tends to misclassify long HPs as short. Class-aware sampling mitigates this issue but introduces the problem of overfitting on long HPs since they are oversampled.

4.4 Improving the Homopolymer Length Classification

Since training the model in the previous section, we have consulted our bioscience contacts [2], resulting in a slight change in the homopolymer dataset. The reference assembly for this dataset is acquired by sequencing with PacBio, which is more accurate and more expensive than using ONT's sequencers. However, according to Mads Albertsen from Albertsen Lab [2] told us that PacBio still suffers from inaccurate homopolymer calling when they exceed a length of 10.

Therefore in these next experiments only homopolymers of 10Xs and below are allowed in the dataset. Class-aware sampling now also becomes less of a problem as the class distribution is now more evenly spread out. There are still far fewer 10Xs than other HPs (Appendix A.1).

Continuing the improvement of training a model for homopolymer length classification, several ablation studies were conducted. The purpose of ablation studies are to both simplify the model by removing unnecessary features and to improve the accuracy.

4.4.1 Hyperparameter Tuning

The residual network has several hyperparameters that can be tuned. In this study we investigate which of these hyperparameters yield the best trade-off between a simple model (few layers and few learnable parameters) and a complex model (many layers and many learnable parameters). The following hyperparameters were tuned:

- Kernel size: the model's receptive field. Adjusting the kernel size allows the model to create kernel features from more of the signal at once.
- Number of residual blocks (each residual block has two convolutional layers).
- Channel size: the size of the latent representation for each input element.

All these hyperparameters influence the total amount of learnable parameters of the model (see Equation 4.4 on page 44). Initial experiments showed that varying the channel and kernel sizes throughout the network resulted in poor performance. Therefore, when setting channel size and kernel size, the same channel size and kernel size are is in all residual blocks.

4.4.2 Dataset Preprocessing

Several modifications to the preprocessing of the training data were also made:

- Padding modes: padding the homopolymer signal with:
  - Signals before the homopolymer (Backward).
  - Signals after the homopolymer (Forward).
  - Signals from before and after the homopolymer (Both).
- Padding size: amount of signals to use as padding in the corresponding padding mode.

The padding modes were to see if there is a specific side of the homopolymer that provides more information when predicting its length. In addition, padding size was adjusted to see how many signals are needed to give valuable features for length classification.

4.4.3 Ablation Results

The setup for all runs was:

- Dataset for tuning hyperparameters were capped at 1,500 homopolymer segments per length per base. Data splits were created with stratified sampling (see Table 4.8). Validation set was set to be 5% of the HP dataset for each length and base. Training set

was the remaining 95%, the data left after sampling the validation set. Homopolymers were sampled from 18,491 reads preprocessed with Taiyaki. The reads were shuffled before sampling.

- Number of epochs was set to 15. Experiments showed that running more than 15 epochs only improved validation error slightly or not at all. In Section 4.4.5 a model is trained over more epochs.
- Batch size was set to 512. Each sequence in the batch is padded with zeroes to match the maximum sequence in all samples to keep the same dimension in all batches. This makes dataset sampling easier as there are no dimensionality changes between batches.
- Learning rate was set to 0.01.
- Optimizer was Adadelta [45].

	Training set					Validation set			
	A	C	G	T		A	C	G	T
1–7	1425	1425	1425	1425	1–7	75	75	75	75
8	1424	181	159	1425	8	75	10	9	75
9	1425	254	258	1425	9	75	13	14	75
10	1424	62	58	1425	10	75	3	3	75

Table 4.8: Homopolymer distribution per length and base for training and validation sets.

Multiple metrics for determining the best model was used:

- Per base accuracy over all lengths: to see if there are any biases in the accuracies for each base.
- Average base accuracy (Bases avg.): a total score for per base accuracy by averaging.
- Average length accuracy (Length avg.): a total score defined by the proportion of correct classifications of each length and then averaging over all length accuracies. An analysis of per length accuracies of the final model for homopolymer length classification is conducted in Section 4.4.5. Length accuracies are excluded in this section to not pollute the result tables for hyperparameter tuning.

Hyperparameter Tuning Results

The size of the kernel has a significant impact on the accuracy of the model (Table 4.9). A kernel size of 5 results in poor accuracy (11% average base accuracy on validation set). In contrast, a kernel size of 97 improves the accuracy significantly (34.9% average base accuracy on validation set). It is also seen that Cs and Gs have slightly less accuracy. This may be due to the disproportionate distribution of Cs and Gs compared to As and Ts in the training set (Appendix A.1).

Keeping the kernel size of 97 fixed, two runs were made with with increased channel sizes (Table 4.10). Increasing the channel size from 4 to 16, gives a slightly smaller accuracy

(34.9% vs 34.7% average base accuracy on validation set). Setting channels to 64, gives a slight improvement. Also worth noting is that increasing the channel size also increases the training time significantly. Training over 15 epochs with 64 channels, 2 residual blocks, and a kernel size of 97 took 45 minutes on a Nvidia GeForce GTX 1070. In comparison, running the same setup with only 2 channels took only 3 minutes on the same hardware. Hence, there is a trade-off between accuracy and training time that is worth considering.

Increasing the amount of residual blocks tells a similar story (Table 4.11). Setting the kernel size to 97 again, and changing the residual blocks from 2 to 8 only provides a slight improvement, with the training time being 13 minutes.

Dataset	Kernel	A	C	G	T	Bases avg.	Length avg.
train	97	0.362	0.313	0.290	0.393	0.339	0.373
val	97	0.377	0.321	0.284	0.412	**0.349**	**0.386**
train	67	0.357	0.308	0.290	0.367	0.330	0.368
val	67	0.355	0.284	0.296	0.348	0.321	0.355
train	27	0.297	0.293	0.282	0.320	0.298	0.315
val	27	0.288	0.320	0.284	0.317	0.302	0.310
train	17	0.246	0.242	0.235	0.262	0.246	0.261
val	17	0.235	0.259	0.220	0.276	0.247	0.254
train	5	0.099	0.095	0.110	0.113	0.104	0.104
val	5	0.096	0.104	0.116	0.125	0.110	0.110

Table 4.9: Tuning kernel size. Channel size is 4, context size is 5 with mode forward-backward, and blocks is 2.

Dataset	Channels	A	C	G	T	Bases avg.	Length avg.
train	64	0.392	0.335	0.318	0.412	0.364	0.401
val	64	0.403	0.315	0.296	0.407	**0.355**	**0.400**
train	16	0.391	0.327	0.316	0.415	0.362	0.399
val	16	0.364	0.321	0.297	0.405	0.347	0.374

Table 4.10: Tuning channel size. Kernel size is 97, context size is 5 with mode forward-backward, and blocks is 2.

Dataset	Blocks	A	C	G	T	Bases avg.	Length avg.
train	8	0.385	0.322	0.301	0.409	0.355	0.392
val	8	0.368	0.337	0.289	0.415	0.352	0.388

Table 4.11: Tuning number of blocks. Kernel size is 97, channel size is 8, and context size is 5 with mode forward-backward.

Dataset Preprocessing Results

Adjusting the context and padding mode yields only minor improvements compared to other runs (Table 4.12).

The accuracy for each run shows also only minor differences. Also, note that in each run, the reads are shuffled before sampling the homopolymers and this may give a bias towards the accuracy between runs. This is seen by comparing the run with a context of 5 and with padding mode `both` with the top run in Table 4.9 that uses the same context and padding mode settings. These two runs should closely match in average base/length accuracy. Therefore, it may be difficult to tell whether improvements stems from shuffling the reads or the padding mode and size.

There is, however, the tendency that using a large context results in poor accuracy.

Dataset	Ctx	Mode	A	C	G	T	Bases avg.	Length avg.
train	25	Both	0.348	0.301	0.276	0.361	0.322	0.356
val	25	Both	0.333	0.323	0.282	0.324	0.316	0.344
train	15	Both	0.354	0.306	0.280	0.359	0.325	0.360
val	15	Both	0.391	0.308	0.273	0.373	0.336	0.381
train	5	Both	0.347	0.316	0.284	0.371	0.330	0.368
val	5	Both	0.348	0.313	0.265	0.349	0.319	0.356
train	0	Both	0.362	0.303	0.288	0.375	0.332	0.367
val	0	Both	0.355	0.313	0.303	0.357	0.332	0.374
train	25	F	0.359	0.306	0.286	0.374	0.331	0.365
val	25	F	0.336	0.289	0.276	0.372	0.318	0.361
train	15	F	0.362	0.316	0.293	0.382	0.338	0.372
val	15	F	0.344	0.296	0.288	0.373	0.325	0.364
train	5	F	0.371	0.320	0.288	0.387	0.342	0.379
val	5	F	0.393	0.343	0.284	0.381	**0.350**	**0.393**
train	0	F	0.358	0.301	0.275	0.388	0.331	0.371
val	0	F	0.381	0.295	0.258	0.381	0.329	0.365
train	25	B	0.348	0.293	0.260	0.357	0.314	0.352
val	25	B	0.364	0.284	0.274	0.356	0.319	0.358
train	15	B	0.350	0.284	0.273	0.363	0.318	0.356
val	15	B	0.324	0.313	0.260	0.387	0.321	0.357
train	5	B	0.362	0.327	0.292	0.378	0.340	0.375
val	5	B	0.349	0.324	0.319	0.383	0.344	0.373
train	0	B	0.368	0.315	0.294	0.385	0.341	0.376
val	0	B	0.363	0.297	0.295	0.383	0.334	0.369

Table 4.12: Different preprocessing of training data. Kernel size is 97, channel size is 4, and number of blocks is 2.

4.4.4 Length Regression

The problem of classifying homopolymer lengths can also be considered as a regression task. For this, mean squared error (MSE) was used:

$$\text{MSE}(\hat{Y}, Y) = \frac{1}{N}\sum_{i=1}^{N}\left(Y_i - \hat{Y}_i\right)^2$$

Where $\hat{Y} \in \mathbb{R}^N$ are logits, and $Y \in \{1, \cdots, 10\}^N$ are the target lengths.

The decoder is changed from the one used in classification to only give a single output. That is, let $D : \mathbb{R}^{C_{out}} \to \mathbb{R}$ where D is defined as a convolutional layer with a single output channel.

A multi-task setting was also considered that combines mean squared error and cross entropy:

$$L(\hat{Y}_{REG}, \hat{Y}_{CLASS}, Y) = \text{MSE}(\hat{Y}_{REG}, Y) + \text{CE}(\hat{Y}_{CLASS}, Y)$$

Where $\hat{Y}_{REG} \in \mathbb{R}^{C_{label}}$ is the output of the classification decoder, $\hat{Y}_{CLASS} \in \mathbb{R}$ is the output of the regression decoder, and $Y \in \{1, \cdots, 10\}^N$ are the target lengths. The decoders receives side-by-side the same encoded vector e_{out} from the encoder.

To get the length of a homopolymer from the regression network, the output is rounded. For the multi-task network, the length is taken with argmax over the classification decoder. The output of the regression decoder was not used in the final length classification.

Regression performs slightly worse than classification (Table 4.13), 32.4% versus 34.9% average base accuracy on the validation set. Similarly, the multi-task network is slightly worse than the regression network, making it also worse than classification with cross entropy.

Dataset	Task	A	C	G	T	Bases avg.	Length avg.
train	Reg. Class.	0.351	0.287	0.276	0.353	0.316	0.350
val	Reg. Class.	0.359	0.263	0.265	0.357	0.311	0.338
train	Reg.	0.347	0.298	0.264	0.352	0.315	0.342
val	Reg.	0.364	0.323	0.251	0.360	0.324	0.360

Table 4.13: Regression task and multi-task with regression and classification. Kernel size is 97, channel size is 4, blocks is 2, and context size is 5 with mode forward-backward.

4.4.5 Training a Model for More Epochs

In the results of the ablation study we found that a large kernel size and a large channel size were the biggest contributors to improving the validation error. These hyperparameter findings were then used to train a model with more samples in the training set, and over more epochs. The total data distribution is seen in Table 4.14.

