Electronics and IT
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Palamut - An Expansion of the Bonito basecaller using language models

Theme:
MI-Bio

Project Period:
Spring Semester 2020

Project Group:
MI104F20

Participant(s):
Andreas Christian Meyer Larsen
Christian Aae Knudsen
Magnus Nørhave Hansen

Supervisor(s):
Katja Hose
Thomas Dyhre Nielsen

Copies: 1

Page Numbers: 17

Date of Completion:
June 11, 2020

Abstract:

In this paper we discuss methods used in modern basecallers and the end-to-end ASR architecture adopted by the Bonito basecaller to increase accuracy. We investigate the prospect of increasing accuracy by applying common ASR approaches to basecalling.
We expand the architecture of the Bonito nanopore basecaller, by introducing a decoder algorithm, allowing for the use of language model probabilities, to increase accuracy of basecalls. We train and compare n-gram and RNN character-level language models.
Our results show that while an introduction of language models gives a slight increase in consensus accuracy of basecalls, our current language models decrease read accuracy by an equal margin. We finally conclude that the decrease in accuracy is caused by poorly optimized hyperparameters of our decoder, and present potential solutions to the problem.

ACKNOWLEDGEMENTS

We would like to thank our supervisors Thomas Dyhre Nielsen and Katja Hose for their continuous support and feedback. We would also like to thank the people at the Center for Microbial Communities at Aalborg University for spending their time sparring with us and further our understanding of the DNA domain, namely Mads Albertsen, Rasmus Kirkegaard and Mantas Sereika. Finally we would like to thank Chris Seymour at Oxford Nanopore Technologies for introducing us to the idea of applying language models in basecallers.

Palamut - An Expansion of the Bonito basecaller using language models

Andreas Christian Meyer Larsen[1], Christian Aae Knudsen[1] and Magnus Nørhave Hansen[1]

***Abstract*— In this paper we discuss methods used in modern basecallers and the end-to-end ASR architecture adopted by the Bonito basecaller to increase accuracy. We investigate the prospect of increasing accuracy by applying common ASR approaches to basecalling. We expand the architecture of the Bonito nanopore basecaller, by introducing a decoder algorithm, allowing for the use of language model probabilities, to increase accuracy of basecalls. We train and compare n-gram and RNN character-level language models. Our results show that while an introduction of language models gives a slight increase in consensus accuracy of basecalls, our current language models decrease read accuracy by an equal margin. We finally conclude that the decrease in accuracy is caused by poorly optimized hyperparameters of our decoder, and present potential solutions to the problem.**

1. Introduction

Since the introduction of Single Molecule Sequencing (SMS) approaches, the accuracy of SMS, or third-generation sequencers and basecallers have increased. During the period from 2019 to early 2020, state-of-the-art basecallers developed by Oxford Nanopore Technologies (ONT) [1] have increased from a median per-base accuracy of around 88% [2] to a median accuracy of 96.5% [3]. This increase is in part due to new algorithms being developed by ONT, such as the Flip-flip algorithm [4] used in basecallers such as Flappie [5] and Guppy [6], or a newer convolutional approach [3] used in the Bonito [7] basecaller. [8] covers our previous research into the open-source Taiyaki basecaller also making use of the Flip-flop algorithm. While the Bonito basecaller is currently producing promising results [3], the basecaller has only been in development since late 2019. This leaves an opening for potential improvements to the basecaller. The Bonito basecaller has adapted an approach from Automatic Speech Recognition (ASR) and makes use of a deep convolutional network for basecalling. The motivation for this project is to increase the accuracy of the bonito basecaller by introducing language models in the decoding step. This approach has seen great results in ASR tasks [9] [10], and based on previous research on the topic of representing DNA through a language model [11] [12] should be applicable in the DNA-domain as well. Both domains have an analog signal that is converted to a digital signal. The signal represents a sequence of characters from a fixed alphabet, and the problem is to convert the digital signal into the correct sequence of characters. Both Recurrent Neural Network (RNN) and Convolutional Neural Network (CNN) models are used in state-of-the-art basecallers. We will present a modification to the Bonito architecture, and a method for modelling common DNA patterns and inferring new sequences. To distinguish between the original Bonito architecture and our modified architecture we will refer to our architecture as Palamut.

