Improving Basecalling Accuracy With Transformers

Felix Gravila
Aalborg University
fgravi18@student.aau.dk

Miroslav Pakanec
Aalborg University
mpakan18@student.aau.dk

***Abstract*—DNA sequencing has recently undergone rapid improvements due to the Oxford Nanopore Technologies sequencing devices. These devices are fast and can read longer sequences than other sequencers, but have a lower accuracy due to the process of translating the measured electric signal into the corresponding DNA bases. This process is done using machine learning models called basecallers, which greatly impact the overall sequencing accuracy.**

Current basecallers process the electric signal sequentially, relying on recurrent layers and connectionist temporal classification for decoding. We propose an open source transformer-based model, *FishNChips*, which eliminates the need of recurrence by relying solely on attention. We compare it to our own implementation of a recurrent model, *Gravlax*, and show that *FishNChips* outperforms both *Gravlax* and the current state of the art basecallers.

***Index Terms*—ONT nanopore sequencing, artificial neural network, deep learning, transformer, attention**

I. Introduction

DNA sequencing is the process by which genetic material is transcribed into a sequence of nucleobases A,T,C,G, which encode the information that builds all living cells. Oxford Nanopore Technologies (ONT) [1] have caused a paradigm shift in the field by creating nanopore sequencing devices. Devices such as the MinION sequencer [2], which fits in the palm of a hand, pass the DNA through a biological membrane called a nanopore and measure how the electrical current changes with different bases. The genetic information is obtained by analysing the electrical signal afterwards. This process greatly improves sequencing speeds and maximum analysable sequence lengths at the cost of accuracy [3] compared to alternative methods such as Illumina, which are slower, have much shorter maximum read lengths, but are very accurate [4, 5].

Translating the electric signal into the series of corresponding bases, a process known as *basecalling*, is not a trivial task. The high error rate of nanopore sequencing (currently between 7-15% [6]) is caused both by measuring the signal and its translation to bases. The error rate of the former is due to high signal-to-noise ratio caused by several factors:

- structural similarity of bases causes small variability of electric current, especially since there are five bases inside the nanopore at a time of measurement resulting in $4^5 = 1024$ possible current values [7]
- electric current does not change with homopolymers (sequence of bases of the same kind) [8]
- non-uniform speed of the DNA passing through the nanopore [9]

The error rate of the latter is caused by the inability of the basecalling tools to map the electric signal into its correct base interpretation. In this paper, we focus on reducing the error rate of basecallers.

The nanopore technology and machine learning techniques go hand in hand. Programs that perform the task of translating the signal to a string of bases are called *basecallers*, and employ machine learning techniques to learn the mapping between the electric current and the corresponding sequence of bases.

The first basecallers started by segmenting the raw signal into a series of events, where each event corresponds to bases present at the nanopore in a given time. These basecallers then used Hidden Markov Models [10] to determine the final base sequence [11].

Modern models transitioned to translating the raw sequence directly to the sequence of DNA using approaches such as *Connectionist temporal classification* (CTC) [12]. This has been shown to be more effective than performing the extra step of event segmentation [6]. CTC allows the model not only to learn which base to pick, but also where in the signal it occurs. State of the art basecallers such as *Guppy*, *Chiron*, and *Bonito* obtain high accuracies by using CTC based sequence-to-sequence models [6, 13, 14]. Recent research has shown the effectiveness of replacing recurrent models with attention-based transformers that eliminate the need for recurrent layers altogether [15]. *SACall* [16] offers promising results by replacing the recurrent layers with self-attention, but keeping CTC as the decoder.

This paper will investigate the effectiveness of Transformers as basecallers. Since the current state of the art depends on recurrent models, we start by presenting *Gravlax*, our recurrent model based on Chiron. We then propose *FishNChips*, our basecaller that uses a Transformer to replace the recurrent layers of *Gravlax*, and show how such architecture improves basecalling accuracy. Finally, the last part of the paper discusses our results, the advantages and disadvantages of this approach, and possible improvements.

II. Preliminaries

A. Recurrent models

Popular state of the art basecallers employ recurrent neural networks (RNNs), which are capable of processing sequences of arbitrary lengths by applying transition functions on their internal hidden states for each element of the input sequence.

Long Short Term Memory (LSTM) improves memorisation of long term dependencies of recurrent models by propagating an additional cell state. It also uses specialised gates, which allow it to distinguish which information to remember and forget [17]. Appendix A Subsection C further expands on RNNs and LSTMs.

B. Convolutional neural networks

Convolutional neural networks (CNNs) are commonly used in models that handle images. By moving a learnable filter over the input, areas closely resembling the filter will have a higher activation than the less similar areas. Each filter forms a feature map, and multiple filters can be used. Each filter is able to represent different patterns.

Pooling layers are used for dimensionality reduction, only keeping the most important activations in each area of the input [18]. Appendix A Subsection B explains CNNs in greater detail.

C. Connectionist Temporal Classification

Connectionist Temporal Classification (CTC) is a loss function used to train sequence-to-sequence neural networks where the timing

of the outputs varies. In the case of basecalling, there is a relationship between events in the signal and the corresponding bases which preserves order, but the distance between these events is not constant.

CTC allows the model to learn the probability of a base occurring at each time step of the signal. Since the length of the input is much longer than the output, adjacent predictions of the same base are removed since they refer to the same base. This, however, does not allow the model to predict genuine adjacent bases. CTC solves this by introducing a blank symbol which the model predicts between genuine consecutive bases. The model can therefore predict adjacent identical bases since the removal of bases is performed before removing the blank symbols. CTC is further defined in Appendix A Subsection D.

D. Transformer and Attention

The transformer is a sequence-to-sequence model that improves on recurrent models by replacing recurrence with attention. It has an encoder-decoder structure. The encoder maps the source sequence $X = (x_1, x_2, \ldots, x_n)$ to a sequence of representations $Z = (z_1, z_2, \ldots, z_n)$. The decoder then uses Z to generate the target sequence $Y = (y_1, y_2, \ldots, y_m)$, one element at the time [15].

The main effective difference between transformers and recurrent networks is the fact that transformers do not need to iterate through the data one time step at a time. Instead, a transformer uses *Positional encoding* to distinguish time separation. Appendix A Subsection G expands on positional encoding.

A Transformer utilises attention to reason about relations between:

- **source-source -** which source signal values influence a particular source signal value
- **target-target -** which target base values influence a particular target base value
- **source-target -** which source signal values influence a particular target sequence base

For all the above mentioned cases, the transformer first computes *keys*, *values*, and *queries*. The intuition is that multiplying keys and queries results in an attention matrix, which marks the important information to be extracted from the values. For *source-source* all keys, queries, and values are constructed from X. *Target-target* constructs them from Y. Finally, *Source-target* constructs keys and values from Z and queries from Y:

$$\begin{aligned} K &= ZW_k \\ V &= ZW_v \\ Q &= YW_q \end{aligned} \tag{1}$$

The learnable weight matrices W_k, W_v and W_q of dimensionality $(d_{model} \times d_{model})$ allow the model to learn how to obtain K, V and Q from the source and target sequences. Given that these layers contain d_{model} neurons, the resulting shapes are $len_{source} \times d_{model}$, $len_{source} \times d_{model}$ and $len_{target} \times d_{model}$ respectively. Rather than performing attention directly on these matrices, it significantly improves accuracy to split them along their d_{model} dimension into H *heads* [15]. We then calculate *Scaled dot product attention* for each head K_h, V_h, Q_h:

$$Attention(K_h, V_h, Q_h) = Softmax(\frac{Q_h K_h^T}{\sqrt{d_{model}/H}})V_h \tag{2}$$

The dot product of Q_h and K_h^T results in a matrix of size $len_{target} \times len_{source}$. This matrix is then scaled, which according to [15], produces more stable gradients. $Softmax$ ensures that this matrix sums to one across its len_{source} axis. The result is an *attention weight* matrix, which marks how much each signal measurement in the source sequence influences a particular base in the target sequence (in case of source to target attention). The matrix V_h captures the value of a particular signal measurement itself. The scaled dot product attention output is a matrix $head_h$ of size $len_{target} \times d_{model}/H$.

Computation of every head is independent and can be performed in parallel. The resulting heads are concatenated and linearly transformed:

$$\begin{aligned} &Multihead(K, V, Q) \\ &\quad = Concat(head_1, \ldots, head_H)W_o \\ &\text{where } head_h = Attention(KW_k^h, VW_v^h, QW_q^h) \end{aligned} \tag{3}$$

The resulting matrix has size $len_{target} \times d_{model}$. This method is similar for source-source and decoder-decoder attention, where all keys, values, and queries are obtained from source sequence and target sequence respectively.

E. Transformer structure

The transformer consists of an encoder and a decoder. Since the dimensionality of the intermediate representation of the data between blocks is preserved to d_{model}, both encoder and decoder can be composed of multiple layers by stacking them.

Each encoder layer begins by applying multi head attention where all $keys$, $values$, and $queries$ are constructed from the source sequence. This is the "source-source" part of Section II-D. A point-wise feed-forward neural network is used on the time axis, enabling information to flow between the heads. This network consists of two dense layers. The first layer is tasked with reorganising features across heads, while the second layer always has d_{model} neurons, ensuring the dimensionality is kept between encoder layers.

The encoder layer uses residual connections, dropout and normalisation layers for the two sub-blocks.

The decoder operates in a similar fashion to the encoder, but adds an extra attention sub-block. In the first sub-block, Multi Head Attention is applied using queries, keys, and values from the target sequence. Its output is used as the query to the second Multi Head Attention sub-block, while keys and values are provided by the encoder. Finally, a point-wise feed-forward neural network is used over the output of the second sub-block. Just like in the encoder, all sub-blocks are wrapped in a residual connection and followed by dropout and normalisation layers.