Due to the sparsity of lengths 8–10, model accuracy on these lengths are poorer. This is the same problem as explored in Section 4.3.3 where the model does not learn to accurately classify minority classes. Therefore, we apply the same solution as in Section 4.3.3 where minority classes are oversampled in the training set to create a balanced dataset. One

	Training set					Validation set			
	A	C	G	T		A	C	G	T
1–5	19K	19K	19K	19K	1–5	1K	1K	1K	1K
6	19K	4180	4275	19K	6	1K	220	225	1K
7	19K	1520	1425	19K	7	1K	80	75	1K
8	10450	181	159	10450	8	550	10	9	550
9	9.5K	254	258	9.5K	9	500	13	14	500
10	3325	62	58	3325	10	175	3	3	175

Table 4.14: Homopolymer distribution per length and base for training and validation sets.

problem, first discovered after training, was that the 62 samples for homopolymer 10C (and the other minority classes) were also oversampled to 19K samples.

Hyperparameters for training this model were:

- Model is trained over 160 epochs.
- Channel size is 16.
- Number of residual blocks is 2.
- Dropout is 0.25. Initial experiments for this model showed slight overfitting, therefore dropout was added.
- Kernel size is 97.
- Batch size is 1024. More GPU memory was available, therefore the batch size was also increased from 512 to 1024.
- Optimizer is Adadelta.

Results

Training a model over more epochs and with more samples yielded a 44.8% average base accuracy on validation set, and a 44.1% average length accuracy (Table 4.15). Bases C and G have 20% higher accuracy than bases A and T on the training set. As stated above, minority classes are oversampled in the training set to match 19K samples, which is very likely the reason for this. Confusion plots on the training set (Appendix F) paints the same picture where Cs and Gs have high accuracy for length 8–10, which are the lengths with fewest Cs and Gs.

Dataset	A	C	G	T	Bases avg.	Length avg.
train	0.443	0.632	0.638	0.437	0.538	0.538
val	0.454	0.441	0.454	0.442	0.448	0.441

Table 4.15: Model trained over 160 epochs with 19K samples for each length and base in training set.

Interestingly, this problem does not seem to affect model performance, at least not when comparing the A and T accuracies with C and G accuracies in Table 4.15. All accuracies lie

around 44%–45%. The confusion plots in Appendix F does show a more spotted matrix for C and G on the validation set. In contrast to A and T which seem to be impacted by off-by-one errors. This is seen more clearly when examining the classifications for the lengths only (see Appendix G). For example, in row 7 of Figure G.2 it is seen that the proportions of classifications for 7 are: 30% for 6, 31% for 7 (actual class), and 14% for 8.

One peculiarity is the accuracies for lengths 7–10 in Appendix G on the validation set. Length 10 has an accuracy of 44%, while 7, 8, and 9 have accuracies of 31%, 26%, and 20%, respectively. This seems odd as there are more samples for lengths 7–9 than for length 10. It may be due to length 10 being a boundary class, i.e. the last element in output logits, and thus due to there being no length 11, off-by-one errors are not seen for length 10 as with the other lengths.

4.4.6 Summary

The purpose of this experiment was to create a homopolymer length classifier. Multiple learning tasks were tried to reach this end: length classification using cross-entropy, and length regression using mean squared error.

Cross-entropy produces similar results compared to (1) mean squared error and (2) multi-loss that combines mean squared error and cross-entropy. We therefore opted to train a model with more samples and over more epochs using cross-entropy.

From hyperparameter tuning it was found that a large kernel size contributed the most to a good accuracy. Channel size and number of residual blocks contributed less. Padding the homopolymer region also did not yield improvements. In fact, it was found that having a large context around the homopolymer yields worse results.

For training a model over more epochs, kernel size was set to 97, blocks was set to 2, and channel size was set to 16. For padding, the homopolymer was padded from both sides with 10 signals. After training for 160 epochs, this model achieved an average length accuracy of 44.1% on the validation set. In addition, the confusion matrix (Appendix G) shows that the model is mostly plagued by off-by-one errors.

4.5 Evaluation

Having trained U-Seg to distinguish homopolymer regions from non-homopolymer regions in a read, and a model for classifying the length of these homopolymer regions, we now combine the two into our proposed pipeline, and evaluate the system on nanopore reads.

Using our proposed pipeline (described in Chapter 4) was problematic due to misalignments (described in Section 5.1.4) in the predicted sequences from the U-Seg model and Bonito. So we decided to simplify the pipeline to circumvent the misalignemnt problems. An overview of the pipeline is shown in Figure 4.10.

The signals from a read is fed to both the Bonito model and U-Seg and homopolymer regions found in both models are fed the HP length model. Signals from Bonito are passed to the HP length model, if Bonito predicts a homopolymer, or if an event is longer than 35 signals. Additionally, homopolymer regions found by the U-Seg model are fed to the HP length model. For non-homopolymeric regions we use the predictions produced by Bonito.

This evaluation will show if our method can refine homopolymer lengths in reads that have been basecalled by Bonito. The reads are compared in the following way. First, a

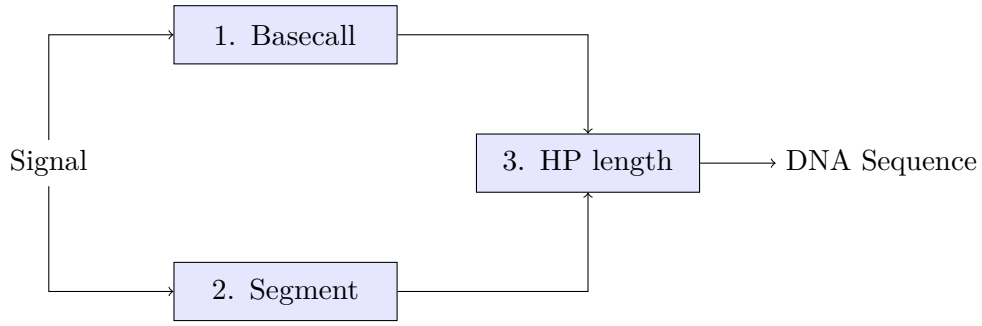


Figure 4.10: Simplified pipeline for basecalling HP signals.

baseline for finding homopolymers is established with Bonito. Second, a refinement of the homopolymers found by Bonito is performed with the length classifier by examining the signal events from Bonito. Third, a refinement of homopolymers are performed with U-Seg and the length classifier, and then inserted into the basecalls from Bonito.

Running the system on 2,443 reads (selected for their abundance of homopolymers) yields the results in Table 4.16. Slight improvements are seen for lengths 5–9 with Bonito + U-Seg + Length, whereas length 10 has gained a 6% improvement over both Bonito and Bonito + Length.

Method	5	6	7	8	9	10
Bonito	**0.81**	0.71	0.56	0.46	0.33	0.24
Bonito + Bonito HPs + Length	0.8	0.71	0.56	0.44	0.34	0.24
Bonito + U-Seg HPs + Length	**0.81**	0.71	**0.57**	**0.47**	**0.35**	**0.3**

Table 4.16: Accuracy for HP lengths 5–10.

We used the alignment tool minimap2 [23] to quantify how well the reads compared to the references. This provides counts of correct bases, mismatches, insertions, and deletions, akin to Levenshtein distance.

Examining the correct bases, mismatches, insertions, and deletions, it is seen that Bonito + Length decreases all these quantities significantly compared to Bonito (Table 4.17). This decrease in basecall errors, is also noted by a slight increase in mean identity-rate and a decrease in mean error-rate (Table 4.18). These were calculated using the counts from minimap2 in the following way:

$$identity\_rate = 1 - \frac{del\_count + mis\_count}{ref\_length} \tag{4.5}$$

$$error\_rate = \frac{del\_count + mis\_count + ins\_count}{ref\_length} \tag{4.6}$$

In contrast, Bonito + U-Seg + Length increases significantly for insertions while the other quantities are mostly unchanged compared to Bonito. This is seen by the 2% increase for the error-rate in Table 4.18. In our study Bonito achieved an identity-rate of 0.81, however, this falls short of the 97% accuracy advertised by ONT [27]. Though it should be noted that the accuracies are from different genomes, so this comparison may be moot.

Method	Correct	Mis	In	Del
Bonito	15906373	1746745	803747	1920198
Bonito + Bonito HPs + Length	**12674284**	**1360751**	**779317**	**1253259**
Bonito + U-Seg HPs + Length	15930803	1773618	1157982	1868895

Table 4.17: Error types: Correct, mismatches, insertions, and deletions.

Method	Identity-rate	Error-rate
Bonito	0.81	0.23
Bonito + Bonito HPs + Length	**0.83**	**0.22**
Bonito + U-Seg HPs + Length	0.81	0.25

Table 4.18: Identity-rate and error-rate.

In summary, a small improvement over Bonito is seen with our homopolymer correction models. Length 10 has the largest increase of 6% in accuracy by using Bonito + U-Seg + Length. For identity-rate and error-rate, only Bonito + Length improves with 2% and −1%, respectively, over Bonito.

Chapter 5

Closure

5.1 Discussion

Our system has now been completed and evaluated. During our experimentation and the development of the pipeline we encountered many difficulties with homopolymer predictions. Some of these difficulties have already been highlighted in the development sections, and we will discuss these further in the following. For convenience the topics of discussion will be covered individually in the subsequent sections.

5.1.1 Overlapping Homopolymer-signal Lengths

In Section 2.2 we found that there are significant overlaps between the corresponding signal lengths for homopolymers. This was true both for Tombo and Taiyaki as seen in Appendix A.3. The overlaps were discouraging, as it would evidently make it difficult to reliably infer homopolymer lengths directly from signal-lengths. Despite this significant overlap, our HP length classifier had an average accuracy ∼45% across lengths 1–10, and our pipeline approach achieved a 2% improvement compared to Bonito.

5.1.2 Improvement Validates Approach

Our pipeline approach managed to achieve an identity-rate of 0.83 compare to Bonito's 0.81. An improvement of 2% is modest, but new basecallers also typically only outperforms the previous by a few percent, therefore our efforts have been worthwhile. It also validates that our approach works, whereby a focused effort into homopolymer basecalling provided beneficial results. Though none of the individual components of our pipeline are outstanding, the pipeline as a whole works commendably. Because our approach make the basecalls more accurate, we consider the added complexity worthwhile. Future efforts into maximizing the performance of the individual components may very well yield yet greater results.

5.1.3 Autoencoder

The autoencoder was, like the U-Seg model, also meant to segment HP and non-HP segments in the signal. However, due to difficulties with the approach we never got to focus on homopolymeric regions, and the model was only compared to Tombo events for individual

bases. Our idea to check for homopolymeric regions was to check the number of signals between adjacent breakpoints, as the homopolymer region should not contain breakpoints to divide the homopolymer into singular bases. Alternatively, a model might have been trained to achieve the same effect of classifying homopolymeric regions.

Furthermore, the autoencoder was only evaluated on metrics of our own design, to get an understanding of how to choose thresholds for breakpoint detection. The method is therefore not directly comparable to the U-Seg model or other approaches. Preferably the approach should be evaluated using false positive/negative rates. These might be calculated using a distance-threshold s.t. if a breakpoint is within n signals of an event, it is a true positive. A ROC curve could possibly be created by varying the size of this distance-threshold.

5.1.4 Misalignment

A problem we encountered when combining the output of the segmentation model (described in Section 4.1) and the Bonito model, is a misalignment in the predictions. The segmentation model predicts a region of the signal as homopolymer whereas Bonito predicts the same region at a slight offset.