[1]Students at Aalborg university

2. Related Work

The bonito basecaller is based on QuartzNet [9]. QuartzNet is an end-to-end neural acoustic model used for ASR. The goal of the QuartzNet paper was to reduce the number of parameters normally found in CNNs used for ASR tasks by introducing 1D time-channel separable convolutions, significantly reducing the size of their models compared to similar CNNs not using this technique. The QuartzNet architecture and decoding method is further described in Appendix I. While the Bonito basecaller documentation is sparse and only present on the ONT Github page [7], by reading their implementation in combination with the QuartzNet paper [9] information can be extrapolated regarding the workings of the basecaller itself. The Bonito basecaller is composed of two distinct parts; the model adopted from QuartzNet, used for translating sequences of nanopore reads to probability distributions representing the bases of DNA, and the decoder algorithm, used to convert these probabilities into a final DNA sequence of bases.

[9] presents the option of including language models in the decoding algorithm allowing for an increase in accuracy. This approach is not explored by the current version of the Bonito basecaller, which only decodes using the probability scores provided by the model. The QuartzNet decoder operates using word-level language models, however [10] explores the option of using character-level language models for a similar problem. It is worth noting the shift in domain, from natural language processing in QuartzNet, to Bonitos focus on DNA. Bonito however has been able to implement the QuartzNet model for use in basecalling with great results [13].

[12] shows that DNA sequences can be segmented into 'words' and also show that the average lengths of these "words" in DNA is between 12-15 characters long. This shows that DNA could contain a natural 'language', and therefore a language model could prove useful for increasing the accuracy of basecallers. While attempts have been made to model DNA as words, we believe that character-level modelling could potentially help increase accuracy as well. The decoding algorithm of Bonito could be modified to accept language model probabilities [10]. [12] presents potential uses for DNA segmentation such as a DNA search engine or proofreading DNA to identify mutated genomes. While these applications have a lot of potential they do not present an immediate solution for increasing

basecalling accuracy. Based on our previous documentation of the Taiyaki basecaller [8] it is apparent that there are two major sources of errors when basecalling. One being random errors produced by the nanopore either halting, skipping bases or electrical interference. These errors are commonly removed by calculating a consensus sequence from several basecalls. The other being systematic errors, which can occur during sequencing but can also be caused by the basecaller misinterpreting reads consistently. These errors are not removed in a consensus sequence. While ONT is currently working on producing better training sets for their basecallers [14], modeling DNA on a character-level could also help remove systematic errors, by enforcing common patterns when a decoder is presented with two or more possible bases with similar probabilities. Word-level language models could potentially also increase accuracy. However the nature of DNA being a continuous sequence of bases, not containing spaces, lends itself more to a character-level approach.

In this paper we experiment using different character-level language models for DNA sequences, applied to the existing bonito architecture.

In Appendix VI we have included sections from our previous report about the Taiyaki basecaller. We refer to this when relevant throughout the paper.

3. Model Architecture

The data used in this project is nanopore sequence data. A sequence of reads produced by running a strand of DNA through a nanopore and measuring the current. Due to the width of nanopores, each measurement is a representation of five DNA bases present in the nanopore, a so-called 5-mer. Each DNA base is on average present in 9 measurement due to the frequency of measurements and the speed in which a DNA strand is pulled through the nanopore. Several problems can however occur when sequencing DNA, this includes a base being pulled through the nanopore, getting stuck or passing through too fast. Electric interference can also result in a measurement not representing bases accurately. We cover the process of sequencing DNA and errors further in our previous work documenting the Taiyaki basecaller [8]. Appendix VI-C also contains a brief description of nanopore basecalling from our previous work documenting basecallers.