III. Related work

Nanopore basecallers are still new and rapidly improving. Models such as Guppy and Chiron currently define the state of the art by using recurrent layers to capture the dependencies between signal points [6, 13]. A recurrent network lies at the core of these models. Chiron uses a stack of bidirectional LSTM layers while Guppy uses the RGRGR architecture which consists of alternating GRU and reverse GRU layers. GRU layers use a different recurrent cell designed to improve backpropagation through time, similarly to LSTM layers [19].

While recurrent layers have been the best way to handle time-based dependencies in the past, such tasks can be performed without recurrence [15, 20].

Recent research shows that deep 1-dimensional CNNs have potential to be just as powerful as RNNs at capturing time dependencies, while having the advantage of faster computation time due to the

possibility of executing in parallel [20]. Modern state of the art basecallers such as Bonito [14] use a purely convolutional architecture, skipping recurrence completely.

Additionally, Transformers have brought a novel approach to the task of machine translation by using attention to learn which parts of the input to pay attention to [15].

SACall [16] is a transformer-based basecaller. Like Chiron, it first uses convolution layers, followed by max-pool layers to reduce dimensionality. Instead of using the entire transformer, it only uses the encoder to compute the sequence of representations Z, which is then passed through a fully connected layer with a softmax activation to obtain a sequence of probability distributions. Finally, it uses CTC loss to directly train the encoder, instead of decoding Z with the decoder.

While the aforementioned basecallers use different architectures, they all use the CTC loss. Our transformer based model, *FishNChips*, deviates from this pattern by using the transformer decoder in order to decode Z. This loosens the strict relationships between input and output enforced by the CTC loss, enabling the model to have more flexibility in mapping the signal to the output.

IV. Data preprocessing

Training a sequence-to-sequence model requires having a dataset of signal windows with the corresponding DNA sequence as the ground truth. This is not a trivial task since the locations of bases in the signal are not obvious, because of the various lengths of the two sequences and the changing speed of the DNA passing through the nanopore. Raw data from sequencing is unlabelled and it is even unknown what organism it belongs to.

We use Taiyaki [21], a tool that aligns a sequence of DNA to the nanopore signal. First, the raw signal is basecalled using another basecaller to obtain an approximation of the ground truth. Since this approximation contains errors, it then needs to be aligned to a reference genome. A reference genome is the correct DNA corresponding to the samples being read. This reference is usually obtained using high accuracy sequencing tools. The approximation basecall is aligned to the reference using Minimap [22], which tells us which part of the reference is represented by the signal. We then use Samtools [23] to extract the corresponding reference, which can be used as the ground truth. We are able to do this due to the fact that we use copies of known bacteria genes with low variability, so we know that the information represented in them is identical to the reference. Finally, we use Taiyaki to match the reference to the signal. Taiyaki returns a list of indices which match each base of the reference to the signal [24].

We normalise each signal by subtracting its mean and dividing by its standard deviation in order to account for possible differences between the sensor sensitivities:

$$Signal_{norm} = \frac{Signal - \mu(Signal)}{\sigma(Signal)}$$

V. Gravlax - recurrent model approach

Gravlax is our *recurrent* basecaller based on the architecture of Chiron. We call it recurrent because it uses recurrent LSTM layers in its architecture. Recurrent models such as Guppy and Chiron have significantly improved basecalling accuracies in the last decade [6]. Because Guppy and Chiron still achieve state of the art accuracy, recurrent models remain an important category of basecallers. We begin by studying a recurrent model since its ubiquity makes it a good candidate for our baseline.

Recurrent models are flexible with regard to the lengths of the input and output sequences, since the same weights are applied recurrently over the time axis. However, because we want to create a generic basecaller, we only want to give it a small portion of the signal at once. Therefore, we split the signal into windows of 300 signal measurements, while neighbouring windows overlap by 270 signal measurements. After data preprocessing we know the positions of each base in the signal. Using this we can construct the corresponding labels.

A. Architecture

Inspired by Chiron, *Gravlax* has 5 residual blocks, each containing 3 convolutional layers [25] of kernel size 1,3,1 respectively and 256 filters, and a residual connection as described in subsection A. We added batch normalisation layers between the residual blocks [26]. Additionally, a max-pooling layer between residual block number 3 and 4 reduces the length of the sequence to half. Since CTC requires an input length $len_{input} \geq 2 \times len_{output}$ we cannot reduce the signal more than this.

We follow the residual blocks with three bidirectional recurrent layers using LSTM cells with 250 units each. Instead of using LSTM cells with internal batch normalisation, we added batch normalisation layers between the bidirectional LSTM layers. Finally, we employed the CTC loss function to compute the loss. Figure 1 shows the architecture of *Gravlax*.

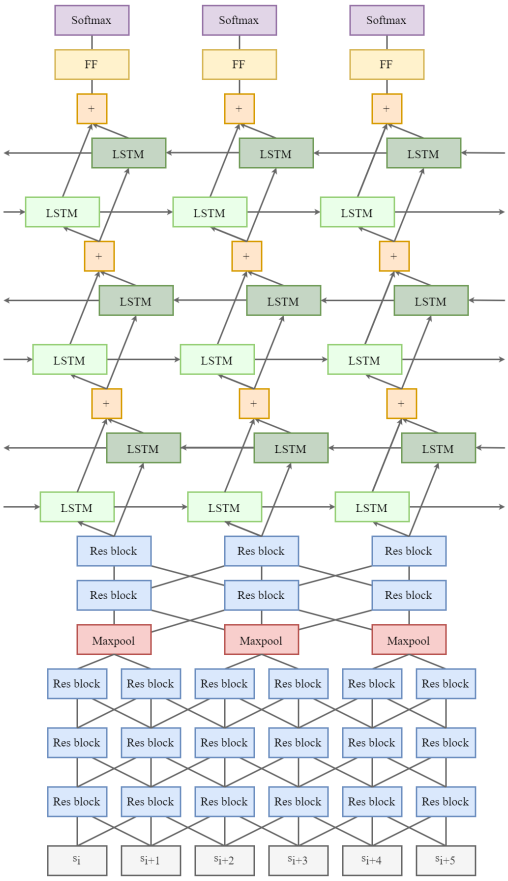

Fig. 1. Architecture of *Gravlax*

B. Training

We use *adam* with learning rate 0.001 as a training optimiser due to its popularity [27].

An *epoch* typically refers to a pass over the entire dataset while training. Because our dataset is too large, we define an epoch as training over $10,000$ windows split into batches of 32. After each epoch ends, we validate the model by using data outside the training set. We measure the validation accuracy using $editdistance$, an algorithm which computes the minimum number of character changes required to transform one string into another. In our case, we average the edit distances between each model prediction and its ground truths. We save the best model based on the average edit distance on a batch of 500 windows. Training stops when no improvement has been measured over 300 epochs.

C. Inference

To obtain a prediction, a signal window is first passed through the model, which yields a sequence of softmax outputs. We then perform *greedy decoding* by taking the most likely character at each position. We obtain the predicted output by removing adjacent characters and the blank symbols. The final DNA sequence is assembled using our custom assembler. Each window's prediction is aligned to the previous n predictions based on similarity. After all window predictions are aligned, consensus is performed and the most likely base is picked for each position. The number of previous predictions to be matched against depends on the window overlap, as less overlap means fewer overlapping windows. We use a stride of 30 (overlap of 270 signal measurements) and $n = 5$ windows of assembly overlap used for alignment.

$$
\begin{array}{l}
CGTTTAGTATTGGGTAGAGAA \\
\quad GTTTAGTATTG\_GTAGAGAAACC \\
\quad\quad TTGTATTGGGTAGAGAAACCAGG \\
\quad\quad\quad AGTATTGGGTAGAGAAACGAGGTG \\
\quad\quad\quad\quad ATTGGGTAGAGAAACCAGGTGA \\
\hline
CGTTTAGTATTGGGTAGAGAAACCAGGTGA
\end{array}
$$

D. Insight

We observed that a fundamental problem of our recurrent model is that it is very hard for it to learn how to differentiate between information observed 30 time steps ago and 50 time steps ago. In order to remember the neighbourhood of bases present in the nanopore and remain unbiased towards previous measurements, the model needs to remember 20-30 time steps, but forget everything else. When investigating the aligned predicted windows in the assembling process, we observed that this is not the case. *Gravlax* commonly changes its prediction of a base in the centre of a window compared to its neighbours. Since our chosen input window size contains 300 signal measurements, the number of bases per prediction we expect is ~25. Considering the physical properties of the nanopore device, any single base is affected by its neighbouring 4 bases. Consequently, bases located in the centre of a prediction should not be affected by the end of the signal (12 bases away). The only differences between adjacent windows are the ends of the signal; it must therefore mean that the model takes them into account even when it should not.

$$
\begin{array}{l}
CGTTTAGTATTGGGTAGAGAA \\
\quad GTTTAGTATA\_GGTAGAGAAACC
\end{array}
$$

We conclude that the recurrent layers are often unable to distinguish relevant signal measurements and disregard far away information.

VI. *FishNChips* - Transformer approach

Based on the intuition gained from exploring *Gravlax*, we create a basecaller which uses a transformer to perform the sequence-to-sequence translation. As opposed to the recurrent model which struggles with discerning long-term dependencies, the positional encoding utilised by the transformer enables it to more accurately keep track of the neighbouring relevant data. Rather than recurring over the source sequence of signals and choosing which information to memorise and which to forget, the transformer is able to look at the entire source and target sequences as a whole and identify relevant parts.