An example of this problem is shown in Figure 5.1 where the blue squiggle is the normalized data, the dotted line is the homopolymer probabilities from the segmentation model, and the colored horizontal line at the top is the predictions from Bonito. The yellow

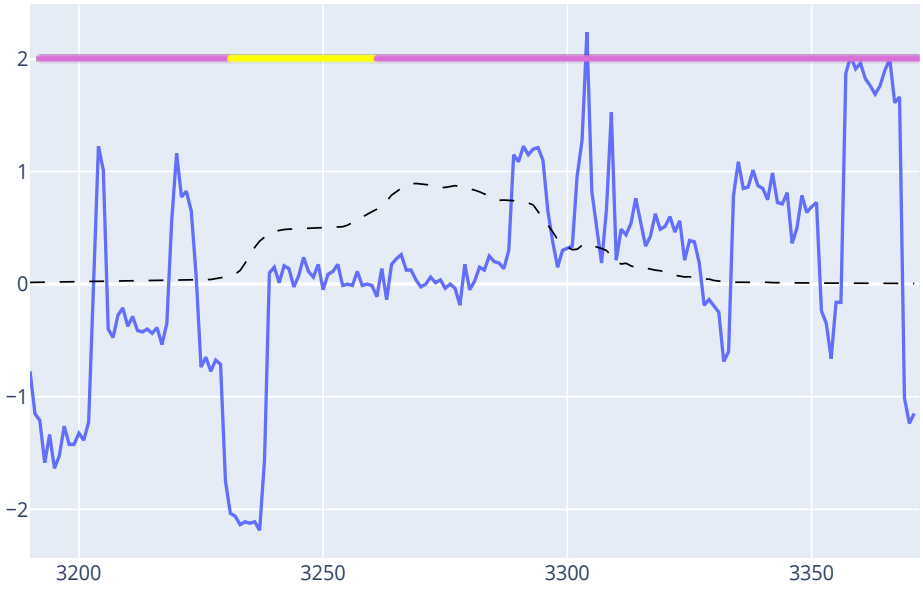


Figure 5.1: A figure containing the normalized data (blue squiggle), the homopolymer probabilities found by the U-Seg model (dotted black line) and the events found by Bonito(horizontal lines in the top)

horizontal line from the Bonito predictions is a 5T homopolymer. Bonito segments this homopolymer to occupy the signals from 3,230 to 3,261, occupying a total of 31 signals. The U-Seg model correctly identifies the beginning of the homopolymer region. When comparing the ending of the region, the probability starts decreasing 20 signals after the homopolymer regions stops by Bonito. In addition, the probability reaches 0% after a total of 80 signals after the Bonito region.

It is unclear whether the segment of the 5T found by Bonito was too short, or if there exists an another homopolymer region after the 5T. This misalignment between Bonito and U-Seg also complicates picking the correct base for a homopolymer that is reclassified by the length model. For example, if U-Seg produces a segment that spans over two or more Bonito segments, it is difficult determine which base should be inserted into the base sequence.

A similar alignment issue exists between Tombo events and Bonito events as well. For example, Figure 5.2 shows an 8T found by Bonito offset at 27 signals from the 7T found by Tombo.

Figure 5.2: A figure containing the normalized data (blue squiggle), the events found by Bonito(horizontal dotted lines), and Tombo events (horizontal solid lines)

5.2 Future Work

The following will cover some thoughts we have about future endeavors that may be worthwhile.

5.2.1 Training a Common Decoder

One way to alleviate the problem of misalignment between U-Seg and Bonito is to train a decoder that learns the correct alignment between the two models. For example, the outputs

of the Bonito and U-Seg can be fed to a common decoder module that merges the output of both models and produces a corrected output for homopolymer-length insertion.

5.2.2 Translocation Speed

Tombo events were used in Section 2.4 to analyse the number of signals per event to investigate translocation speed. To properly determine the amount of signals for a base, Tombo needs to accurately find event boundaries. However, as seen in Section 2.4 the mode signals per event is 3, which is the minimum number of signals per base in Tombo. Therefore, it may be the case that the signals per base fluctuates substantially due to Tombo setting events to have a length 3, which would make the alignment with the reference easier. Therefore, to properly analyse the translocation speed, Tombo would need to produce more accurate event boundaries.

Also, we only considered the lengths Tombo events individually and in close proximity. If one insteads considers global patterns, e.g. DNA self-entanglement occurs more frequently depending on patterns in the DNA sequence. The DNA bases that were entangled, however, would first be sequenced after their slowing effect would be observed, i.e. they would appear chronologically later than the bases that were slowed. A comprehensive investigation of translocation speed may be interesting, but could easily be an entire project in itself.

5.2.3 Altering Physical Homopolymer Segments

In this project we have had repeated difficulties with homopolymers signals. Though some homopolymer signals are identifiable from a longer-than-average idle period, a significant overlap in signal-lengths of homopolymers occur, as seen in Appendix A.3. This discrepancy has not been satisfactorily relieved by our approach, nor anyone else's. It is difficult to imagine that the problem can reliably be solved with software approaches, and it might therefore be necessary for alternative sequencing tools or even to alter the physical DNA.

The R9.4.3 nanopore by ONT has a context of $\sim$5 bases that influence the signal, and it reliably predicts homopolymers up to this length. The new R10.3 nanopore from ONT has increased the context to about $\sim$9 [32], and is also reliable up to this length [20]. However, the size of the context cannot be increased indefinitely, and even so, increasing the context size of sequencer only slightly solves the problem, as homopolymers beyond the length of the context are still lost.

Alternatively, it is possible to manipulate the DNA molecule to make homopolymers more pronounced. Albertsen Lab [2] is investigating just such a technique, whereby the bases of a homopolymer are modified at each nth base through an enzymatic process, where n is less than the context of the sequencer. For instance, the HP sequence `ATTTTTTTTTTTTC` could be modified into a sequence `ATTTmTTTmTTTmTTTC` with three modified bases mT and subsequently sequenced by a R9.4.3 nanopore. Doing such means that the consecutive 5-mers of the homopolymer are distinct, and a reliable basecalling can be performed. This also only partially solves the problem, as the technique is supposedly limited in the number of modifications it can perform. And, a technique that did not have to alter the physical DNA would be preferred, as when the DNA is physically altered, some of its characteristics are irrevocably removed, and some downstream analyses cannot be performed on this sample.

5.3 Conclusion

To address our problem statement, which asked *"how can accurate inference of homopolymers be achieved from nanopore sequences?"*, we developed a pipeline that facilitates the basecaller Bonito to improve homopolymer predictions.

Chapter 4 explored the overall solution pipeline containing three overall steps: 1) segment the signal into homopolymer and non-homopolymer regions, 2) basecall the segments using an already existing basecaller, and predict the lengths of homopolymer regions found by the segmentation model, and 3) combine the results to construct the final DNA sequence.

For the segmentation part of the pipeline we tried two different methods: 1) an autoencoder approach of finding breakpoints in the latent space representation. This approach did not perform satisfactorily due to it being difficult to fine-tune the hyperparameters for reliable breakpoint detection. 2) a U-Seg model based on U-Net, with the task of classifying each element in the sequence to either be part of a homopolymer or non-homopolymer region. The U-Seg showed promising results in both testing and training with non-HP segmentation accuracy of ∼97% and homopolymer segmentation accuracy of ∼50%. But the evaluation of the model showed underwhelming result with non-HP accuracy of ∼80% and homopolymer accuracy of ∼19%.

For the length prediction of homopolymers we used the widely known ResNet architecture with the task of classifying signal segments into 10 different classes (lengths 1-10). This achieved promising results with an average validation accuracy of ∼44% across the 10 lengths.

The pipeline showed modest improvements in the classification of homopolymer, with an improved identity-rate of 2% compared with Bonito. An improvement of 2% is not remarkable, still, we showed a correlation between the signals and homopolymers that can be exploited with positive results.

Bibliography

[1] B. Alberts et al. *Molecular Biology of the Cell*. Garland Science, 2008. ISBN: 9780815341116. URL: https://books.google.dk/books?id=iepqmRfP3ZoC.

[2] *Albertsen Lab – Applied bioinformatics*. 2020. URL: https://albertsenlab.org/.

[3] V. Badrinarayanan, A. Kendall and R. Cipolla. 'SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.12 (2017), pp. 2481–2495.

[4] Léon Bottou. 'Large-Scale Machine Learning with Stochastic Gradient Descent'. In: *Proceedings of COMPSTAT'2010*. Ed. by Yves Lechevallier and Gilbert Saporta. Heidelberg: Physica-Verlag HD, 2010, pp. 177–186. ISBN: 978-3-7908-2604-3.

[5] Daniel Branton et al. 'The potential and challenges of nanopore sequencing'. In: *Nature Biotechnology* 26.10 (Oct. 2008), pp. 1146–1153. ISSN: 1546-1696. DOI: 10.1038/nbt.1495. URL: https://doi.org/10.1038/nbt.1495.

[6] Clive G. Brown. *Clive G Brown: Nanopore Community Meeting 2018 talk*. 29th Nov. 2019. URL: https://nanoporetech.com/about-us/news/clive-g-brown-nanopore-community-meeting-2018-talk.

[7] Z. Cheng et al. 'Deep Convolutional AutoEncoder-based Lossy Image Compression'. In: *2018 Picture Coding Symposium (PCS)*. 2018, pp. 253–257.

[8] *Dam and Dcm Methylases of E. coli | NEB*. URL: https://international.neb.com/tools-and-resources/usage-guidelines/dam-and-dcm-methylases-of-e-coli (visited on 15/01/2020).

[9] John Duchi, Elad Hazan and Yoram Singer. 'Adaptive Subgradient Methods for Online Learning and Stochastic Optimization'. In: *J. Mach. Learn. Res.* 12.null (July 2011), pp. 2121–2159. ISSN: 1532-4435.

[10] Xavier Glorot, Antoine Bordes and Yoshua Bengio. 'Deep Sparse Rectifier Neural Networks'. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Geoffrey Gordon, David Dunson and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Nov. 2011, pp. 315–323. URL: http://proceedings.mlr.press/v15/glorot11a.html.

[11] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[12] Richard H. R. Hahnloser et al. 'Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit'. In: *Nature* 405.6789 (2000), pp. 947–951. ISSN: 1476-4687. DOI: 10.1038/35016072. URL: https://doi.org/10.1038/35016072.

[13] Nima Hatami, Yann Gavet and Johan Debayle. 'Classification of time-series images using deep convolutional neural networks'. In: *Tenth International Conference on Machine Vision (ICMV 2017)*. Ed. by Antanas Verikas et al. Vol. 10696. International Society for Optics and Photonics. SPIE, 2018, pp. 242–249. DOI: 10.1117/12.2309486. URL: https://doi.org/10.1117/12.2309486.

[14] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. eprint: arXiv:1512.03385.

[15] National Human Genome Research Institute. *ACGT*. URL: https://www.genome.gov/genetics-glossary/acgt.

[16] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. URL: https://arxiv.org/pdf/1502.03167.pdf.

[17] Fabian Isensee et al. 'Automatic Cardiac Disease Assessment on cine-MRI via Time-Series Segmentation and Domain Specific Features'. In: *Statistical Atlases and Computational Models of the Heart. ACDC and MMWHS Challenges*. Ed. by Mihaela Pop et al. Cham: Springer International Publishing, 2018, pp. 120–129. ISBN: 978-3-319-75541-0.

[18] Bingyi Kang et al. *Decoupling Representation and Classifier for Long-Tailed Recognition*. 2019. arXiv: 1910.09217 [cs.CV].

[19] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[20] Rasmus H. Kirkegaard. *We aR(10.)3 pretty close now!!!* Feb. 2020. URL: https://albertsenlab.org/we-ar10-3-pretty-close-now/.

[21] Colin Lea et al. 'Temporal Convolutional Networks: A Unified Approach to Action Segmentation'. In: *Computer Vision – ECCV 2016 Workshops*. Ed. by Gang Hua and Hervé Jégou. Cham: Springer International Publishing, 2016, pp. 47–54. ISBN: 978-3-319-49409-8.

[22] Wei-Han Lee et al. 'Time Series Segmentation through Automatic Feature Learning'. In: (Jan. 2018).

[23] Heng Li. 'Minimap2: Pairwise alignment for nucleotide sequences'. In: *Bioinformatics* (2018). ISSN: 14602059. DOI: 10.1093/bioinformatics/bty191. arXiv: 1708.01492.

[24] LomanLab. *Home | Loman Lab Mock Community Experiments*. URL: https://lomanlab.github.io/mockcommunity/.

[25] Jonathan Long, Evan Shelhamer and Trevor Darrell. *Fully Convolutional Networks for Semantic Segmentation*. 2014. eprint: arXiv:1411.4038.

[26] Dmytro Mishkin, Nikolay Sergievskiy and Jiri Matas. 'Systematic evaluation of convolution neural network advances on the Imagenet'. In: *Computer Vision and Image Understanding* (2017). ISSN: 1077-3142. DOI: https://doi.org/10.1016/j.cviu.2017.05.007. URL: http://www.sciencedirect.com/science/article/pii/S1077314217300814.

[27] *New research algorithms yield accuracy gains for nanopore sequencing*. Mar. 2020. URL: https://nanoporetech.com/about-us/news/new-research-algorithms-yield-accuracy-gains-nanopore-sequencing.