In the Bonito basecaller and subsequently Palamut, a sequence of nanopore reads are fed to the network producing a list of probabilities. These probabilities are then translated to a DNA sequence using the decoder. As mentioned previously, nanopore data can contain errors. While some errors such as a short random spike in current can be suppressed by the convolutions used in the QuartzNet model, others may persist resulting in probability distributions not representing the actual bases of the sequence. Palamut attempts to resolve some of these errors by introducing a language model based on actual DNA references to enforce a bias towards common patterns.

While DNA does not contain easily distinguishable sentences as seen in natural languages, patterns of bases are still shared between species. Using the software suite blastn [15] to lookup one Escherichia reference shows that the bacterial part shares 97% with one Salmonella reference also present in our dataset. These common pattern can be identified by modelling DNA using language models. By modelling patterns in DNA we can potentially decrease errors in basecalling by supplying the decoding algorithm with additional language model probabilities to enforce common behaviour when probabilities are evenly distributed between bases.

The Palamut architecture revolves around replacing the decoder algorithm of Bonito with an algorithm allowing for the use of language models when basecalling sequenced DNA. The output of the QuartzNet model used in Bonito is the Connectionist Temporal Classification (CTC)-probability distributions for a DNA sequence, which is decoded to produce a basecall. CTC is described in subsection VI-B. Based on related work we have implemented two different types of language models for use in the decoder step. The changes to the existing Bonito architecture is illustrated in Figure 1. In both the original Bonito architecture and our updated Palamut architecture, the QuartzNet model is used for translating nanopore sequence data into probability distributions used for CTC-decoding. Palamut implements the language model as a shallow fusion. This method, uses a pre-trained language model at inference time only. This type of integration can be modeled with the following optimization problem:

$$\mathbf{y}^* = \underset{\mathbf{y}}{\operatorname{argmax}} P(\mathbf{y}|\mathbf{x}) \cdot P_{lm}(\mathbf{y})^{\alpha} \tag{1}$$

where $\alpha > 0$, $\mathbf{x}$ is the input sequence, $\mathbf{y}$ is a possible predicted output sequences, and $\mathbf{y}^*$ is the best fitting output sequence. α is a parameter which controls the influence from the language model probabilities.

A. Connectionist Temporal Classification

This description of CTC is derived from our previous report [8] and [16]. QuartzNet models are trained with a CTC loss function. CTC is used to derive an objective function, and subsequent loss function, in sequence-to-sequence models, where the input is longer than the output, and where the alignment between the input and output is unknown [17].

For an input sequence X of length T the output of the QuartzNet model is a set of probabilities $p(c|x_t), t = 1, ...T$ where c is a character. CTC assumes these outputs are conditionally independent. CTC further assumes the ground truth DNA sequence W has length τ where $\tau \leq T$, therefore a function is required to construct possible shorter output sequences. The *collapsing function* is used for this purpose. It introduces a *blank* character, denoted by "-". W consists of characters in Σ, where $\Sigma = \{A, C, T, G\}$. The model however needs to reason over the extended alphabet which is denoted as $\Sigma' = \Sigma \cup -$.