As opposed to SACall [16], we use the entire transformer. While it is not common to use an encoder-decoder structure for basecalling, we believe that the extra context offered by the target language in the process of decoding would improve the ability of the model to learn details about the data being basecalled. In the case where the previous 10 bases are related to the bases which follow, *Gravlax* might be able to remember that it has seen the bases. However, the *Gravlax* architecture has difficulty memorising their order and distance from the current time step. SACall is able to mostly solve this problem purely in the encoder, however we show that having the decoder directly apply attention using the so-far decoded bases further improves accuracy.

A. Encoder Architecture

Similarly to *Gravlax*, *FishNChips* utilises the same 5 residual blocks, together with max-pooling to extract relevant features from the source sequence. Because *FishNChips* does not use CTC, we can increase the pooling kernel size and reduce the source sequence more compared to what was possible in *Gravlax*. Our experiments show that a max-pooling layer with kernel size 6 between residual blocks 3 and 4 achieve the best accuracy.

The CNN layers also embed the signal into our d_{model} space. Positional encoding is then added to the CNN output.

Our findings show large accuracy gains by increasing the number of heads used during Multi Head Attention. We found that 25 heads perform the best in conjunction with a d_{model} of 250, picked such that $d_{model} \mod H = 0$. Further improvements can be achieved by increasing both d_{model} and H, however, we use these numbers for a similar model size to *Gravlax*. The multi head attention mechanism of *FishNChips* is shown on Figure 2.

We stack $N = 4$ encoder layers to form the encoder. We found that reducing the number of blocks to 2 yielded a small improvement when testing models with 8 attention heads, however this did not remain true when the number of heads was increased.

We always use $d_{ff} = 2 \times d_{model}$ neurons in the first dense layer of the point-wise feed-forward network.

Finally, since we always operate on signal windows of fixed length, there is no need to use start and end tokens in the source sequence.

B. Decoder Architecture

Our target language is a sequence of bases. Because the target sequences have variable lengths, we alter it before performing decoding by adding a start and end token and padding it to a fixed length $len_{target} = 100$. We picked this length to ensure that no training example exceeds it.

We use an *Embedding* layer in order to encode the target sequences into our desired d_{model}. This layer takes the positive integers which

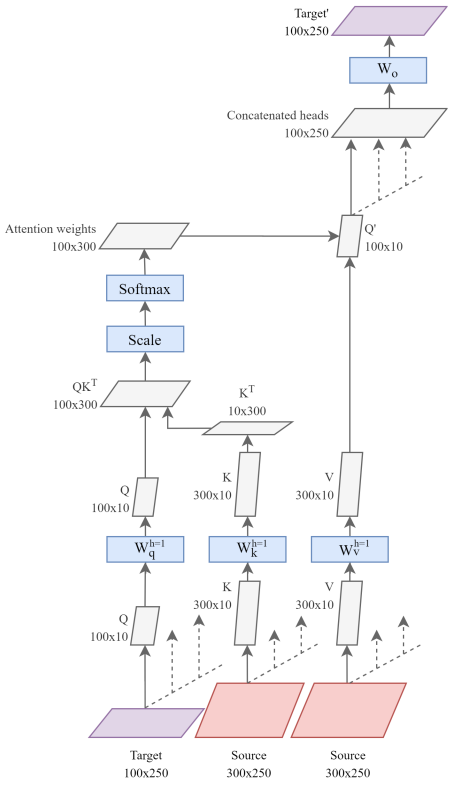

Fig. 2. Structure of the multi head attention mechanism

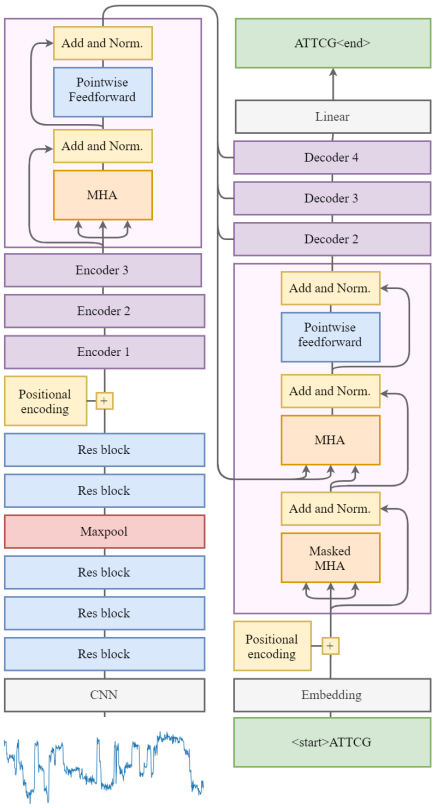

Fig. 3. Architecture of *FishNChips*

are used to represent the tokens in the target language and converts them into dense vectors of fixed size of d_{model} [28].

Our decoder follows the same parameters as the encoder. We use 25 heads for both Multi Head Attention sub-layers in each decoder layer. Similarly, we stack $N = 4$ decoder layers.

The output of the decoder passes through a final dense layer with 7 neurons corresponding to our target alphabet:

$$\Sigma = \{A, T, C, G, < start >, < end >, < pad >\}$$

The architecture of *FishNChips* is represented by Figure 3.

C. Training

Teacher forcing is used during training. A mask is used to hide any future tokens, which includes both the future bases in the sequence which the model needs to predict, and the padding. A separate mask is created for each iteration of the prediction, such that the model gets the opportunity to make a prediction for each character. Without the mask, the model would be able to just look at the next token and predict that directly. By using masking we ensure that the model trains properly. Appendix A Subsection H describes teacher forcing.

Transformers are especially difficult to train, requiring a period of learning rate warm-up. We use the *Adam* optimiser and gradually increase the learning rate until it peaks after 4000 batches. Afterwards, the learning rate slowly decays. We use the formula presented in [15]:

$$lrate = d_{model}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

D. Inference

Inference is performed iteratively by passing the source signal sequence to the encoder and a list only containing the start token as input to the decoder. The decoder outputs the most likely token that follows, which is appended to the decoder input in the next iteration. This process is repeated and the decoder predicts a new token each iteration until it outputs an end token, in which case the inference ends.

In order to optimise the process, we cache the encoder output between iterations of every example. This reduces inference time by half.

We predict the entire read similarly to *Gravlax* – the raw signal is split into windows which are then individually decoded using *FishNChips*. Finally, we assemble the outputs using our assembler.

VII. Experiments

A. Data

We use signal data obtained using an Oxford Nanopore Technologies MinION device (version R9.4.1) from bacterial samples containing the 16S-23S sequence from the following bacteria [29]:

- Bacillus subtilis
- Enterococcus faecalis
- Escherichia coli
- Lactobacillus fermentum
- Listeria monocytogenes
- Pseudomonas aeruginosa
- Salmonella enterica
- Staphylococcus aureus

The 16S-23S sequence is of special importance since it forms the ribosome, an important molecule which exists in any living cell. Due to its ubiquity, it is commonly used for bacterial identification.

Each signal represents approximately 4200 bases and consists of around 50000 signal measurements. The signals start and end before the DNA begins to pass through the nanopore, therefore, the beginning and the end of each signal does not contain any information. Additionally, the DNA contains leading and trailing barcodes used for identification added during amplification. The barcodes are not represented in the reference genome and therefore not matched to the signal, allowing us to cut the signal based on the first and last indices provided by Taiyaki.

From these reads we generate signal windows and corresponding labels as described in Section IV. It is not feasible for us to pre-generate all the training data beforehand due to the amount of data we have. We therefore use a generator pipeline which maintains a pool of training examples from at least 5 random reads. We then shuffle these examples before using them to train the model. This ensures both that a batch will not contain too similar data, and that a deformed read does not significantly affect the weights.

B. Implementation

We implemented both of our models in *Tensorflow 2.1* [30]. We used the *ctc_batch_cost* function implemented in *Keras* as the loss function to *Gravlax* [31]. For the embedding layer used in the *FishNChips* decoder we used the Keras Embedding layer [28]. We have implemented our own greedy CTC decoding function. Our Transformer code was adapted from the Tensorflow Transformer tutorial [32].

We train on an nVidia V100 GPU. One epoch of training consisting of 1000 batches of 32 windows each passes in 2 minutes.

C. Accuracy metric

We calculate the accuracy based on entire reads. We split a new raw signal into windows which we pass through the model. The assembled window predictions represent the models prediction of the whole signal. We use minimap to align our prediction to the reference. The alignment tells us the length of the matching sequence and the number of correctly identified bases and mismatches. We define our per-read accuracy as:

$$Accuracy = \frac{\textit{number of matching bases in the alignment}}{\textit{the length of the alignment}}$$

We average the accuracy over 200 reads. We consider predictions that have not been able to be matched by minimap to have accuracy 0.

D. Results

This section will cover the results we obtained by training and testing on the pool of all our 8 bacteria. While representing the same DNA, testing is done on new reads which have never been used during training.

Table I shows the results we have obtained using *Gravlax*. Our baseline, A, has very similar parameters to Chiron. The lower accuracy we obtained could be due to multiple factors. We use batch normalisation only between LSTM layers and not inside the cells. Chiron also trains on variable windows sizes 200, 400 and 1000, potentially increasing its robustness.

Adding a max-pooling layer with kernel size 2 between CNN blocks 3 and 4, increases our accuracy by 5% in B. We use this CNN architecture going forward. C further improves accuracy by removing batch normalisation.

In *C big*, we attempt to improve our accuracy by increasing the size of *Gravlax*. We approximately double the size of both filters used in CNN layers and the number of units in the LSTM cells, which dropped our accuracy.

We finally attempt to use the trained model to predict the entire raw signal of length 50,000 in the *C whole read* experiment. This means that we did not have to perform any preprocessing on the signal besides normalising it, and the decoded output did not need to be assembled. This was faster than the assembly method due to not having to repeat computations on overlapping windows.