[28] Chigozie Nwankpa et al. 'Activation Functions: Comparison of trends in Practice and Research for Deep Learning'. In: *CoRR* abs/1811.03378 (2018). arXiv: 1811.03378. URL: http://arxiv.org/abs/1811.03378.

[29] Jakob Stig Olesen, Nicklas Kirkegaard Breum and Martin Nyholm Jørgensen. *Groundwork for Developing a Basecaller*. https://projekter.aau.dk/projekter/da/studentthesis/groundwork-for-developing-a-basecaller(57a8bb11-bb4d-4432-aa42-21e1c0f00384).html. 2020.

[30] William Press. *Numerical recipes in C : the art of scientific computing*. Cambridge Cambridgeshire New York: Cambridge University Press, 1992. ISBN: 0-521-43108-5.

[31] PyTorch. *Cross Entropy Loss*. URL: https://pytorch.org/docs/stable/nn.html#crossentropyloss.

[32] *R10.3: the newest nanopore for high accuracy nanopore sequencing – now available in store*. Jan. 2020. URL: https://nanoporetech.com/about-us/news/r103-newest-nanopore-high-accuracy-nanopore-sequencing-now-available-store.

[33] Franka J. Rang, Wigard P. Kloosterman and Jeroen de Ridder. 'From squiggle to basepair: computational approaches for improving nanopore sequencing read accuracy'. In: *Genome Biology* 19.1 (2018), p. 90. ISSN: 1474-760X. DOI: 10.1186/s13059-018-1462-9. URL: https://doi.org/10.1186/s13059-018-1462-9.

[34] Olaf Ronneberger, Philipp Fischer and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. eprint: arXiv:1505.04597.

[35] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. arXiv: 1609.04747 [cs.LG].

[36] Oxford Nanopore Technologies. *Guppy v3.6.0 Release*. URL: https://community.nanoporetech.com/posts/guppy-v3-6-0-release.

[37] Oxford Nanopore Technologies. *Tombo*. 2019. URL: https://github.com/nanoporetech/tombo.

[38] Oxford Nanopore Technologies. *Training models for basecalling Oxford Nanopore reads*. https://github.com/nanoporetech/taiyaki. 2020.

[39] Marijke M.A. Thijssen. 'Calling homopolymers in nanopore sequencing data'. In: (2019). URL: https://edepot.wur.nl/472283.

[40] John M. Urban et al. 'Sequencing ultra-long DNA molecules with the Oxford Nanopore MinION'. In: 2015.

[41] Yueqing Wang et al. 'An efficient and effective convolutional auto-encoder extreme learning machine network for 3d feature learning'. In: *Neurocomputing* 174 (2016), pp. 988–998. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2015.10.035. URL: http://www.sciencedirect.com/science/article/pii/S0925231215014940.

[42] Ryan R. Wick, Louise M. Judd and Kathryn E. Holt. 'Performance of neural network basecalling tools for Oxford Nanopore sequencing'. In: *Genome Biology* 20.1 (2019), p. 129. ISSN: 1474-760X. DOI: 10.1186/s13059-019-1727-y. URL: https://doi.org/10.1186/s13059-019-1727-y.

[43] Xiangyue Wu and Gary Brewer. 'The regulation of mRNA stability in mammalian cells: 2.0'. eng. In: *Gene* 500.1 (May 2012). S0378-1119(12)00319-8[PII], pp. 10–21. ISSN: 1879-0038. DOI: 10.1016/j.gene.2012.03.021. URL: https://doi.org/10.1016/j.gene.2012.03.021.

[44] Bing Xu et al. 'Empirical Evaluation of Rectified Activations in Convolutional Network'. In: *CoRR* abs/1505.00853 (2015). arXiv: 1505.00853. URL: http://arxiv.org/abs/1505.00853.

[45] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. 2012. arXiv: 1212.5701 [cs.LG].

[46] Yao-zhong Zhang et al. 'Nanopore base-calling from a perspective of instance segmentation'. In: *bioRxiv* (2019). DOI: 10.1101/694919. eprint: https://www.biorxiv.org/content/early/2019/07/08/694919.full.pdf. URL: https://www.biorxiv.org/content/early/2019/07/08/694919.

Appendices

Appendix A

Taiyaki and Tombo

A.1 Homopolymer Counts

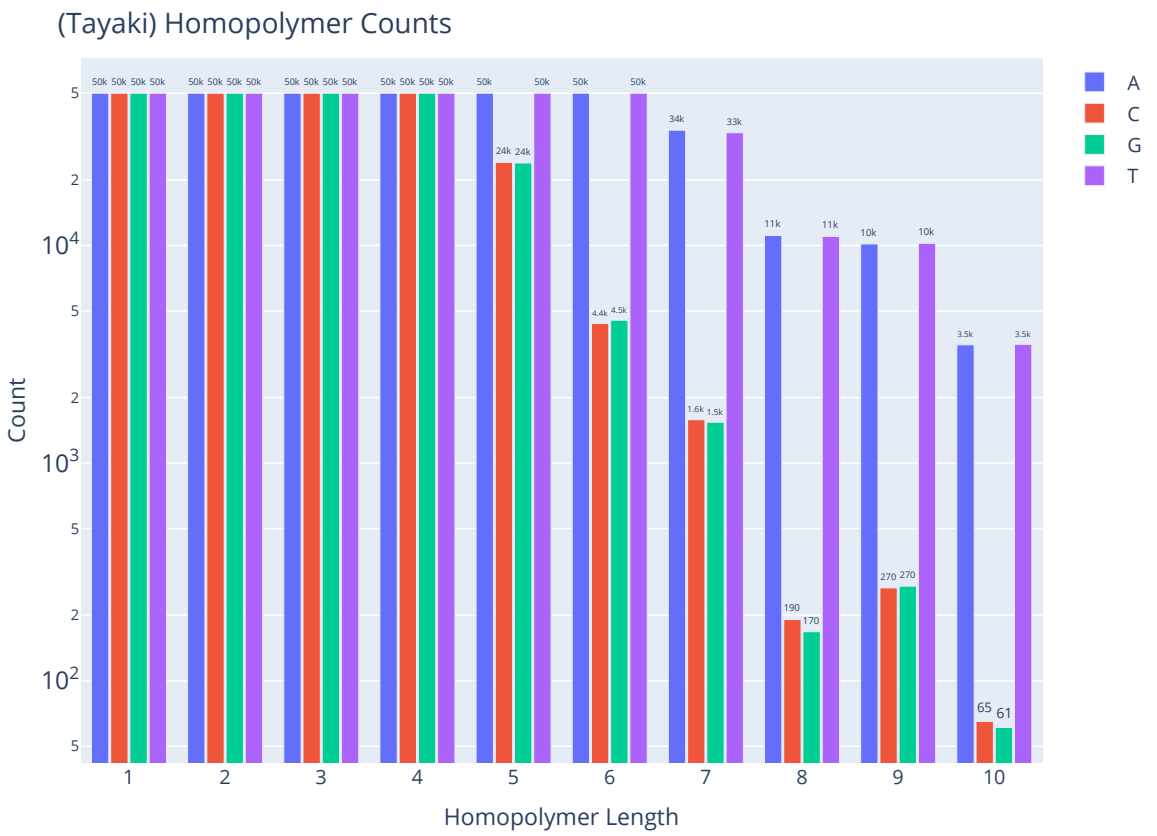


Figure A.1: Homopolymer count from Taiyaki.

[H]

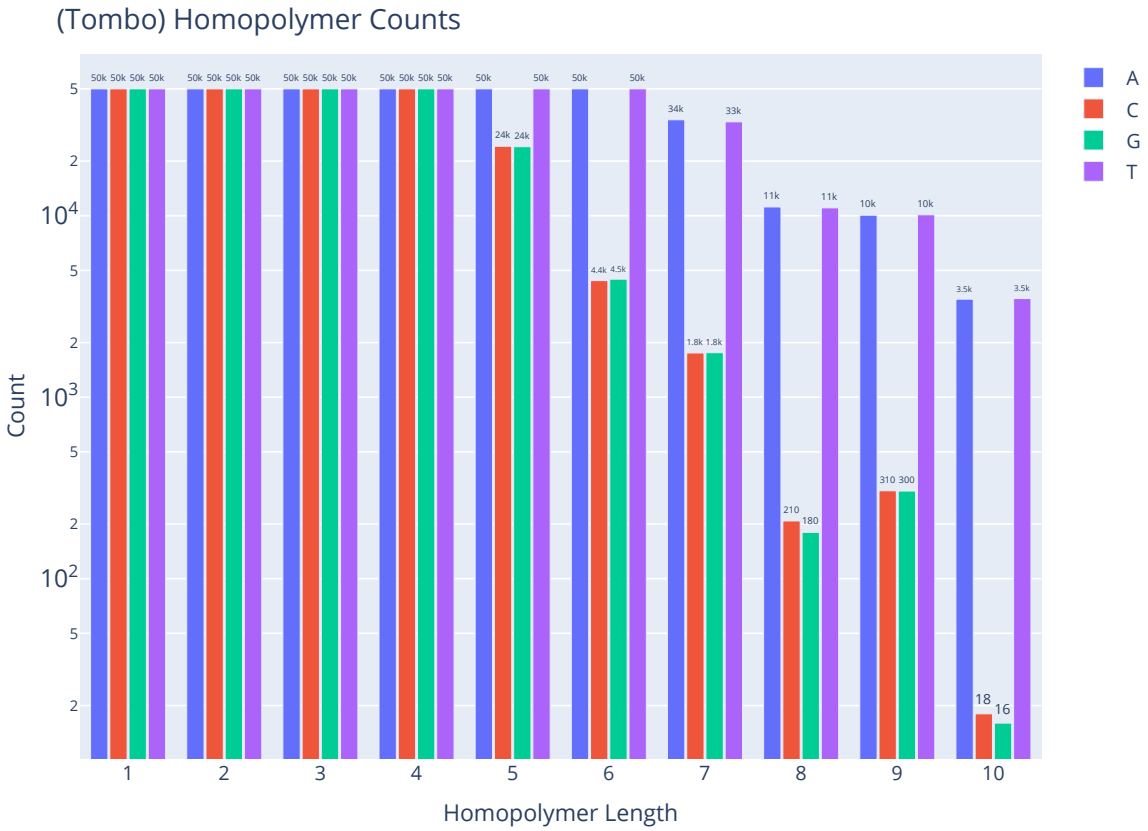


Figure A.2: Homopolymer count from Tombo.