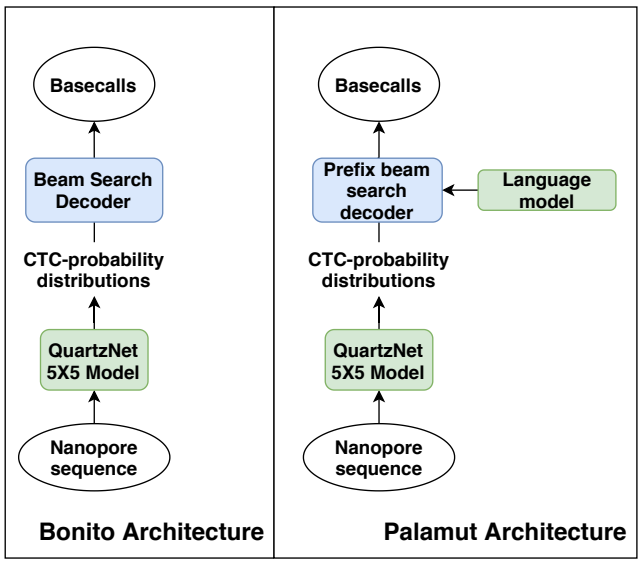

Fig. 1: Original Bonito architecture and our attempted updates

The *collapsing function* works by collapsing the characters generated by a sequence π, by removing all *blank* characters and repeated characters. The *collapsing function* is denoted by $cf(\pi)$, where π is a sequence that can contain "-". For example the set of all sequences of length $T = 3$ that generates the sequence AC after being collapsed are: $\{AAC, ACC, -AC, A-C, AC-\}$. $cf^{-1}(\cdot, T)$ is the inverse of the collapsing function for length T, i.e. it generates all uncollapsed sequences of length T that collapses sequence. T is required in our definition of the inverse collapsing function because "-" and repeated characters otherwise can occur an infinite number of times, resulting in an infinite number of uncollapsed sequences.

The CTC loss function $L_{CTC}(X, W)$ is the probability of the correct sequence W given X. This is the probability of all sequences of length T over the Σ' which when collapsed generates W;

$$P(W|X) = \sum_{\pi \in cf^{-1}(W,T)} P(\pi|X) \tag{2}$$

The probability of each sequence π is can be calculated as product:

$$P(\pi|X) = \prod_{t=1}^{T} P(\pi_t|x_t) \tag{3}$$

Notice that each element of the product in Equation 3 are the probabilities we expect the model to output, completing our description of the CTC objective function.

Applying the objective function by calculating the probability of each π can however take a long time, as the number of possible π grow by $|\Sigma'|^T$. Using a dynamic programming algorithm, called the forward-backward algorithm, it is possible to efficiently compute the loss function as well as gradients. In Appendix VI-B.1 we describe this algorithm.

B. Decoding

CTC decoding is how the QuartzNet model is used during inference to estimate a character sequence W^* given an input sequence X. As established in subsection VI-B the QuartzNet model outputs a set of probabilities $p(c|x_t), t = 1, ...T$ when given an input sequence X of length T. The simplest decoder simply chooses the character with highest probability at each timestep t, and applies the *collapsing function*:

$$W^* = cf(\underset{c}{\operatorname{argmax}} \prod_{t=1}^{T} P(c|x_t)) \tag{4}$$

This is however a greedy approximation and ideally we would compute the CTC objective function for every possible transcription W^* of length $\leq T$ and use the W^* assigned the highest score. This is however not feasible given the number of unique possible W^*. One CTC decoder which is better than the greedy approach is CTC beam search. Where the greedy approach only considers one sequence π, CTC beam search considers $k \cdot T$ sequences where k is referred as the beam size or beam width Another property of CTC beam search is that a language model can easily be integrated as proposed in [16, 18]. We refer to CTC beam search with a language model as *prefix beam search* in this paper. In algorithm 1 *prefix beam search* is defined.

The input consists of the CTC probabilities $P_{ctc}(c|x_t)$ and a character-level language model $P_{lm}(c|s)$.

For each timestep $t = 1, ..., T$ the algorithm iteratively computes the k most probable sequences s at each timestep t. The probability of a sequence s is split into two probabilities; $P_b(s|x_{1:t})$ and $P_{nb}(s|x_{1:t})$ referred to as the *blank probability* and *non-blank probability*. The *blank probability* is a probability of sequences π_b that when collapsed generate sequence s and **end with "-"**. The *non-blank probability* is a probability of sequences π_{nb} that when collapsed generate sequence s and **end with any** $c \in \Sigma$, i.e. any *non-blank* character. The sum of the two probabilities is the *total probability* of s; $P_{tot}(s|x_{1:t}) = P_b(s|x_{1:t}) + P_{nb}(s|x_{1:t})$.