Table II shows the hyperparameter tuning and accuracy of *FishNChips*. Experiments A, B, C and D show that 2 attention blocks are ideal for a small number of heads. *C big* shows that increasing the d_{model} improves the accuracy. However, we keep the d_{model} around 250 to achieve a comparable number of parameters to *Gravlax*.

Experiments E, F, G and H demonstrate how increasing and decreasing the number of attention heads impacts the accuracy. We observed that 25 heads achieves the highest accuracy of 93.41%.

I and J show it is possible to increase the number of heads further when using a higher d_{model}.

In K, L, M, N, O and P we added residual CNN blocks in front of the encoder. K, L, M show how different number of residual blocks impact accuracy. In experiment N we try using 2 attention blocks since they yielded the best accuracy in previous experiments. In the current configuration however, 2 blocks performed worse. Finally, O and P show how increasing the max-pooling kernel increases the accuracy as well. A max-pooling kernel of 6 achieved the best accuracy of 96.98%. We assume this is because it reduced the target sequence such that $len_{target} \approx len_{source}$.

In *P big* we set d_{model} to 1015, increase the number of blocks to 5 and used 35 heads. This yielded the best accuracy of 98.68%.

E. Performance of other basecallers

Table III shows the accuracies obtained by Guppy, Chiron and Bonito on our data using our accuracy metrics. Our aforementioned results are obtained by basecalling reads which have had the leading and trailing empty signal and barcodes removed. This would give us an unfair advantage, since the basecallers operate on the complete read. To be able to compare, we create a basecaller based on our best model. We don't do any data preprocessing or filtering. Raw signal is read from the file, normalised and split into windows which are predicted and assembled.

VIII. Discussion

Our experiments show that *FishNChips* performs better than the current state of the art basecallers. *FishNChips* outperformed Chiron, Guppy and Bonito on their pre-trained models taking into account the additional difficulties pertaining basecalling the entire raw signal.

Comparing the best accuracies of *Gravlax* and *FishNChips* also shows that, given the same data preprocessing and CNN layers, the transformer significantly outperforms bidirectional LSTM layers and CTC.

We discovered that using CNN layers and the size of the max-pooling kernel greatly affects the accuracy of our models. *Gravlax* benefited from adding a max-pool layer to its architecture. Similarly, *FishNChips* yielded better results both after adding the residual convolution blocks and when increasing pooling. While [15] claims that the transformer eliminates the need of convolutional layers, we observe that the CNN blocks greatly help with the initial filtering and dimensionality reduction of the raw signal.

Name	CNN Blocks	Filters	LSTM Units	Batch Norm	Maxpool	Accuracy
A	5	256	250	T		85.10%
B	5	256	250	T	T	90.79%
C	5	256	250		T	93.15%
C big	5	512	512		T	86.73%
C whole read	5	256	250		T	91.35%

TABLE I
Gravlax RESULTS WITH EVALUATION ON THE SAME BACTERIA AS TRAINING

Name	D_Model	Blocks	Heads	CNN Blocks	MaxPool Kernel	Accuracy
A	256	1	8			68.54%
B	256	2	8			90.74%
C	256	4	8			81%
D	256	8	8			0%
C big	1024	4	8			90%
E	256	4	4			85%
F	256	4	16			88.7%
G	250	4	25			93.41%
H	256	4	32			92.62%
I	1000	2	25			93.86%
J	1000	2	50			94.37%
K	250	4	25	5	2	96.64%
L	250	4	25	4	2	95.53%
M	250	4	25	6	2	94.82%
N	250	2	25	5	2	91.66%
O	250	4	25	5	3	96.60%
P	250	4	25	5	6	**96.98%**
P big	1015	5	35	5	6	98.68%

TABLE II
FishNChips RESULTS WITH EVALUATION ON THE SAME BACTERIA AS TRAINING

Model	Accuracy
Chiron DNA_default	84.03%
Guppy dna_r9.4.1_450bps_hac	92.83%
Bonito dna_r9.4.1	93.46%
Gravlax C	81.44%
Gravlax C whole read	86.38%
FishNChips	95.80%

TABLE III
PERFORMANCE OF OTHER BASECALLERS ON OUR DATA.

A. Basecalling unknown bacteria

In order to see how well the models generalise on DNA they have not seen before, we re-train our best models on 7 of the 8 bacteria. We randomly choose Escherichia Coli for testing and use the rest for training. After each epoch ends, we use 300 windows of the excluded bacteria for validation. We predict the individual windows and obtain an accuracy metric by computing the edit distance to their corresponding labels. We average the edit distance over the 300 reads to obtain the *average edit distance*. We save the best model based on this metric and stop training when no improvement has been seen in over 300 epochs. Table IV shows the accuracies obtained inside and outside the training dataset. Similarly, table V tests on Escherichia Coli and Salmonella enterica while training on the rest.

We observed that the lower the accuracy, the worse our assembler performs. The results in this subsection have been obtained both with and without the assembler. In order to obtain a prediction without an assembler, the model predicted non-overlapping windows, and the complete DNA was formed by concatenating all the results. Appendix C Subsection D shortly describes this.

Basecalling on unknown bacteria drops the accuracy significantly. We believe that more varied training data would increase the ability of the model to generalise. We perform k-mer analysis which shows that a very small subset of possible k-mers are represented in the reference DNA. Appendix C Subsection H presents this analysis.

While the drop in *FishNChips* accuracy between known and unknown DNA might be interpreted as the inability of the model to generalise, we do not believe this to be the case. *Gravlax* suffered an even larger drop although its layers and method of operation closely resemble state of the art basecallers. We therefore conclude that the discrepancy must stem from the low variability of our data or from our pipeline, which are common elements between *FishNChips* and

Model	Accuracy on 7 trained bacteria	Accuracy on 1 untrained bacteria
Gravlax (C) (concat)	93.37%	85.60%
FishNChips (P) (concat)	96.61%	90.50%
FishNChips (P) (assembled)	98.45%	89.30%

TABLE IV
MODEL EVALUATION ON BACTERIAL DNA IT WAS TRAINED ON AND BACTERIAL DNA IT WAS NOT TRAINED ON IN RATIO 7:1.

Model	Accuracy on 6 trained bacteria	Accuracy on 2 untrained bacteria
FishNChips (P) (concat)	97.08%	61.06%

TABLE V
MODEL EVALUATION ON BACTERIAL DNA IT WAS TRAINED ON AND BACTERIAL DNA IT WAS NOT TRAINED ON IN RATIO 6:2.

Gravlax.

The fact that the accuracy on 2 untrained bacteria dropped even more proves the fact that our model suffers from lack of data.

We observed that when validating on untrained bacteria, the accuracy on trained bacteria has also risen. We believe this stems from the following:

- We only use 3 reads for validation, meaning randomness plays a larger role in when the model is saved. Validating on unknown DNA minimises the effect of randomness since the power of the model to generalise is much more important.
- Removing a bacteria from a training set implies less variability in data. This forces the model to generalise less and allows it to better memorise windows it has encountered previously and hence potentially overfit.

Finally, the accuracy on trained bacteria seems to drop the more bacteria are being used. We consider it necessary to train on more bacteria in order to improve the ability of the model to generalise, but it is uncertain where the accuracy on trained bacteria would converge, and if it would continue to decrease.

B. Time complexity

The time complexity of transformers is $O(n^2 \cdot d)$ per layer, where n is the length of the sequence and d the representation dimension, while the complexity of recurrent models is $O(n \cdot d^2)$ [15]. This makes it prohibitive for transformers to be scaled to long sequences at once, while much easier for recurrent models. This also makes increasing d_{model} a more attractive proposition for *FishNChips*, while it would slow *Gravlax* considerably.

Finally, our experiments with *Gravlax* show that a recurrent model still performs well even when basecalling the entire signal at once. While less accurate, we consider this to be a great advantage in situations where real time performance is more important than accuracy.

IX. CONCLUSION

In this paper we investigated the shortcomings of recurrent models and the potential of transformers for the task of basecalling. By examining the output of our recurrent model *Gravlax*, we discovered its difficulty in distinguishing the importance of far and near dependencies. To overcome this, we implemented a transformer based basecaller, *FishNChips*, which is able to overcome this by having the ability to pay attention to any element of the input sequence. We show that *FishNChips* performs significantly better compared to *Gravlax* in our testing conditions. Furthermore, we show that *FishNChips* obtains accuracies better than the current state of the art.

Additionally, we tuned our models and experimentally showed how changing the parameters affects the final result.

In conclusion, the paper achieved its goal of improving basecalling accuracy by implementing a model combining a deep residual convolutional neural network with a transformer.

X. FUTURE WORK

A. Parameter Tuning

Given extra time, there are a number of additional elements we would have liked to explore related to our work. Since we strongly consider the low variability of data to be a root cause of the reduced ability of our models to generalise, we would like to train our models on more varied DNA. Additionally, we would have liked to be able to implement varying window sizes for training since we believe it would improve robustness.

We would consider it important to analyse the correctness of our data preprocessing by using alternatives to Taiyaki such as Nanopolish eventalign and Tombo re-squiggle [33, 34] to label the raw signal.

While we have attempted to tune a number of parameters important to the models, there are even more which we have not touched.

We have discovered that pooling affects the obtained accuracy greatly. However, we have always used maximum pooling, and we have always placed the pooling layer between the 3rd and 4th convolutional blocks. We would like to experiment with average pooling, and different positions for the pooling layer. Since *FishNChips* can be pooled further than *Gravlax*, we would also experiment with multiple pooling layers of filter size 2, which might enable the model to extract features even better.

Considering the residual blocks, we have used the 1-3-1 pattern as described in paper [35]. Guppy, while not having a deep residual CNN architecture, uses a convolutional layer with filter size 11 and stride 5 to extract longer term dependencies and condense data simultaneously. We therefore believe that it is possible to further improve performance by increasing filter sizes or improve performance by increasing the stride.