A.2 Homopolymer Signals

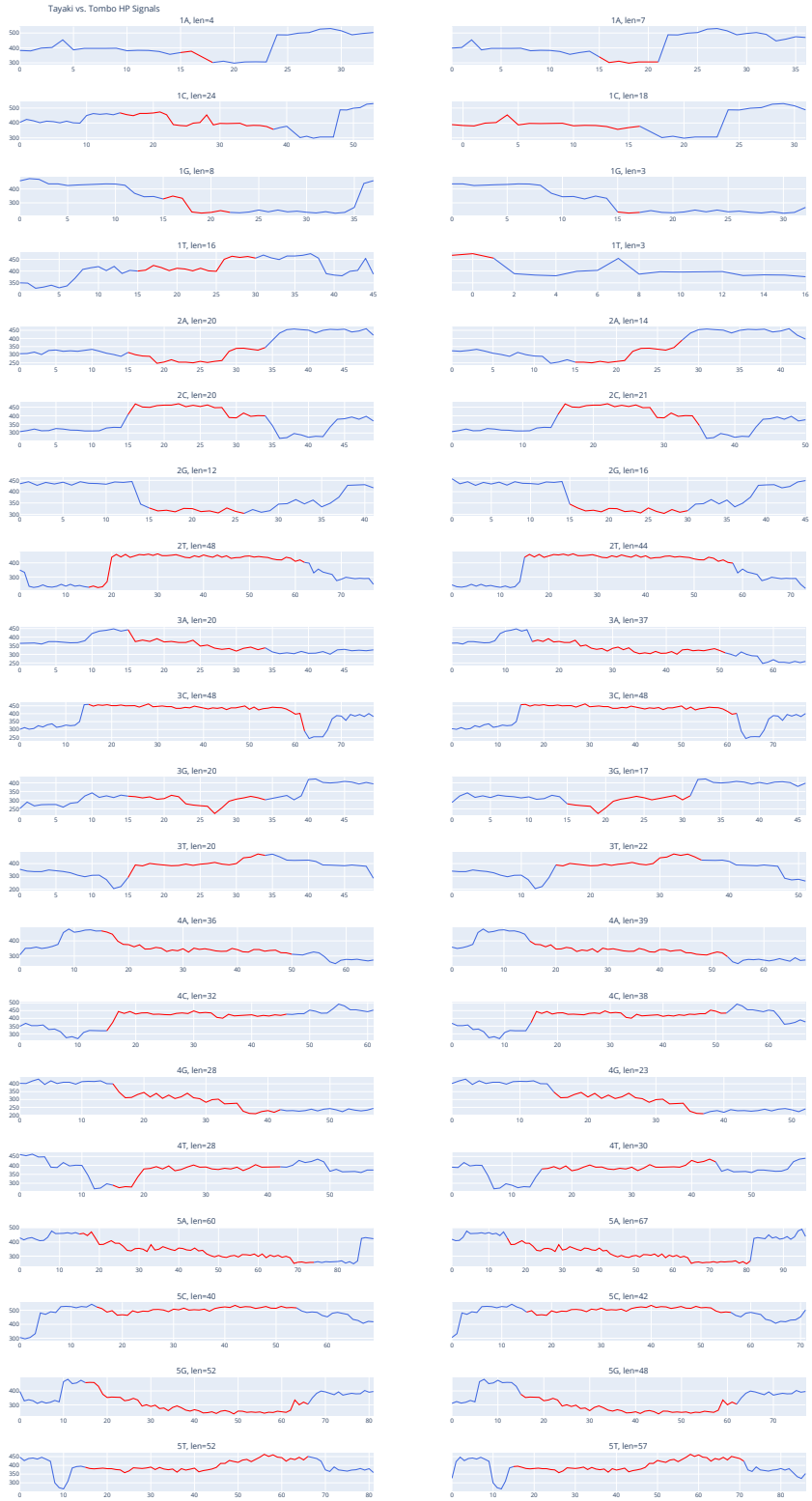


Figure A.3: Homopolymer signals (in red) for Taiyaki (left) and Tombo (right). X-axis shows relative signal position and y-axis is the data acquisition (DAC) values.

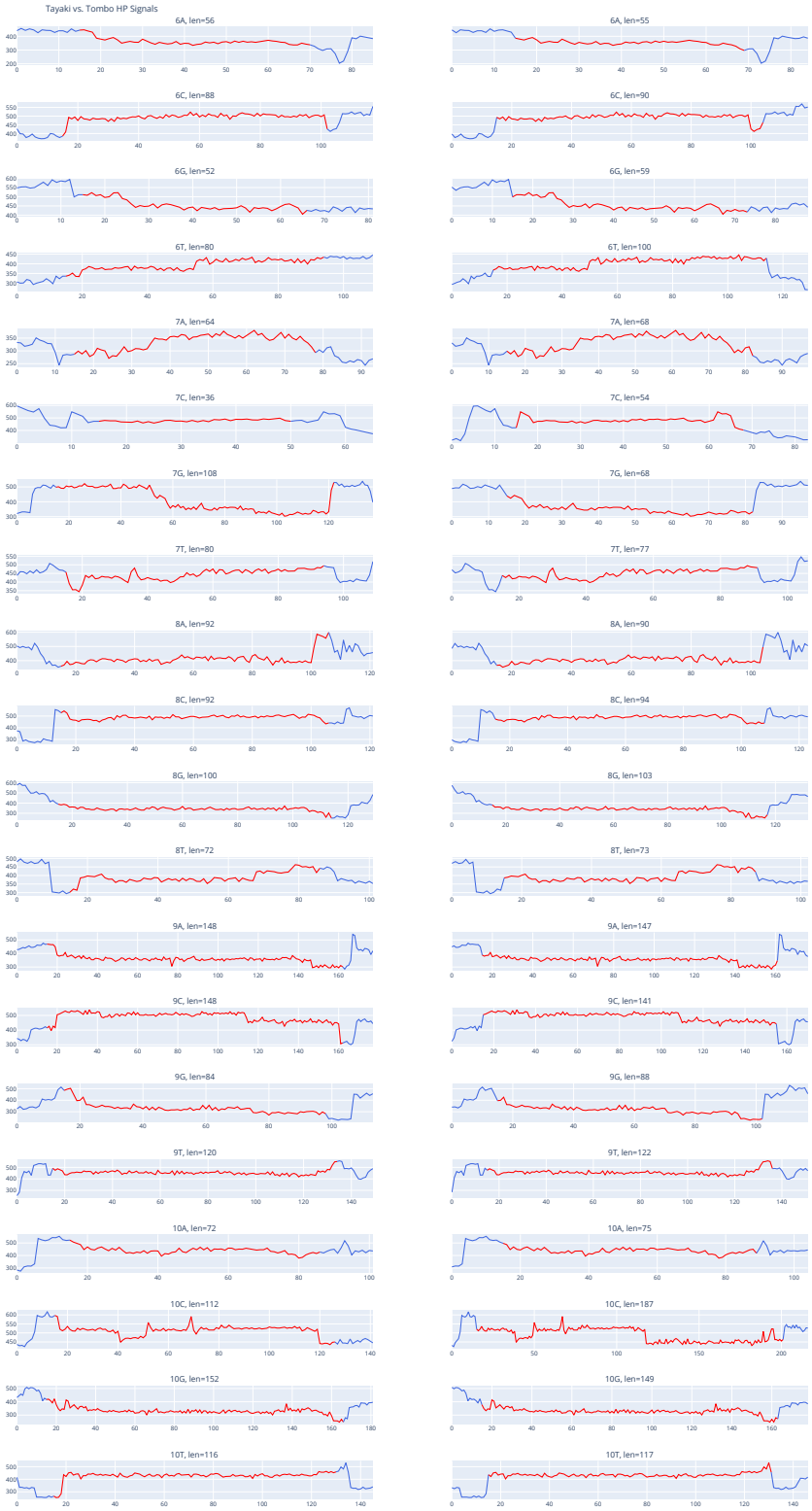


Figure A.4: Homopolymer signals (in red) for Taiyaki (left) and Tombo (right). X-axis shows relative signal position and y-axis is the data acquisition (DAC) values.

A.3 Signal Length Distribution

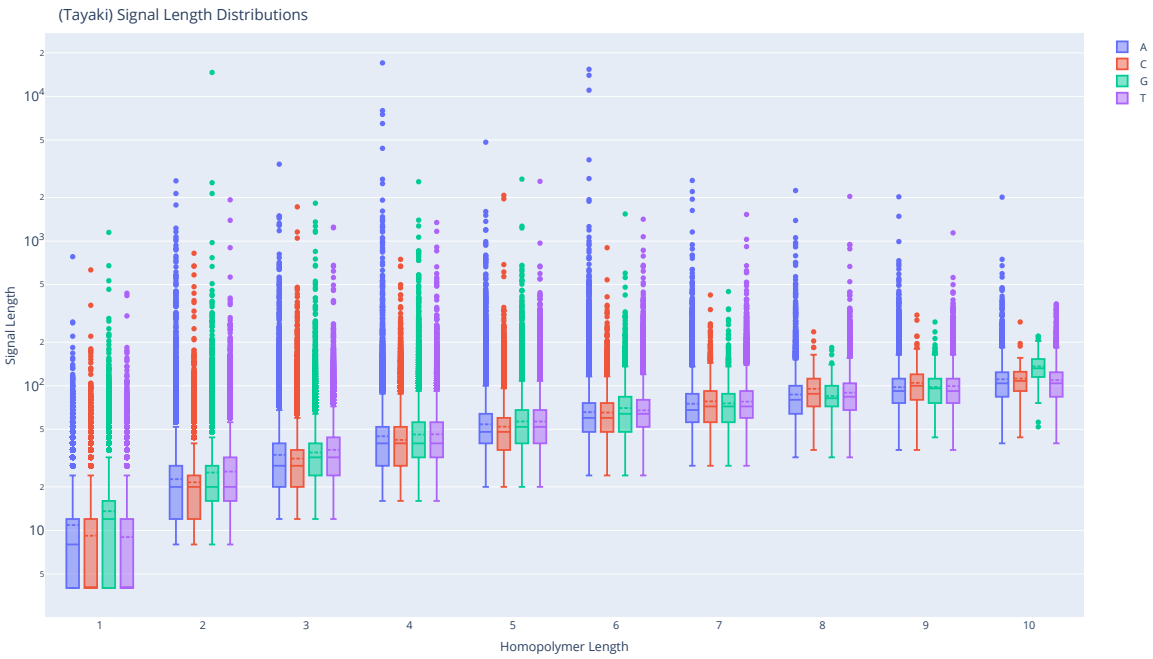


Figure A.5: Signal length distribution from Taiyaki.

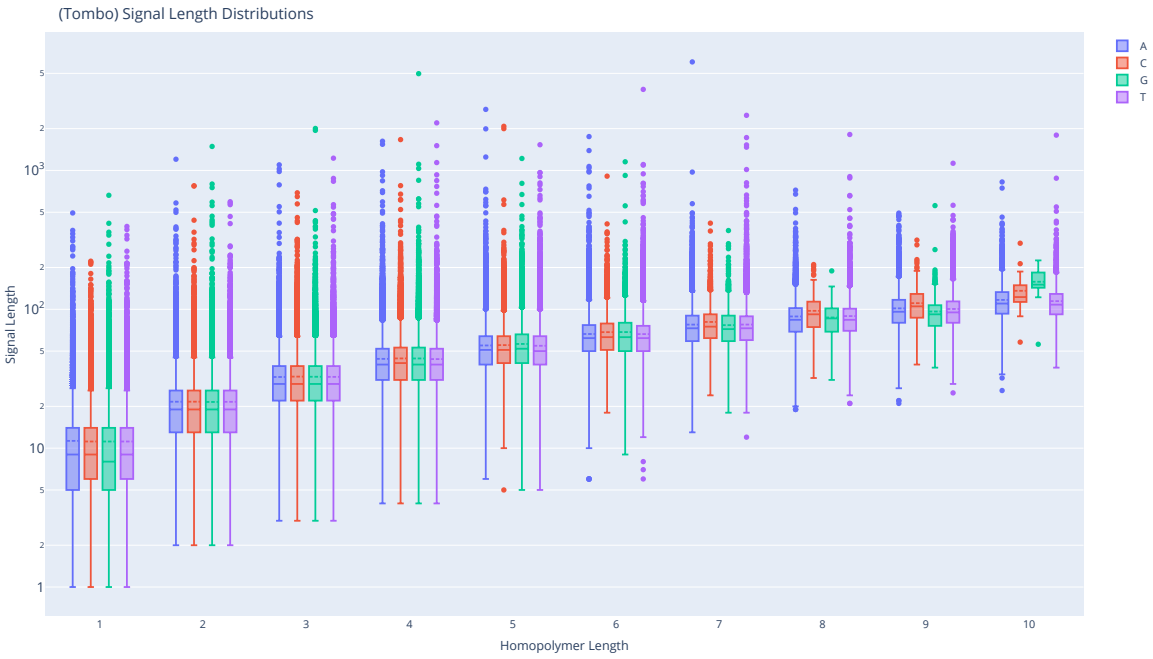


Figure A.6: Signal length distribution from Tombo.