For each timestep the current k most probable sequences s are considered. Each sequence s is extended by each character in Σ except for the last character of s denoted s_{-1}. Since s is a collapsed sequence, repeated characters and "-" are not added to s, this way the *collapsing function* is applied implicitly. The probabilities of extending by "-" and repeating characters are however maintained, for potential use in the next timestep. The probabilities of extending by other characters are likewise calculated. Notice that the language model is applied when extending by *non-blank* characters, and that hyperparameter α is used to control the influence of the language model.

The reason for the separation of the total probability is to allow the algorithm to handle the extension of π_b and π_{nb} separately. An example of this is when a sequence s is extended by the last character s_{-1} in line 12. This equation will result in zero when the last term $P_b(s|x_{1:t-1})$ is zero. $P_b(s|x_{1:t-1}) > 0$ is only true if the current s was extended

Algorithm 1: Prefix Beam Search: The pseudo code is derived from [16]

Input: CTC probabilities $P_{ctc}(c|x_t)$, character-level language model $P_{lm}(c|s)$
Parameters: language model weight α, insertion bonus β, beam width k
Initialize: $B_{prev} \leftarrow \{\emptyset\}$, $P_b(\emptyset|x_1:0) \leftarrow 1$, $P_{nb}(\emptyset|x_{1:0}) \leftarrow 0$

1 **for** $t = 1, ..., T$ **do**
2 $B_{next} \leftarrow \{\}$;
3 **for** $s \in B_{prev}$ **do**
4 $P_b(s|x_{1:t}) \leftarrow P_{ctc}(-|x_t)P_{tot}(s|x_{1:t-1})$; ▷ Handle blanks
5 $P_{nb}(s|x_{1:t}) \leftarrow P_{ctc}(c|x_t)P_{nb}(s|x_{1:t-1})$; ▷ Handle repeat character collapsing
6 Add s to B_{next};
7 **for** $c \in \Sigma$ **do**
8 $s^+ \leftarrow$ concatenate s and c;
9 **if** $c \neq s_{-1}$ **then** ▷ s_{-1} is the last character in s
10 $P_{nb}(s^+|x_{1:t}) \leftarrow P_{ctc}(c|x_t)P_{lm}(c|s)^\alpha P_{tot}(s|x_{1:t-1})$;
11 **else**
12 $P_{nb}(s^+|x_{1:t}) \leftarrow P_{ctc}(c|x_t)P_{lm}(c|s)^\alpha P_b(s|x_{1:t-1})$; ▷ Repeat characters that have "-" between
13 **end**
14 Add s^+ to B_{next};
15 **end**
16 **end**
17 $B_{prev} \leftarrow k$ most probable s by $P_{tot}(s|x_{1:t})|s|^\beta$ in B_{next};
18 **end**
19 **return** $\text{argmax}_{s \in B_{prev}} P_{tot}(s|x_{1:T})|s|^\beta$;

by "-" in the last timestep. Note that in line 4 and in the initialization are the only places the *blank probability* is set otherwise it is assumed to be zero.

After each iteration the candidate sequences in B_{next} are scored and sorted by $P_{tot}(s|x_{1:t})|s|^\beta$ in 17, where hyperparameter and β is a compensation term.

Without the compensation term extending by *blank* will be more favored by the scoring, because extending by *non-blank* requires the application of the language model which lowers score. The compensation term, or insertion bonus $|b|^\beta$, compensates by contributing higher score to longer candidate sequences in B_{next}.

The pseudocode of algorithm 1 can be modified to not include a language model, i.e. normal CTC beam search, by removing the terms $P_{lm}(c|s)^\alpha$ and $|s|^\beta$.

Integrating a language model can alleviate the poor assumption of CTC models, which is that the outputs are conditional independent. The language model can directly model dependencies from DNA sequences.