Finally, we have only used the activation functions as recommended by papers [35] and [15]. Since activation functions have a large impact on model performance, we would like to see how different activation functions would perform. Specifically, we would like to investigate whether replacing our ReLU activation functions with ELU would improve convergence and accuracy by enabling the model to use the gradients for negative inputs [36].

B. Training data distribution

We noticed a correlation between the average per-bacteria accuracy and the representation of said bacteria in the training dataset. We think a better training method would be to ensure no bacteria is overly used in training.

C. Simplifying transformer architecture

Paper [37] argues that, while the multi head attention layers of the transformer are often considered to be the dominant components of its success, the followed point wise feed forward networks contains most of the parameters. This suggests that the role of these networks is important for the attention mechanism. Therefore, they propose an *attention layer*, which combines a multi head attention layer and a feed forward layer and hence simplifies the transformers architecture without suffering from accuracy drops.

D. Restricted attention and adaptive attention span

The proposed transformers attention mechanism computes a square attention matrix, which determines the importance of each element of one sequence to each element of another sequence. Consequently the time complexity of each multi head attention layer is $O(n^2 \cdot d_{model})$, where n is the sequence length. To improve scalability with large sequences, attention can be restricted, considering only the neighbourhood of size r. This reduces the time complexity to $O(n \cdot r \cdot d_{model})$ [15].

The neighbourhood r can be viewed as an *attention span* of the transformer. The transformer assumes that the attention span is constant for every head h, however [38] shows that this is not the case as some heads need to focus on the recent history, while others require information from the entire sequence. Furthermore, [38] proposes *adaptive attention span* mechanism, capable of learning the optimal attention span of every head. Such mechanism allows the transformer to use longer input sequences at no additional computational or memory cost.

Code

FishNChips and *Gravlax* are open source and the code can be found at https://github.com/fishnchips-basecaller/fishnchips.

Acknowledgements

We would like to thank our supervisors, Katja Hose and Thomas Dyhre Nielsen for their guidance.

We also thank Mads Albertsen, Rasmus Kirkegaard, and other members of Albertsen Lab who have answered all our questions pertaining biology, and who have provided us with the bacterial data used in this project.

References

[1] Oxford Nanopore Technologies. *Nanoporetech Community*. URL: https://nanoporetech.com (visited on 05/21/2020).

[2] Oxford Nanopore Technologies. *MinION*. URL: https://nanoporetech.com/products/minion (visited on 05/21/2020).

[3] Oxford Nanopore Technologies. *How it works*. URL: https://nanoporetech.com/how-it-works (visited on 05/21/2020).

[4] Illumina. *Illumina sequencing platforms*. URL: https://www.illumina.com/systems/sequencing-platforms.html (visited on 05/07/2020).

[5] Illumina. *NovaSeq™ 6000 Sequencing System*. URL: https://www.illumina.com/content/dam/illumina-marketing/documents/products/datasheets/novaseq-6000-system-specification-sheet-770-2016-025.pdf (visited on 05/07/2020).

[6] 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.

[7] Vladimír Boža, Broňa Brejová, and Tomáš Vinař. "DeepNano: Deep recurrent neural networks for base calling in MinION nanopore reads". In: *PLOS ONE* 12.6 (June 2017), pp. 1–13. DOI: 10.1371/journal.pone.0178751. URL: https://doi.org/10.1371/journal.pone.0178751.

[8] Peter Sarkozy, Ákos Jobbágy, and Peter Antal. "Calling Homopolymer Stretches from Raw Nanopore Reads by Analyzing k-mer Dwell Times". In: *EMBEC & NBC 2017*. Ed. by Hannu Eskola et al. Singapore: Springer Singapore, 2018, pp. 241–244. ISBN: 978-981-10-5122-7.

[9] Gerald M. Cherf et al. "Automated forward and reverse ratcheting of DNA in a nanopore at 5-Å precision". In: *Nature Biotechnology* 30.4 (Apr. 2012), pp. 344–348. ISSN: 1546-1696. DOI: 10.1038/nbt.2147. URL: https://doi.org/10.1038/nbt.2147.

[10] Zoubin Ghahramani. "An introduction to hidden Markov models and Bayesian networks". In: *Hidden Markov models: applications in computer vision*. World Scientific, 2001, pp. 9–41.

[11] Matei David et al. "Nanocall: an open source basecaller for Oxford Nanopore sequencing data". In: *Bioinformatics* 33.1 (Sept. 2016), pp. 49–55. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btw569. eprint: https://academic.oup.com/bioinformatics/article-pdf/33/1/49/25140287/btw569.pdf. URL: https://doi.org/10.1093/bioinformatics/btw569.

[12] Alex Graves et al. "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks". In: *Proceedings of the 23rd international conference on Machine learning*. 2006, pp. 369–376.

[13] Haotian Teng et al. "Chiron: translating nanopore raw signal directly into nucleotide sequence using deep learning". In: *GigaScience* 7.5 (Apr. 2018). giy037. ISSN: 2047-217X. DOI: 10.1093/gigascience/giy037. eprint: https://academic.oup.com/gigascience/article-pdf/7/5/giy037/28599334/giy037.pdf. URL: https://doi.org/10.1093/gigascience/giy037.

[14] Oxford Nanopore Technologies. *Bonito*. URL: https://github.com/nanoporetech/bonito (visited on 05/04/2020).

[15] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[16] N. Huang et al. "An attention-based neural network basecaller for Oxford Nanopore sequencing data". In: *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 2019, pp. 390–394.

[17] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[18] Yann LeCun, Yoshua Bengio, et al. "Convolutional networks for images, speech, and time series". In: *The handbook of brain theory and neural networks* 3361.10 (1995), p. 1995.

[19] Junyoung Chung et al. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling". In: *CoRR* abs/1412.3555 (2014). arXiv: 1412.3555. URL: http://arxiv.org/abs/1412.3555.

[20] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling". In: *arXiv preprint arXiv:1803.01271* (2018).

[21] Oxford Nanopore Technologies. *Taiyaki*. URL: https://github.com/nanoporetech/taiyaki (visited on 05/05/2020).

[22] Heng Li. *minimap2*. URL: https://lh3.github.io/minimap2/ (visited on 01/06/2020).

[23] Sequence Alignment/Map Format Specification. *samtools*. URL: https://samtools.github.io/hts-specs/SAMv1.pdf (visited on 01/06/2020).

[24] Oxford Nanopore Technologies. *Walk-through*. URL: https://github.com/nanoporetech/taiyaki/blob/master/docs/walkthrough.rst (visited on 05/05/2020).

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[26] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

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

[28] Tensorflow. *tf.keras.layers.Embedding*. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Embedding (visited on 05/22/2020).

[29] Søren M. Karst et al. "Enabling high-accuracy long-read amplicon sequences using unique molecular identifiers and Nanopore sequencing". In: *bioRxiv* (2019). DOI: 10.1101/645903. eprint: https://www.biorxiv.org/content/early/2019/05/28/645903.full.pdf. URL: https://www.biorxiv.org/content/early/2019/05/28/645903.

[30] TensorFlow. *TensorFlow*. URL: https://www.tensorflow.org/ (visited on 05/22/2020).

[31] Keras. *Keras: the Python deep learning API*. URL: https://keras.io/ (visited on 05/22/2020).

[32] Tensorflow. *Transformer model for language understanding*. URL: https://www.tensorflow.org/tutorials/text/transformer (visited on 05/22/2020).

[33] Simpson Lab. *Quickstart - how to align events to a reference genome*. URL: https://nanopolish.readthedocs.io/en/latest/quickstart_eventalign.html (visited on 06/01/2020).

[34] Oxford Nanopore Technologies. *Re-squiggle Algorithm*. URL: https://nanoporetech.github.io/tombo/resquiggle.html (visited on 06/01/2020).

[35] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[36] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (2015).

[37] Sainbayar Sukhbaatar et al. "Augmenting self-attention with persistent memory". In: *arXiv preprint arXiv:1907.01470* (2019).

[38] Sainbayar Sukhbaatar et al. "Adaptive attention span in transformers". In: *arXiv preprint arXiv:1905.07799* (2019).

[39] B. Alberts et al. *Molecular Biology of the Cell*. 6. ed. W. W. Norton & Company, 2014. ISBN: 9780815344322.

[40] CK-12. *The Polymerase Chain Reaction - Advanced*. URL: https://www.ck12.org/book/CK-12-Biology-Advanced-Concepts/section/9.3/ (visited on 06/02/2020).

[41] khanacademy. *Polymerase chain reaction (PCR)*. URL: https://www.khanacademy.org/science/biology/biotech-dna-technology/dna-sequencing-pcr-electrophoresis/a/polymerase-chain-reaction-pcr (visited on 06/02/2020).

[42] Matthew D Zeiler and Rob Fergus. "Visualizing and understanding convolutional networks". In: *European conference on computer vision*. Springer. 2014, pp. 818–833.

[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[44] Junyoung Chung et al. "Gated feedback recurrent neural networks". In: *International Conference on Machine Learning*. 2015, pp. 2067–2075.

[45] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[46] Felix Gers. "Long short-term memory in recurrent neural networks". PhD thesis. Verlag nicht ermittelbar, 2001.

[47] Mike Schuster and Kuldip K Paliwal. "Bidirectional recurrent neural networks". In: *IEEE transactions on Signal Processing* 45.11 (1997), pp. 2673–2681.

[48] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

[49] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. "Effective Approaches to Attention-based Neural Machine Translation". In: *CoRR* abs/1508.04025 (2015). arXiv: 1508.04025. URL: http://arxiv.org/abs/1508.04025.

[50] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[51] Tim Cooijmans et al. "Recurrent batch normalization". In: *arXiv preprint arXiv:1603.09025* (2016).