A.4 All Homopolymers found by Tombo

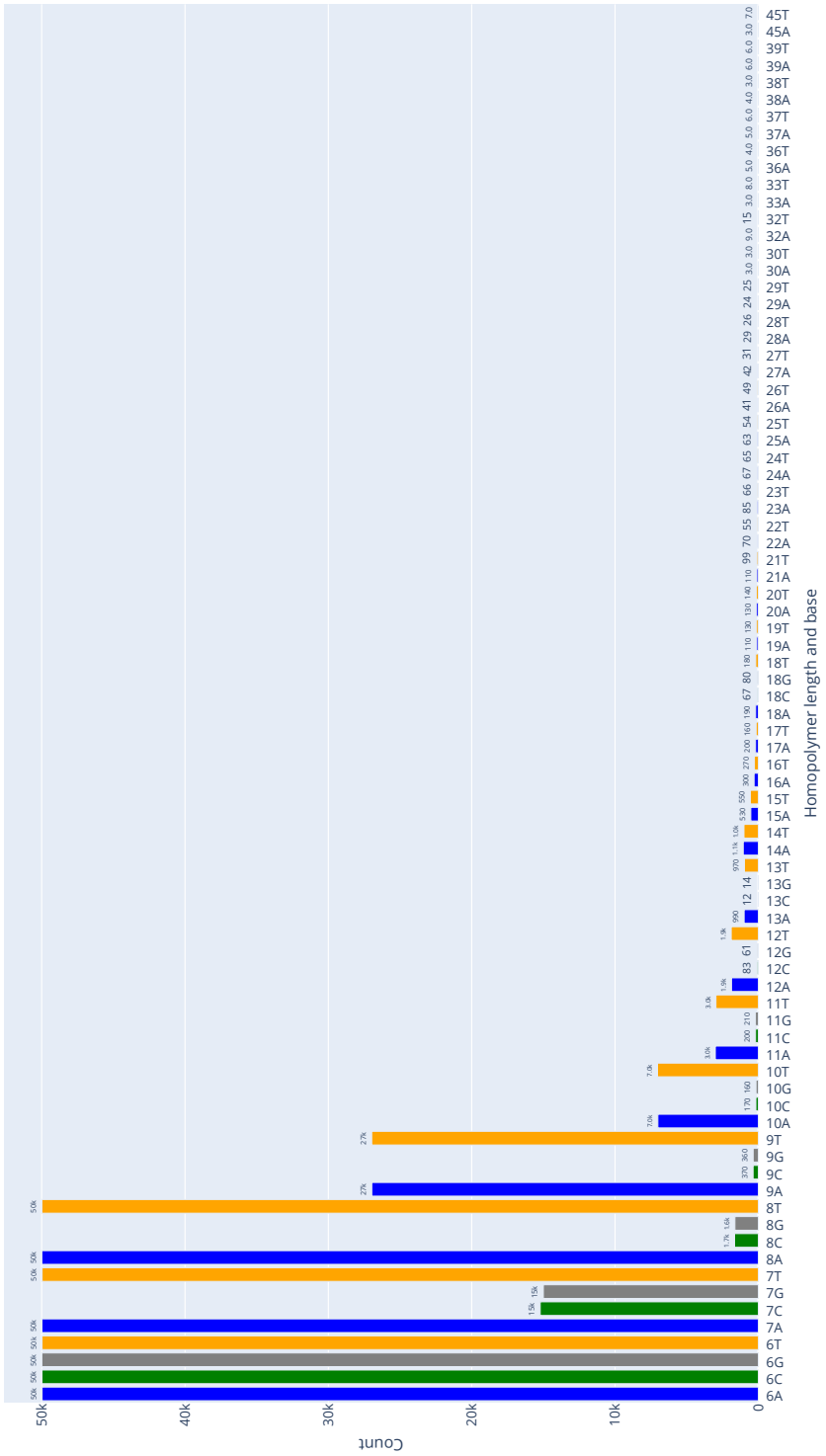


Figure A.7: Homopolymers from 6–45 in Loman Lab Mock Community[24] per base. As the length of the homopolymer grows its abundance steadily declines. nA and nT are overwhelmingly more abundant than nC and nG. In fact, the only nC and nG with $n > 13$ are $18C$ and $18G$ with counts of 67 and 80 respectively.

Appendix B

Translocation speed

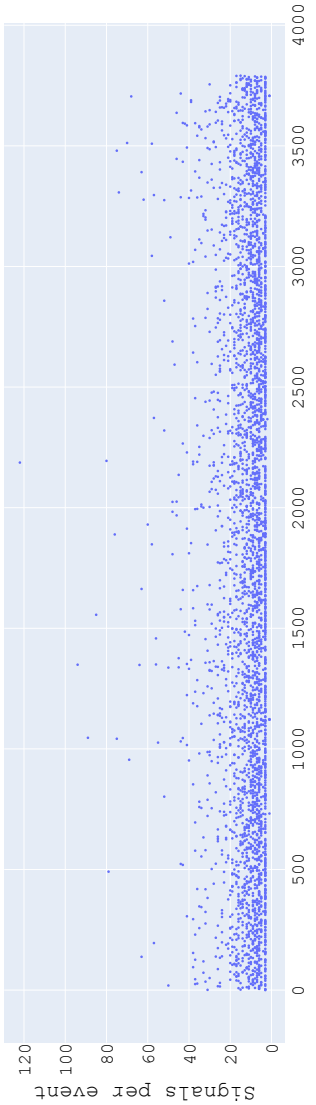


Figure B.1: Number of signals per event produced by Tombo for an entire read

Appendix C

Autoencoder

C.1 Autoencoder Experiment Results

C.1.1 Span of Distance Curve

As described in Algorithm 3, the distance curve is assembled s.t. a distance value spans its corresponding window ignoring overlaps, which is meant to correspond signal indices to distances. In a distance curve, a set of distance values will span $segment\_size - overlap\_size$ signals. This means that a higher overlap will provide a higher resolution of the distance curve. Figure C.1 shows a distance curve that spans 16 signals and Figure C.2 shows a distance curve that spans 8 signals. A higher resolution also provides more breakpoint candidates (the purple spikes in the figures).

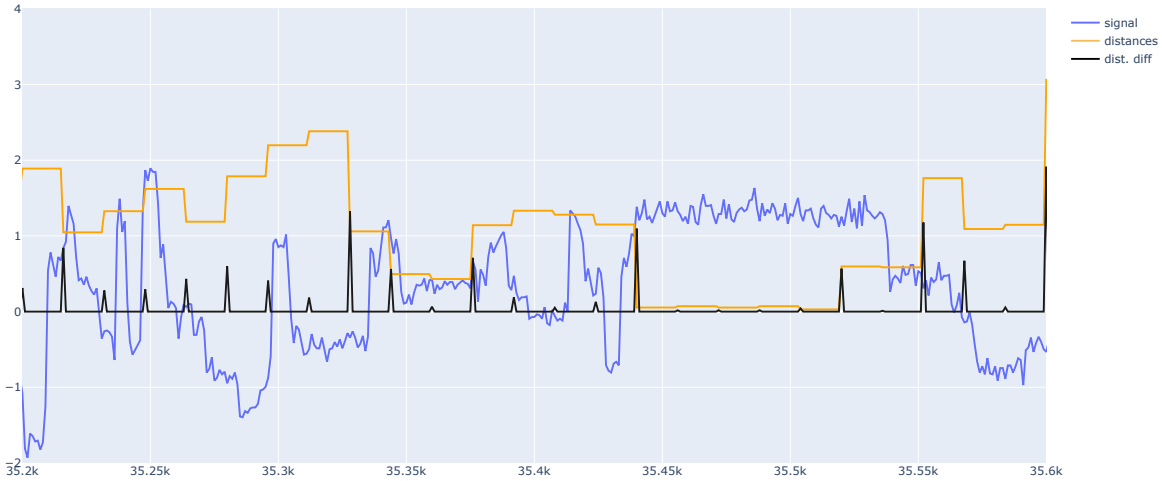


Figure C.1: Example of read window with segment size 16 and overlap of 0. The distance curve (gold) steps every 16th signal.

C.1.2 Read Tail Uncovered by Events

Some reads that have been processed by either Taiyaki or Tombo lack events for the end portion of the read. An example of this is shown in Figure C.3, where the events stop around signal 161,350, while the entire read spans about 178,400 signals. As said this is

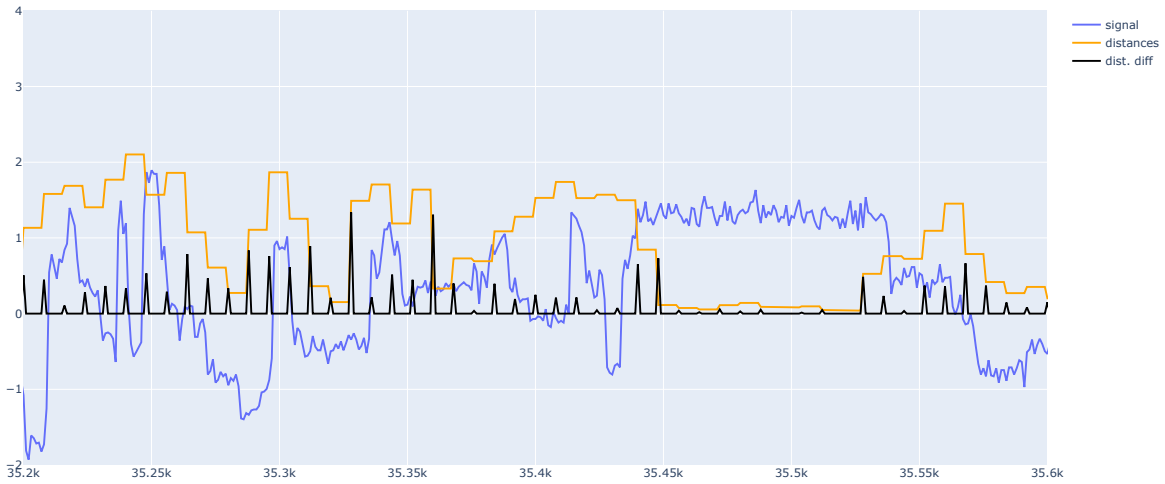


Figure C.2: Example of read window with segment size 16 and overlap of 8. The distance curve (gold) steps every every 8th signal.

evident from both Taiyaki and Tombo, indicating that the problem is inherent to the read. Excluding these reads from our dataset would be disingenuous and would be considered manipulating the dataset to portray our approach in a favorable light.

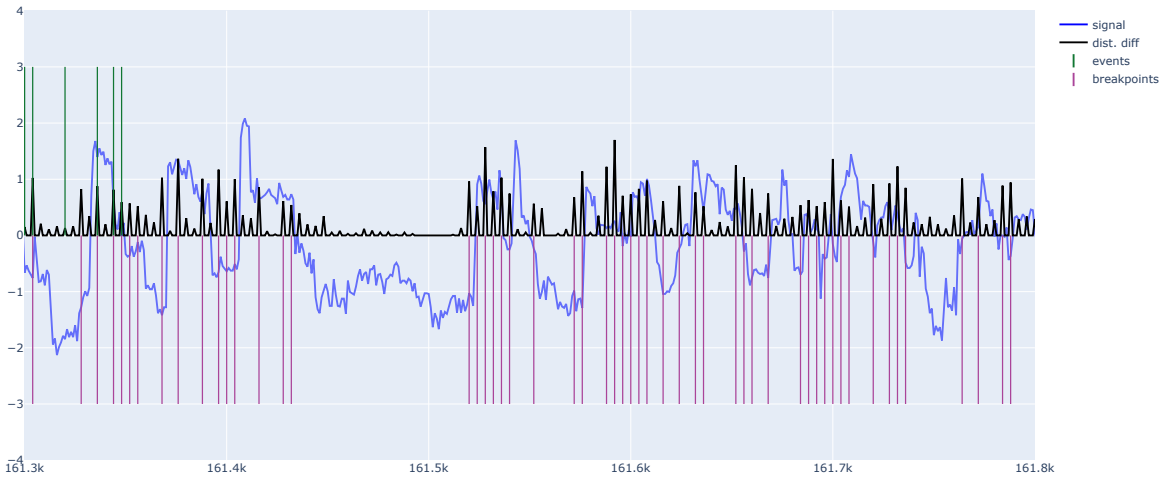


Figure C.3: Example of read lacking events towards the end. The read has a total length of 178,400 signals.