C. Language Model

A language model is a probability distribution over each entry in some type of language. This type of model is useful for prediction-type problems, where context in the text can be utilized to infer the correct prediction. We have compared two classes of language models, a statistical- and a neural model. Statistical models have a fixed amount of context, an example of such a model would be n-gram language models. Neural models make use of neural networks for the language model. These types of models can use a inputs of varied sizes.

A character-level language model can be used to give context when building sequences from characters. A sequence of characters more commonly seen in the training set will out-weight unknown or less occurring sequences. If a character-level language model was trained using a programming tutorial text, the character sequence *hello world* would most likely have been seen more often than *hello worlx*. When attempting to identify the next character in the sequence: *hello worl*, such a language model would produce a higher probability for the addition of the character *d* as opposed to the character *x*.

One way to evaluate a language model is with perplexity. It is a measure for how well a language model can predict a testset, provided that the testset is completely unseen for the language model. The intuition behind perplexity is, that it can be used as an early indication of a language model and how much it struggles with unseen data. A language model for DNA predicting each of the four bases with equal probability will result in a perplexity of 4. With increased accuracy of predictions the perplexity of the model decreased. A language model with a perplexity of 1 has predicted the testset at 100% accuracy. Equations and a more thorough description of perplexity can be found in Appendix IV.

D. n-gram Language Model

n-gram models use a history of $n-1$ previous characters to determine the next characters $P(c|hi)$ for some predicted character c given a history *hi*. The probability for a character c_i in a sequence can be calculated using the previous character, as shown in Equation 5.

$$P(c_1, \ldots, c_n) = \prod_{i=1}^{n} P(c_i|c_{i-1}) \tag{5}$$

However, to avoid underflow the probabilities should be calculated in log space, changing the equation to summarizing over the probabilities instead, as shown in Equation 6

$$log(P(c_1, \ldots, c_n)) = \sum_{i=1}^{n} log(P(c_i|c_{i-1}) \tag{6}$$

The n-gram language models in Palamut are implemented on a character level representing n previous bases in a DNA sequence. [12] shows that the perplexity of DNA n-gram models reduce with the increase of n-gram size until a certain point. We base our n-gram-size on these findings.

This type of language model does have some short-comings. Character-level prediction models needs a long history-window, to accurately predict the next character [10]. For the n-gram model, the history window is $n-1$ long, meaning that the language model captures the context of the n last characters. The bigger n gets, the more context the language model can use. However as the n increases, the number of possible n-grams increases by the size of the alphabet $|\Sigma|^n$. This makes the training data sparser, and will require more data to cover possible n-grams, resulting in an increase in the size of the model. Sparsity in data was counteracted by using Laplace smoothing [19]. This results in low-occurrence but non-zero n-grams having less impact on the probabilities.

This provided the best perplexity of the n-gram models in informal tests.

Based on the findings of [12] we attempt to find an appropriate balance between context and available data. Removing all occurrences in an n-gram model with the most common probability can help reduce the size of the model. Removing all n-grams with a probability of 25% in an n-gram model created from a small dataset can greatly reduce the size of the model as all occurrences of $n-1$-grams not seen are removed. n-gram models created from larger dataset can however not be decreased in size as much due to the occurrence of more unique n-grams.

E. RNN Language Model

Because of the context limitations of the n-gram model, we chose to also implement an neural language model. Both a feed-forward network as well as an RNN can be used to build a neural language model, however a feed-forward neural network can only take a fixed number of inputs, meaning the history size will be fixed. For this reason, we chose to implement an RNN language model, since this model can take a variably sized number of inputs. Common recurrent architectures make use of Long Short-Term Memorys (LSTMs) or Gated Recurrent Units (GRUs) units, since they address the problem of learning long-term dependencies of sequences introduced by recurrent neural networks. We chose to use a GRU over an LSTM because of the fewer operations required at inference time. This is important because the language models needs to be applied $T \times k \times |\Sigma|$ times, where T is the number of time-steps, and k is the beam width meaning how many beams are expanded each timestep.