[52] Metrichor LTD. *Metrichor LTD*. URL: https://metrichor.com/ (visited on 05/26/2020).

[53] M. Stroiber. *Detecting methylation and more from raw nanopore signal*. URL: https://nanoporetech.com/resource-centre/marcus-stoiber-detecting-methylation-and-more-raw-nanopore-signal (visited on 06/01/2020).

[54] Samuel Kriman et al. "Quartznet: Deep automatic speech recognition with 1d time-channel separable convolutions". In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2020, pp. 6124–6128.

[55] Suyoun Kim, Takaaki Hori, and Shinji Watanabe. "Joint CTC-attention based end-to-end speech recognition using multi-task learning". In: *2017 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE. 2017, pp. 4835–4839.

[56] Yu Li et al. "DeepSimulator: a deep simulator for Nanopore sequencing". In: *Bioinformatics* 34.17 (2018), pp. 2899–2908. DOI: 10.1093/bioinformatics/bty223. eprint: /oup/backfile/content_public/journal/bioinformatics/34/17/10.1093_bioinformatics_bty223/2/bty223.pdf. URL: http://dx.doi.org/10.1093/bioinformatics/bty223.

APPENDIX

APPENDIX A
BACKGROUND

A. DNA and PCR

Deoxyribonucleic acid (DNA) is a molecule present in living cells, which stores all the genetic information of the organism. Each of the two chains which constitute it contain a series of bases which encode the information itself. The bases are Adenine (A), Cytosine (C), Thymine (T) and Guanine (G). Due to their structures, Adenine can pair with Thymine while Cytosine with Guanine. The two chains are therefore able to represent the same genetic information. During biological cell processes, the DNA strand splits into the two chains, enabling other molecules to transcribe the information, whether for replication purposes or to be transformed into proteins [39].

Polymerase Chain Reaction (PCR) is a synthetic process by which specific genes can be targeted and copied. While the mutations present in our DNA are what enables evolution and the differences between members of a species, DNA contains sequences of low variability, which rarely mutate. DNA which perfectly targets these areas can be synthesised in order to mark the start and end of a gene of interest. Enzymes are then used to transcribe the marked area a large number of times. This process is therefore called *gene amplification* [40, 41].

PCR is useful in basecalling since consensus can be obtained between reads of clones of the same gene, reducing random errors caused by the basecaller and sequencer. Our project uses PCR amplified DNA since it guarantees that the information in each read is identical to the others of the same gene. The ground truth is obtained either by more accurate sequencing techniques or by performing consensus.

B. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are neural networks commonly used for image processing. The convolution operation refers to multiplying a matrix of learnable parameters called a *kernel* or *filter* over the input, generating a *feature map*. Multiple kernels are commonly used, generating as many feature maps. The intuition is that each kernel is able to extract features and relations, which are then further combined by more convolutions or dense layers [18]. While the convolution operation is typically performed on 2 or 3 dimensions such as in the case of pictures, recent research shows that 1-dimensional convolutional neural networks are highly effective in handling time series [20].

Pooling layers are used for dimensionality reduction. A kernel is moved over the feature maps, similar to the convolution operation. The pooling layer has no trainable parameters, instead it reduces the data under the kernel by averaging or taking the largest value (average- or max-pooling).

Deep convolutional neural networks have proven to be very powerful, since they are able to extract both low and high level features from data [42, 43]. Due to the problem of vanishing and exploding gradients which will be explained in the RNN subsection, convergence in the first layers becomes difficult. Residual connections are connections that skip layers by adding a previous output to the current layer output. Paper [35] shows that adding residual connections greatly improves training, since low level layers are easier to optimise through the residual connection as gradients have a shorter path. Their architecture consists of residual blocks with three convolution layers of kernel sizes 1,3,1, and a residual connection that bypasses them.

C. Recurrent Neural Networks

Recurrent neural network (RNN) is a neural network used for machine learning tasks involving sequence modelling [44], such as basecalling. Such a network can accept multiple inputs, while it remembers the previous inputs it has encountered by propagating and altering its hidden state. The current hidden state h_t can be computed as a function of the previous hidden state h_{t-1} and the current input x_t:

$$h_t = \Phi(Wx_t + Uh_{t-1}) \tag{4}$$

where U and W are learnable weight matrices and Φ is a non-linear activation function.

Because the output at a time-step t depends on the previous time step $t-1$ in the feed-forward phase, during back propagation, the gradients at a time-step t depend on gradients at the time step $t+1$. Back propagation in the context of an RNN is therefore called *back propagation through time*. Because the computation of gradients is multiplicative, gradients have a tendency to either *explode* or *vanish*. Consequently, RNNs with many time-steps are unable to learn long term dependencies. This behaviour is known as the *exploding / vanishing gradient problem* [45].

Subsequent research has improved recurrent models with more complex cells such as the LSTM and GRU cells [17, 19]. The LSTM cell, for example, contains forget, input and output *gates* which decide what information to keep and what to disregard. LSTM propagates an additional vector - cell state c, which is not exposed to non-linear transformations. Given an input x_t, previous hidden state h_{t-1} and a previous cell state c_{t-1}, the current cell state c_t and hidden state h_t can be defined as:

$$\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\
\\
\tilde{c}_t &= tanh(W_c x_t + U_c h_{t-1}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
h_t &= o_t \odot tanh(c_t)
\end{aligned} \tag{5}$$

Where W, U and b are learnable parameters and $\odot$ is a Hadamard product. Initial states h_0 and c_0 can be randomly initialised.

Such a cell structure improves the stability of gradients [46] and allows the network to be longer. However, the gradients still have to propagate back in time across the length of the entire sequence.

Furthermore, [47] shows Bidirectional Recurrent Neural Networks (BRNNs). A BRNN consists of two RNNs, each traversing an input sequence from different directions. The outputs of these networks are usually concatenated or added together. This is useful for sequence problems where a time step t depends on both previous and future time steps. BRNNs may also consist of more complex cells, such as LSTM or GRU cells.

D. Connectionist Temporal Classification

Connectionist Temporal Classification (CTC) is a method to train sequence-to-sequence models, where the input and output sequence lengths vary and these sequences are not synchronised [12].

A model first performs a forward pass, obtaining a sequence of output probability distributions $Y = y_1 \dots y_T$ over an alphabet Σ. In terms of basecalling $\Sigma = \{A, T, C, G\}$.

Given a true label $s = s_1, s_2, \ldots, s_M$, instead of unconstrained decoding by simply selecting the base with the highest probability at every time step t, CTC finds all possible alignments of Y and s. It does so, by first constructing a $T \times M$ table, such that each row represents a label symbol s_m and each column corresponds to a time step t. A traversal of such a table from a top left cell to the bottom right cell, allowing only single horizontal or single leading diagonal transitions, results in a sequence $s' = s'_1, s'_2, \ldots, s'_T$. All possible resulting sequences $S' = \{s'^1, s'^2, \ldots, s'^N\}$, are guaranteed expansions of s (e.g. $S' = \{ATTG, AATTGG, ATTTTG, \ldots\}$ and $s = ATG$). To compute the loss, CTC uses a forward-backward algorithm. The forward variable, $\alpha(t, s_m)$, is the probability of all paths from the first time step to the time step t and symbol s_m. The backward variable $\beta(t, s_m)$ is the probability of all paths from the time step t and symbol s_m to the last time step. Forward backward algorithm is a dynamic programming algorithm, due to the recursive nature of α and β. For both we have to consider the possible horizontal or leading diagonal transitions.

$$\begin{aligned}\alpha(t, s_m) &= (\alpha(t-1, s_m) + \alpha(t-1, s_{m-1}))y_t^{s_m} \\ \beta(t, s_m) &= (\beta(t+1, s_m) + \beta(t+1, s_{m+1}))y_t^{s_m}\end{aligned} \tag{6}$$

Additionally, if a cell does not have an incoming or outgoing connections, α or β return 0. The loss at a particular time step is then calculated as:

$$\mathcal{L}_t = -\ln \sum_{m=1}^{M} \frac{\alpha(t, s_m)\beta(t, s_m)}{y_t^{s_m}} \tag{7}$$

It is then possible to compute gradients for a particular time step by computing the partial derivatives of $\mathcal{L}_t$ with respect to the network weights. These gradients can be aggregated over each batch and are then propagated to update network parameters.

For many problems, including basecalling, such a method would result in ambiguities when decoding the expanded label sequence s', as it does not account for consecutive bases of the same type in the true label s. For example, given $s' = ATTTG$, the true label could be ATG, $ATTG$ or $ATTTG$. To overcome this, the alphabet is augmented by adding a blank character $L' = L \cup \{-\}$. The true label is also augmented with blanks:

$$s := -s_1 - s_2 - \ldots s_M - \tag{8}$$

Consequently, the resulting table has size $(2M + 1) \times T$. Furthermore, the transition rules are changed, as it is possible to skip over a blank character. If $s_m \neq -$ and $s_m \neq s_{m-2}$, α can be computed as:

$$\begin{aligned}\alpha(t, s_m) = (\alpha(t-1, s_m) + \alpha(t-1, s_{m-1}) + \\ \alpha(t-1, s_{m-2}))y_t^{s_m}\end{aligned} \tag{9}$$

Similarly for β, if $s_m \neq \{-\}$ and $s_m \neq s_{m+2}$:

$$\begin{aligned}\beta(s, t) = (\beta(t+1, s_m) + \beta(t+1, s_{m+1}) + \\ \beta(t-1, s_{m+2})y_t^{s_m}\end{aligned} \tag{10}$$

If those conditions are not met, α and β are calculated according to Equation 6. This no longer results in ambiguities, because the blanks now indicate consecutive bases of the same kind. For example, $s'^1 = ATTTG$ is an expansion of $s^1 = ATG$ and $s'^2 = ATT{-}TG$ is an expansion of $s^2 = ATTG$.