model		distances<500	distances≥500	bps_pr_event	nearest_bp_val
Segment: 4 Overlap: 2 Channels: 64	count	503453.00	2441.00	240878.00	345299.00
	mean	4.53	3686.82	2.10	0.48
	std	13.38	2164.84	3.67	0.57
	min	0.00	500.00	1.00	0.01
	25%	2.00	1632.00	1.00	0.10
	50%	2.00	3670.00	2.00	0.29
	75%	4.00	5612.00	3.00	0.66
	95%	12.00	7042.00	4.00	1.48
	max	498.00	7446.00	1193.00	26.38
Segment: 8 Overlap: 6 Channels: 64 32	count	622141.00	3290.00	268392.00	345299.00
	mean	4.73	3636.11	2.33	0.59
	std	15.41	2138.80	4.66	0.59
	min	0.00	500.00	1.00	0.01
	25%	2.00	1638.50	1.00	0.16
	50%	2.00	3551.00	2.00	0.40
	75%	4.00	5475.50	3.00	0.88
	95%	14.00	7058.00	5.00	1.67
	max	498.00	7448.00	1576.00	28.95
Segment: 16 Overlap: 14 Channels: 64 32 16	count	366897.00	2087.00	195669.00	345299.00
	mean	5.03	3558.68	1.89	0.35
	std	15.26	2142.63	3.46	0.45
	min	0.00	500.00	1.00	0.01
	25%	2.00	1576.00	1.00	0.08
	50%	2.00	3424.00	2.00	0.20
	75%	6.00	5356.00	2.00	0.45
	95%	12.00	7051.40	4.00	1.18
	max	498.00	7444.00	1025.00	35.96
Segment: 16 Overlap: 14 Channels: 64 32 16 8	count	83638.00	424.00	50705.00	345299.00
	mean	4.34	3772.69	1.66	0.15
	std	14.13	2117.39	1.61	0.24
	min	0.00	500.00	1.00	0.01
	25%	2.00	1815.50	1.00	0.04
	50%	2.00	3748.00	1.00	0.08
	75%	4.00	5333.00	2.00	0.17
	95%	12.00	7151.70	3.00	0.54
	max	482.00	7426.00	221.00	11.03

Table C.1: Metrics for varying segment size for breakpoint detection.

model		distances<500	distances≥500	bps_pr_event	nearest_bp_val
	count	113776.00	388.00	105063.00	345299.00
Segment:	mean	5.92	3574.87	1.09	0.57
16	std	13.27	2167.24	0.89	0.48
Overlap:	min	0.00	512.00	1.00	0.01
0	25%	0.00	1490.00	1.00	0.20
Channels:	50%	4.00	3544.00	1.00	0.44
64	75%	8.00	5490.00	1.00	0.82
32	95%	16.00	7097.20	2.00	1.48
	max	480.00	7444.00	201.00	10.91
	count	139513.00	457.00	122198.00	345299.00
Segment:	mean	5.81	3601.75	1.15	0.51
16	std	12.62	2135.19	0.98	0.42
Overlap:	min	0.00	504.00	1.00	0.01
4	25%	0.00	1580.00	1.00	0.19
Channels:	50%	4.00	3620.00	1.00	0.41
64	75%	8.00	5384.00	1.00	0.73
32	95%	16.00	7042.40	2.00	1.27
	max	496.00	7388.00	218.00	7.49
	count	164462.00	511.00	135474.00	345299.00
Segment:	mean	5.45	3590.15	1.22	0.41
16	std	12.43	2105.64	1.09	0.33
Overlap:	min	0.00	520.00	1.00	0.01
8	25%	0.00	1616.00	1.00	0.16
Channels:	50%	4.00	3676.00	1.00	0.35
64	75%	8.00	5352.00	1.00	0.58
32	95%	16.00	7088.00	2.00	0.99
	max	496.00	7428.00	253.00	7.54
	count	103189.00	546.00	67080.00	345299.00
Segment:	mean	5.26	3460.32	1.55	0.18
16	std	16.34	2126.45	1.74	0.19
Overlap:	min	0.00	502.00	1.00	0.01
14	25%	2.00	1574.50	1.00	0.05
Channels:	50%	2.00	3076.00	1.00	0.12
64	75%	6.00	5302.00	2.00	0.23
32	95%	14.00	7074.00	3.00	0.54
	max	494.00	7378.00	269.00	5.11

Table C.2: Metrics for varying overlap size for breakpoint detection.

model		distances<500	distances≥500	bps_pr_event	nearest_bp_val
Feature: 8 Channels: 64	count	88389.00	109.00	80286.00	345299.00
	mean	5.57	3364.73	1.10	0.36
	std	11.03	2122.62	0.44	0.28
	min	0.00	516.00	1.00	0.01
	25%	0.00	1484.00	1.00	0.14
	50%	4.00	3296.00	1.00	0.30
	75%	8.00	5024.00	1.00	0.51
	95%	16.00	6896.00	2.00	0.89
	max	492.00	7412.00	68.00	6.42
Feature: 4 Channels: 64 32	count	144626.00	471.00	126712.00	345299.00
	mean	5.70	3659.07	1.15	0.53
	std	12.13	2125.13	0.99	0.44
	min	0.00	504.00	1.00	0.01
	25%	0.00	1622.00	1.00	0.19
	50%	4.00	3716.00	1.00	0.42
	75%	8.00	5414.00	1.00	0.76
	95%	16.00	7070.00	2.00	1.34
	max	484.00	7388.00	226.00	6.88
Feature: 2 Channels: 64 32 16	count	165401.00	610.00	144557.00	345299.00
	mean	5.59	3667.79	1.15	0.65
	std	13.43	2123.47	1.19	0.59
	min	0.00	500.00	1.00	0.01
	25%	0.00	1703.00	1.00	0.23
	50%	4.00	3758.00	1.00	0.51
	75%	8.00	5432.00	1.00	0.91
	95%	16.00	7058.60	2.00	1.62
	max	496.00	7424.00	304.00	18.48
Feature: 1 Channels: 64 32 16 8	count	217326.00	865.00	189297.00	345299.00
	mean	5.05	3568.98	1.15	1.56
	std	12.51	2146.78	1.44	6.72
	min	0.00	504.00	1.00	0.01
	25%	0.00	1556.00	1.00	0.41
	50%	4.00	3428.00	1.00	1.10
	75%	8.00	5468.00	1.00	1.86
	95%	16.00	7085.60	2.00	4.04
	max	492.00	7448.00	423.00	1912.35

Table C.3: Metrics for varying feature size for breakpoint detection. The models all used a segment size of 16 and an overlap of 4.

Appendix D

U-Seg HP and non-HP Results

#	Train loss	Test Loss	K_down	U-Seg Encode	U-Seg Decode	Train Acc	Test Acc
0	0.195	0.119	9	[2,4]	[8, 2]	0.967, 0.530	0.967, 0.567
1	0.196	0.134	21	[2,4]	[8,2]	0.973, 0.498	0.956, 0.632
2	0.200	0.106	63	[2,4]	[8, 2]	0.975, 0.469	0.989, 0.332
3	0.179	0.127	9	[2,4], [4,8]	[16, 4], [8,2]	0.974, 0.556	0.961, 0.665
4	0.188	0.118	21	[2,4], [4,8]	[16, 4], [8,2]	0.977, 0.502	0.979, 0.503
5	0.211	0.124	63	[2,4], [4,8]	[16, 4], [8,2]	0.976, 0.430	0.979, 0.406
6	0.182	0.114	9	[2,4], [4,8], [8,16]	[32, 8], [16,4], [8,2]	0.975, 0.545	0.983, 0.387
7	0.197	0.125	21	[2,4], [4,8], [8,16]	[32, 8], [16,4], [8,2]	0.975, 0.471	0.973, 0.515
8	0.201	0.129	63	[2,4], [4,8], [8,16]	[32, 8], [16,4], [8,2]	0.974, 0.462	0.967, 0.525

Table D.1: The U-Seg segmentation model train with a total of 1000 reads, and tested with 500 reads.

#	Post process With Beam	Argmax
0	0.759, 0.214	0.848, 0.126
1	0.732, 0.237	0.811, 0.159
2	0.797, 0.193	0.928, 0.047
3	0.773, 0.198	0.845, 0.130
4	0.775, 0.200	0.894, 0.086
5	0.775, 0.194	0.902, 0.066
6	0.802, 0.172	0.917, 0.067
7	0.76, 0.203	0.875, 0.100
8	0.779, 0.193	0.873, 0.105

Table D.2: Evaluation of the U-Seg model both using the post processing algorithm (with mininum HP length of 15 and removing zero mistakes of length 2) and with Argmax without the post processing tool

Appendix E

HP Length: Confusion Matrices

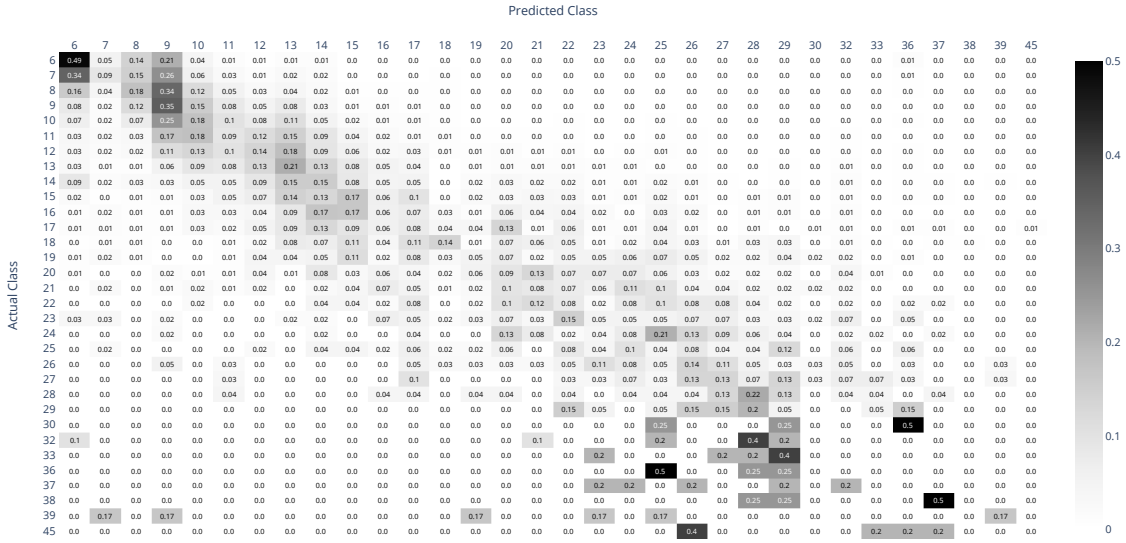


Figure E.1: Class-aware sampling: proportion of predictions in each class.

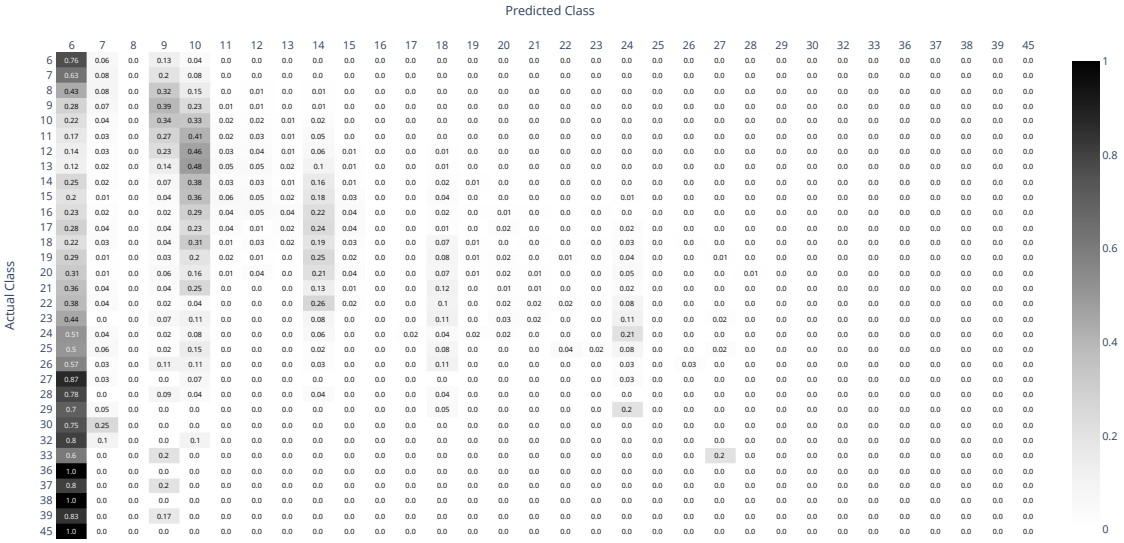


Figure E.2: Instance-based sampling: proportion of predictions in each class.