E. Sequence-to-sequence models

Two recurrent models (for example two LSTM networks) can also be used to map sequences to sequences, as proposed by [48]. The first recurrent model called *Encoder* reads the source sequence representations $X = (x_1, x_2, \ldots, x_n)$ one time step at a time and outputs their vector representations Z. The other recurrent model is called the *Decoder* and learns to extract the target sequence $Y = (y_1, y_2, \ldots, y_m)$ from Z.

F. Attention

Attention is a mechanism initially designed to be used together with sequence-to-sequence models, which helps selectively focus on different parts of the source sequence when determining the target sequence. This approach has shown improvements in combination with encoder-decoder based recurrent neural network models [49], where it is used to give a weight to each time step of an encoder when determining the next target sequence output.

Recently, the *Transformer* [15] showed how a model can determine dependencies between the source sequence and the target sequence while relying solely on attention, without the need of recurrence.

G. Positional encoding

The main effective difference between transformers and recurrent networks is the fact that transformers do not need to iterate through the data one time step at a time. Instead, a transformer needs a different method to distinguish time separation. *Positional encoding* is a way to embed this information into the data. Sine and cosine functions of different frequencies are commonly used:

$$\begin{aligned}PE_{(pos,2i)} &= sin(pos/10000^{2i/d_{model}}) \\ PE_{(pos,2i+1)} &= cos(pos/10000^{2i/d_{model}})\end{aligned} \tag{11}$$

where d_{model} is a parameter of the network corresponding to the depth at which the input is embedded and passed between layers.

The dimensions of the positional encoding therefore correspond to sinusoids with wavelengths that form a geometric progression from 2π to $1000 \cdot 2\pi$. This encoding matrix is added to the input matrix, enabling the model to learn by using relative positions. In other words, close encodings are more similar to one another than far encodings, which the model can use as an indication of temporal closeness [15].

The Transformer first embeds every element of the source and target sequence x_i and y_j to a fixed-sized vector representation and adds the corresponding positional encoding vector.

H. Teacher forcing

Teacher forcing is used to speed up the training. In a standard training loop we would perform inference from the start token to the end token and then apply the loss function. This approach is slow to train, since errors at the beginning of the prediction affect the rest of the inference. Teacher forcing allows the model to predict each step with all the bases predicted beforehand being correct. In practice, this is enforced using a mask. The mask is a matrix of very large negative numbers which, when applied during dot product attention, prohibits the model from paying attention to the masked steps. Training can be therefore done directly on the entire prediction batch at once, where each prediction step has its future masked. This is called *masked multi-head attention*.

I. Batch Normalisation

Paper [50] describes how a distribution of each neural network layer inputs change, with changing parameters of the previous layers. This is known as the *internal covariate shift*. This can be mitigated by normalising the input to each layer of the network according to the mean and standard deviation of the current batch. Batch normalisation of a training example x_i of a batch $\mathcal{B} = \{x_1, x_2, \ldots, m\}$ is defined as:

$$y_i = BN_{\gamma,\beta}(x_i) = \beta + \gamma(\frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma^2_\mathcal{B} + \epsilon}}) \tag{12}$$

Where γ and β are learnable parameters and ϵ is a regularisation parameter [50].

Batch normalisation can be also applied between LSTM cells, where it is leveraged in both input to hidden and hidden to hidden transformations [51].We can alter the definition of an LSTM cell and define a batch normalised LSTM cell as [50]:

$$\begin{aligned}
f_t &= \sigma(BN_{\gamma_f,\beta_f}(W_f x_t) + BN_{\gamma_{f'},\beta_{f'}}(U_f h_{t-1})) \\
i_t &= \sigma(BN_{\gamma_i,\beta_i}(W_i x_t) + BN_{\gamma_{i'},\beta_{i'}}(U_i h_{t-1})) \\
o_t &= \sigma(BN_{\gamma_o,\beta_o}(W_o x_t) + BN_{\gamma_{o'},\beta_{o'}}(U_o h_{t-1})) \\
\\
\tilde{c}_t &= tanh(BN_{\gamma_{\tilde{c}},\beta_{\tilde{c}}}(W_c x_t) + BN_{\gamma_{\tilde{c}'},\beta_{\tilde{c}'}}(U_c h_{t-1})) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
h_t &= o_t \odot tanh(BN_{\gamma_c,\beta_c}(c_t))
\end{aligned} \tag{13}$$

The input terms Wx_t and the recurrent terms Uh_{t-1} are normalised separately, so that the model is able to distinguish parameters γ and γ'. Parameters β and β' can be set to 0 to avoid redundancy with b [50].

APPENDIX B
ALTERNATIVE ARCHITECTURES

A. Historic approaches

Various approaches have been used to perform basecalling. The first basecallers identified *events* in the raw electric signal. The concept of an event stems from the nature of sequencing. It represents the *k-mer* (k consecutive bases) present in the sequencing device at the time of measurement. The k varies depending on the device.

Basecallers such as *Nanocall* [11] and older versions of *Metrichor* [52] attempt to determine these models using Hidden Markov Models (HMMs) [10], where each hidden state represent the most probable k-mer. *DeepNano* [7] shifts from HMMs and instead uses Recurrent Neural Networks to determine those events.

Later, *Scrappy* showed the potential of transitioning from event-based basecallers to basecalling the raw signal directly [6], where it reached higher accuracies. Basecalling the raw signal has become the standard for following basecallers.

B. Chiron

Since *Gravlax* is inspired by *Chiron* they are architecturally very similar. Chiron also uses 5 residual blocks, each containing 3 CNN layers with filter sizes 1,3,1 respectively and each layer having 256 filters.

The residual blocks are followed by 3 bidirectional LSTMs. The outputs of a bidirectional LSTM layer are added. The outputs of the final bidirectional LSTM are concatenated and fed to a Fully connected layer with a Softmax activation, producing a sequence of output probability distributions $y = y_1, \ldots, y_T$ over the alphabet $\sigma = \{A, T, C, G, -\}$, where $-$ represents the blank symbol. Chiron also adds batch normalisation between residual blocks and between LSTM cells according to Appendix A. It is trained with CTC loss between the output sequence y and label sequence $s = s_1, \ldots, s_M$.

Chiron infers the final sequence using either greedy decoding or a beam search decoder. The greedy decoder first selects the bases with the highest probabilities and then removes repetitions and blanks. Beam search decoder is described in Appendix C Subsection B.

C. Guppy

Guppy is an ONT basecaller, which currently represents 3 models - *RGRGR*, *RGRGR big net* and *RGRGR flip flop*. The first model applies a single convolution layer on the raw signal with filter size 11 and stride 5, such that there are 5 raw signal values per prediction, 96 filters and an ELU activation shown in Eq. 14. It then uses 5 RNN layers with GRU cells and $tanh$ activations. Contrary to Chiron, it does not use a bidirectional layer, but rather every second layer is a reversed GRU, hence the name *RGRGR*. The final layer contains a Softmax activation with 1025 possible predictions - 1024 k-mers and additional 'stay' state. The *RGRGR big net* has a similar structure, but the convolution layer filter size is increased to 19 and the layer contains 192 filters.

$$ELU(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases} \tag{14}$$

where α is picked, commonly between 0.1 and 0.3.

The *Flip-flop* model uses CTC to predict a base at each time step. However, instead of predicting over the typically used alphabet with a blank symbol $\Sigma = \{A, T, C, G, -\}$, it flips between an upper-case and a lower-case base. The modified alphabet $\Sigma' = \{A, T, C, G, a, t, c, g\}$ can then be used to decode consecutive bases [53]:

$$A, A, A, a, a, a, a, A, A \longrightarrow A, A, A \tag{15}$$

This improves accuracy, possibly because flip flop causes predictions to be evenly distributed over Σ', while in the typical CTC decoder with alphabet Σ, the blank symbol is used much more frequently than the actual bases. The Flip-flop model has convolution stride 2, which reduces the output length by half, and uses 256 filters.

D. Bonito

Bonito [14] is another ONT basecaller, which relies solely on convolutional layers. It is designed according to the Quartznet architecture [54].

Bonito first applies a convolution with kernel size $k_{c_1} = 33$, stride $s_{c_1} = 3$ and number of filters $f_{c_1} = 256$ on the input signal. This is followed by 5 convolution blocks. A convolution block consists of 5 convolution layers of the same structure, with a residual connection. The first 2 blocks contain 256 filters, while for the last 3 blocks the number of filters is increased to 512. The kernel size of blocks is first increased by 6 followed by increases of 12 after every block - $k_{b_1} = 33, k_{b_2} = 51, \ldots, k_{b_5} = 75$ with stride 1 for every block. The convolution blocks are followed by two convolution layers. The first layer has the following parameters: $f_{c_2} = 512$, $k_{c_2} = 87$, $s_{c_2} = 1$. The second layer increases the number of filters further - $f_{c_3} = 1024$, and reduces the kernel size $k_{c_3} = 1$. The stride remains 1 for both layers. Finally, there is a layer with a softmax activation function, producing an output probability distribution for every signal measurement over the alphabet $\Sigma = \{A, T, C, G, -\}$. Similar to previously mentioned basecallers, Bonito uses the CTC loss for training.

E. SACall

SACall [16] is architecturally the closest to our proposed model, *FishNChips*, because it utilises certain parts of the transformer [15].

SACall splits the signal into several windows of length 2048, which are significantly longer than windows basecalled by *FishNChips* or *Gravlax*. Furthermore, SACall shifts windows without overlap.

The signal window is then passed through a convolution module consisting of two convolution layers, tasked with down-sampling the signal and extracting features. Both layers have the same kernel size of 3 and stride of 2. The first layer has $d_{model}/2$ filters, while the second increases the number of filters to d_{model}. This serves as a replacement for embedding, equivalent to the one used in *FishNChips*. Each convolution layer is followed by batch normalisation as described in Appendix C subsection I.

The convolution layers are followed by positional encoding and the transformer. However, SACall only utilises the encoder of the transformer, omitting the decoder entirely. The implementation of the encoder is equivalent to *FishNChips*, employing both Multi Head Attention and the Point-wise Feed Forward Network. The decoder is replaced by a fully connected layer with a softmax activation, producing a probability distribution over the alphabet $\Sigma = \{A, C, T, G, -\}$. Similarly to Guppy, Bonito and Chiron, SACall uses a CTC loss to train the network and a CTC decoder to obtain the final sequence of bases during inference.

F. Training state of the art basecallers

As a way to obtain a better comparison, we attempted to train both Chiron and Bonito on our data, as they are state of the art open source basecallers. We had little success with this and, due to time constraints, didn't investigate further. Chiron never trained and Bonito, while showing 75% accuracy during validation, only achieved 45.09% using our testing method. We blame this on technical issues, not on our dataset or the models.

G. Unsuccessful attempts

We attempted to implement the SACall architecture as another point of comparison. Modifying *FishNChips*, we only kept the encoder to which we added a feed-forward neural network with 5 neurons at the end, corresponding to the four bases ATCG and the blank symbol. Finally, the training loop used the CTC algorithm as a loss function. The input and labels were therefore the same.

Using $d_{model} = 512$, 4 blocks and 8 heads we obtained an accuracy of 78%. We abandoned work on improving this architecture since it was so low compared to our other models.

We also attempted a hybrid model that adds the CTC loss to the encoder of *FishNChips*, based on [55]. The purpose of this is to increase the robustness of the model by improving the intermediate representation Z. We therefore want to minimise the following:

$$\mathcal{L} = \alpha\mathcal{L}_{CTC} + (1 - \alpha)\mathcal{L}_{Attention}$$

where $\mathcal{L}_{CTC}$ is the CTC loss, $\mathcal{L}_{Attention}$ is the categorical cross-entropy loss used for training the transformer, and α is a tunable parameter which weights the importance of $\mathcal{L}_{CTC}$ in comparison with $\mathcal{L}_{Attention}$.

We add CTC to *FishNChips* with $d_{model} = 512$, 4 blocks and 8 heads. Using $\alpha = 0.1$ yielded 82% accuracy, while $\alpha = 0.4$ did not train at all. We abandoned this model since it lowered the accuracy of *FishNChips*.

H. Synthesising data

We investigated whether or not our data is biased towards certain k-mers. The analysis was performed on the references of the bacterial DNA we use. Since the DNA that passes through the pore can be read in both directions and can also be the reverse strand, we reverse a copy of the original strand, compute the opposite strand and reverse a copy of it as well. Every instance of a k-mer is counted. The list is sorted and plotted such that it shows the popularity of certain k-mers in the data. Figure 4 shows this distribution. Additionally, the x axis shows what percentage of the total number of possible k-mers, 4^{kmer_count}, is represented. While 5- and 7-mers are well represented in our data, we only get under 0.01% representation with 15-mers. We expect 25-mers in our window size, which constitutes 10^{-8}% representation for our data.

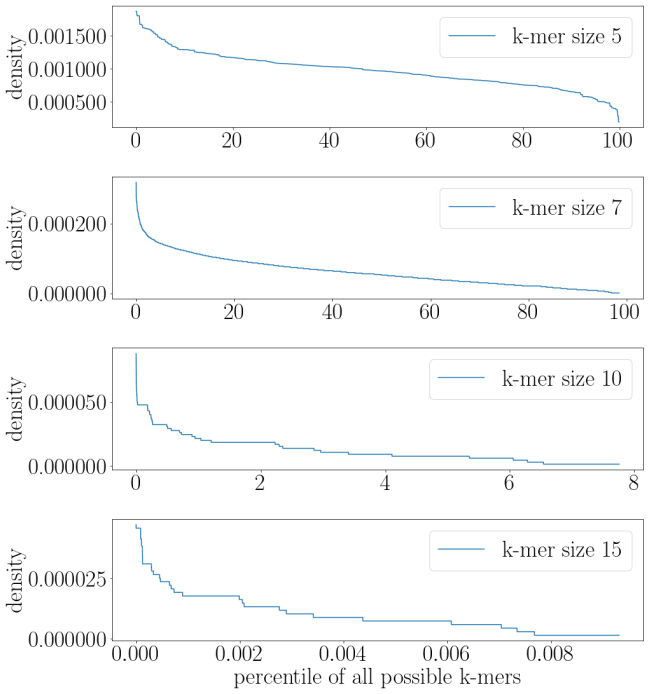

Fig. 4. Distribution of k-mers

We tried using *DeepSimulator* [56], a model that generates simulated nanopore data for given DNA. We generate mock DNA consisting of random bases. DeepSimulator splits the random DNA string into subsequences of 4200 mean length and generates the corresponding reads. Just like for normal data, we use Taiyaki to match each base to a location in the signal. We attempt to train the basecaller using this mock data in hopes that it will improve the accuracy on DNA it has not seen. This, however, is not the case. We attempt both to train purely on the mock DNA and also to dope our training set by using 80% real DNA and 20% mock DNA. Table VI shows the results.

The model which was only trained on mock DNA predicted reads of low accuracy, 28% of which were not able to be mapped to the reference and therefore count as accuracy 0. If we disregard the reads that did not get mapped, the mean accuracy would be 71.55%.

Parameters	Accuracy on trained bacteria	Accuracy on untrained bacteria
Only trained on random DNA		51.51%
80% real DNA, 20% mock DNA	90.34%	88.09%

TABLE VI
MODEL EVALUATION ON DIFFERENT APPROACHES OF USING RANDOMLY GENERATED DNA

APPENDIX C
IMPLEMENTATION DETAILS

A. Transformer optimisation

We were able to considerably speed up inference time by caching the output of the encoder. Furthermore, we can batch multiple windows together. Additional logic is needed to check whether all items have the end token, in which case the inference stops. This requires more computation than performing inference one by one, but ends up being much faster due to the ability of modern GPUs to parallelise.

B. Beam search

Beam search decoding is an algorithm which uses breadth-first search to find the most probable decoding using the probability distributions generated by a CTC model. Since it is often too computationally expensive to consider all paths, commonly only a number of best paths is considered. The maximum number of best paths being considered at any time is called *beam width*. In our paper we use greedy decoding, which picks the most probable element at each time step. Since it only considers one path at a time, it is equivalent to a beam search with beam width 1.

We have attempted to use beam search decoding using the *ctc_decode* function provided by Keras [31]. However, for any $beam\_width > 1$, the results were substantially worse than using greedy decoding. We therefore ran all our experiments using $beam\_width = 1$.

C. Learning rate

We found transformers to be particularly sensitive to learning rate, even after the inclusion of warm-up steps. While the warm-up formula presented in Appendix A was effective for $d_{model} = 250$, the model never converged for $d_{model} > 500$. In order for it to converge, we scaled the learning rate down by multiplying d_{model} in the formula with a scalar η, as shown in Eq 16. We found that a value of $\eta > 10$ helps models with $d_{model} > 1000$ converge by lowering the learning rate.

$$lrate = (\eta \cdot d_{model})^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5}) \quad (16)$$

We noticed our models either never converged or converged very quickly. Figure 5 shows that the model reaches close to maximum accuracy in 25 epochs, after which it does not continue to improve. The learning rate seems to have a very small effect on the final accuracy or on training speed.

Even though the model with the lower learning rate trained longer, its results were worse by around 1%, both on bacteria it was trained on and bacteria it was not trained on.

D. Overlapping and concatenated windows

Assembling the overlapping predicted windows has provided key insights and reasons to pivot towards transformers. We observed that assembling windows allowed models to ignore non-systematic wrong predictions, which improved their overall robustness. Compared to simply concatenating a sequence of non-overlapping windows, assembling increased the accuracy of already well performing models. However, we also observed that when a model struggled to predict the correct sequence of bases and hence appropriately align the windows, assembling impacted the accuracy negatively. We found that the threshold is approximately the accuracy of 90% on concatenated windows.

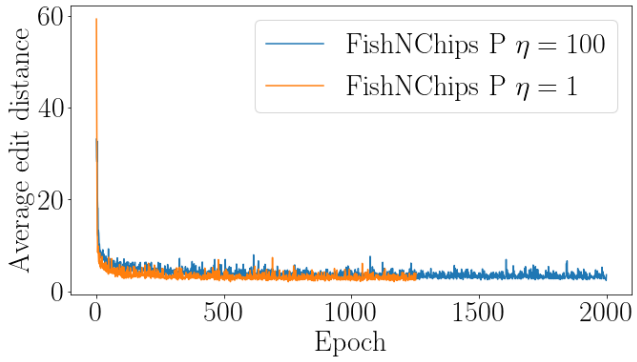

Fig. 5. Convergence of *FishNChips* models with different learning rates

E. End-to-end basecaller

As described in section VII-E, we built our own basecaller which handles the entire read in order to be able to fairly compete with the other basecallers. Its structure, however, is as simple as possible. We take the raw signal, normalise it and split it into corresponding windows. Improvements can be obtained by performing further preprocessing before basecalling.

A reason for the accuracy drop is possibly attributed to the leading and trailing empty signals. Since we never train on empty signal, the model only gets to learn to not predict anything in the rare cases where DNA was stuck in the nanopore, leading to no changes in a long time within a read. We are prohibited from training on the entire read due to the barcodes present in our data, which we do not know and therefore cannot label. In our experience, the model learned to predict nothing during the blank signal even when it has not been deliberately trained on it. However, incorrect predictions still seldom happen.