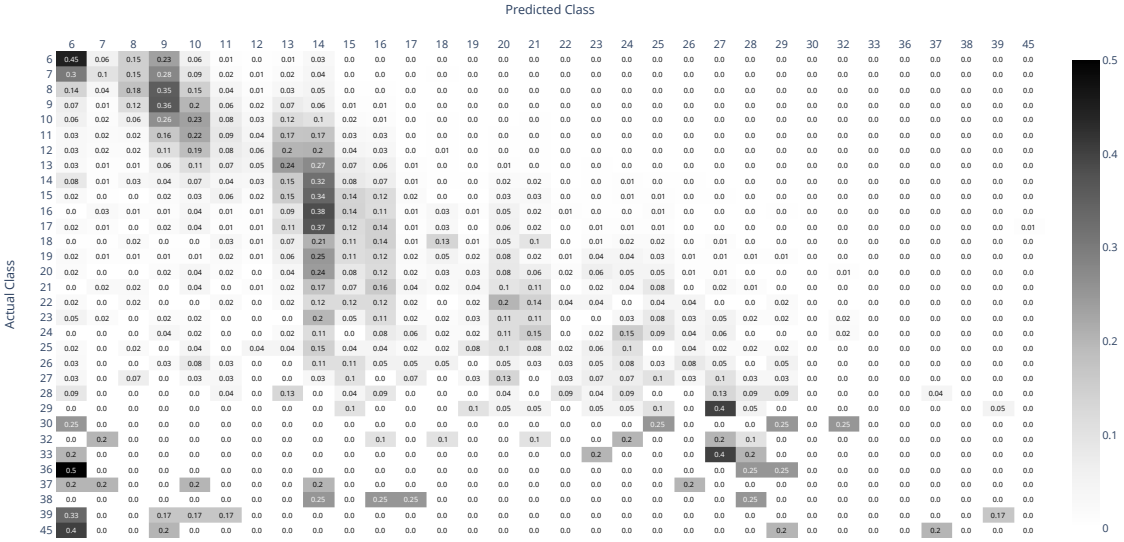
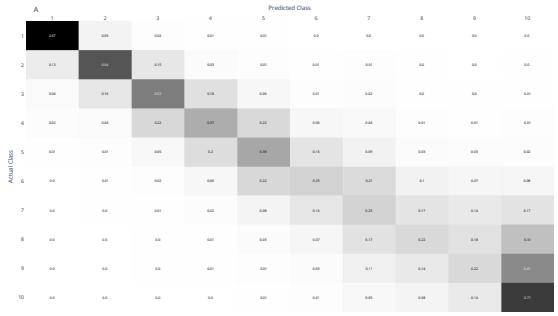


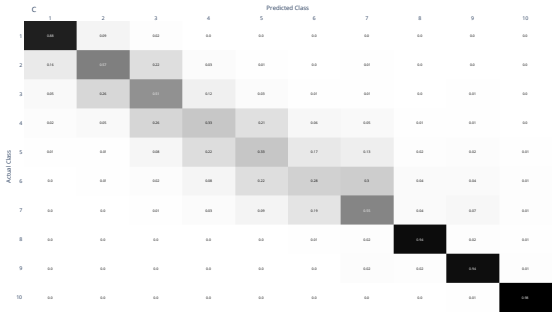
Figure E.3: Loss weighting: proportion of predictions in each class.

Appendix F

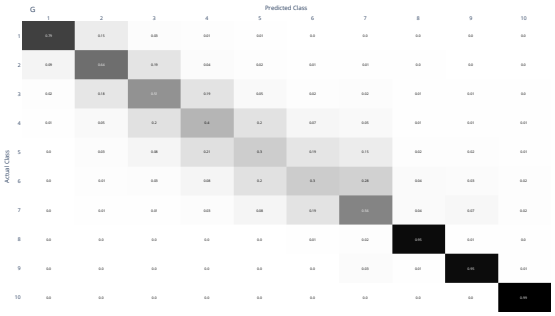
Improved HP Length: Per Base Confusion Matrices



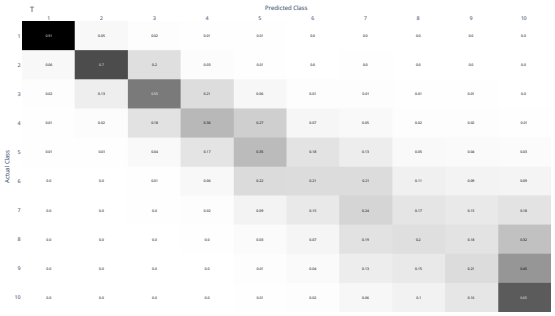
(a) A



(b) C

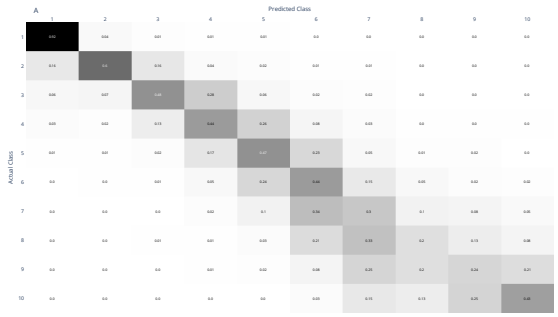


(c) G

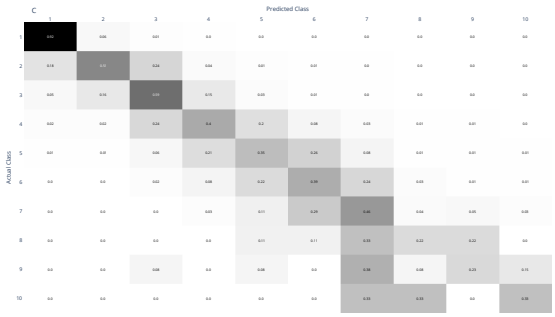


(d) T

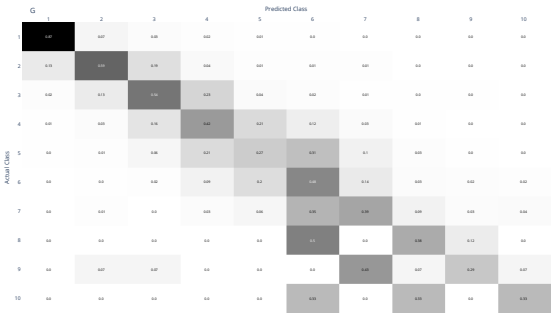
Figure F.1: Confusion matrices of bases ACTG for training set.



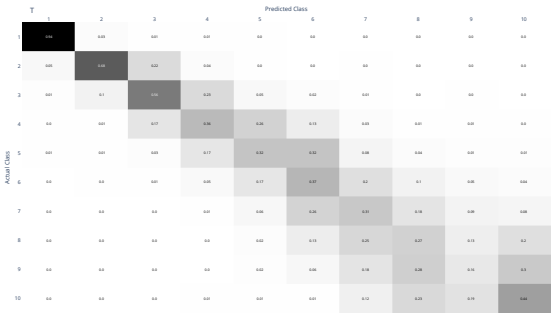
(a) A



(b) C



(c) G



(d) T

Figure F.2: Confusion matrices of bases ACTG for validation set.

Appendix G

Improved HP Length: Length Confusion Matrices

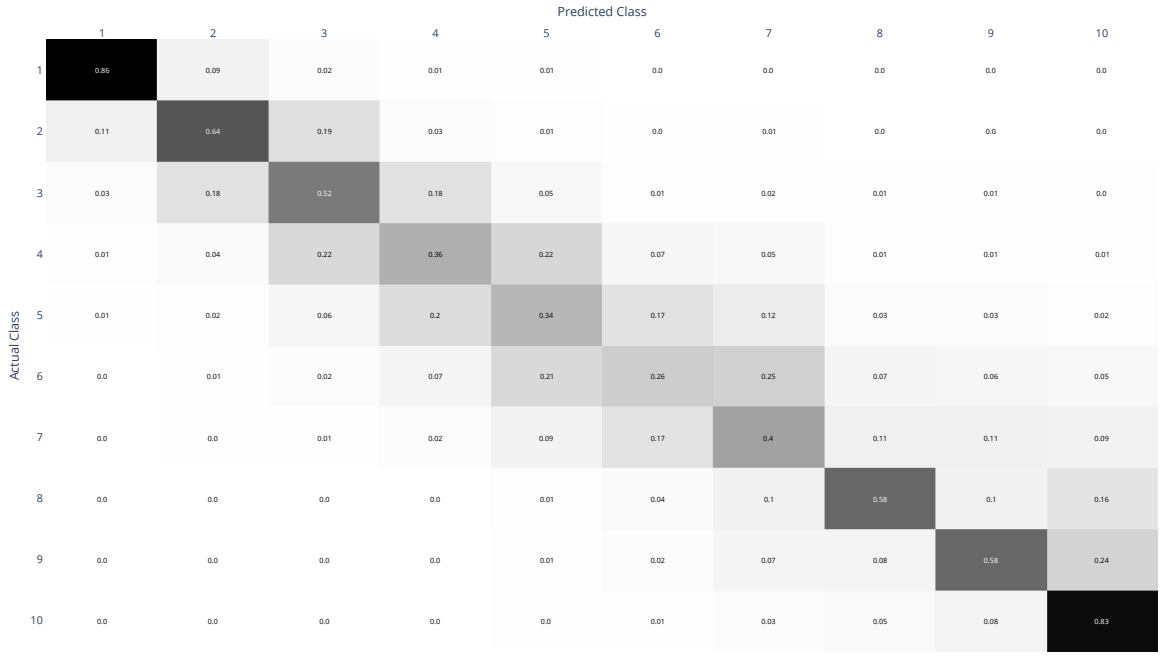


Figure G.1: Confusion matrix for training set.

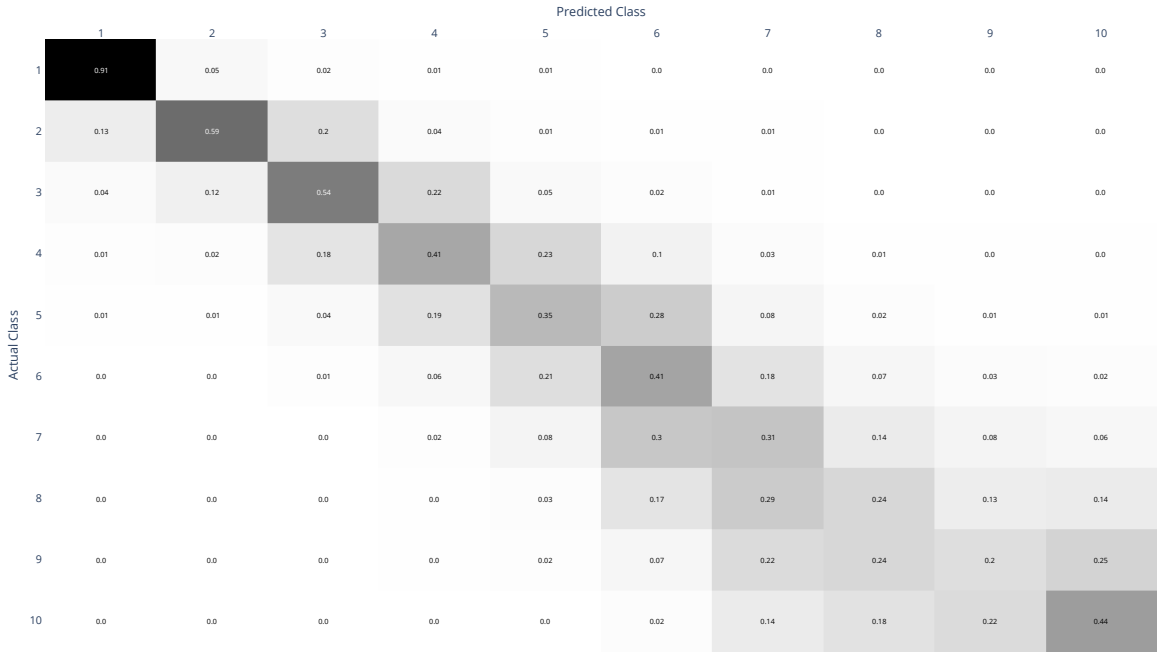


Figure G.2: Confusion matrix for validation set.