With any character-level language model we are interested in $p(c, t|hi)$, that is the probability of character c at time t given a history hi. Our GRU network works by recursively computing a hidden state h_t using the previous hidden state h_{t-1} for each one-hot encoded character x_t in the input DNA sequence x. The first hidden state vector h_0 is initialized to zero in each position. Each hidden state h_t is a vector larger than the one-hot encoded inputs x_t, which allows it to encode long sequences. A linear feed-forward layer is used to downscale each hidden vector to $|\Sigma|$ outputs. This process is defined in Equation 7. After that, a softmax activation function is applied to compute the output probability distribution $\hat{y}_t$, which contains the conditional probability of each c that we are interested in at a given timestep t.

$$\begin{aligned} h_t &= gru(h_{t-1}, x_t) \\ \hat{y}_t &= S(ff(h_t)) \end{aligned} \tag{7}$$

where ff is the feed-forward network and S is the softmax function. A detailed description of GRU is given in Appendix VI-A.2. The softmax activation function is defined as:

$$S(y_i) = \frac{exp(y_i)}{\sum_j exp(y_j)} \tag{8}$$

where y_i is a single character, and the output of the softmax is a probability distribution over all characters.

Since the model is a multi-class classification model, and outputs probabilities we used cross entropy loss. With y as the ground truth label, it is defined as:

$$CrossEntropyLoss(\hat{y}_t, y) = -log\left(\hat{y}_t(y)\right) \tag{9}$$

Cross entropy loss takes the probability of the correct label $\hat{y}_t(y)$ and applies the log function and negation. This results in a much higher loss on wrong predictions compared to correct predictions.

4. Data

The two main sets used in this paper are the trainingset for ONT basecallers and a dataset provided by Center for Microbial Communities (CMC) at Aalborg University. The two main differences between the datasets are the number of unique species and the amount of data.

A. Oxford Nanopore Technologies Dataset

The dataset provided by ONT can be downloaded through the Bonito basecaller Github repository [7], and will be referred to as the ONT-dataset. The dataset is comprised of 66149 reads, with mapped references ranging from 800-80000 bases, sequenced from full genomes. While the dataset is made public by ONT the specific species of DNA contained in the dataset has not been made public. However we have calculated an approximate list of references using the Kraken2 taxonomic classification system developed by the Center for Computational Biology at Johns Hopkins University [20] resulting in 496 unique taxonomic ids. Training and basecalling on full genomes is the method used by ONT basecallers. The ONT-dataset is used to train all ONT basecallers [14].

B. Center for Microbial Communities Dataset

The dataset provided by the CMC at Aalborg University contains 480605 reads ranging from 4000-5000 bases [21], and will be referred to as the CMC-dataset. These reads are produced using a method called DNA-fingerprinting [22]. This method produces a short 4000-5000 base DNA string uniquely representing a specific genome. The dataset provides 8 different species with some species being represented by several DNA fingerprints (based on their position on the genome) resulting in 48 unique reference sequences in total. Individual reference sequences from the same bacteria are referred to as parts and denoted with a number such as *Listeria_1*. In this paper we are using this dataset due to the shorter length of reads and the availability of full references making mapping and consensus accuracy much easier to calculate.

5. Implementation Details

Our implementation is based on the open source Bonito basecaller [7], and an open source CTC decoding library [23], both of which are developed and published by ONT.

The Palamut architecture consists of three main parts; a QuartzNet model, a language model and the prefix beam search decoder which combines the two models. The QuartzNet model accepts nanopore sequences and output CTC-probabilities. It has not been altered from the original Bonito implementation of a QuartzNet model. As we have made no changes to the architecture of the signal model, details regarding training a signal model has been included in Appendix II. Language models are implented in Python. The prefix beam search decoder is a fork of the original CTC decoding library [23], modified to accept language model probabilities based on our documentation in subsection 3-B. Across all prefix beam search implementations, a threshold value was used to prune beams. If the CTC network probability of character c at a timestep is under the threshold the beam will not be extended by that character. In all our tests we used a threshold of 0.001.

A. n-gram language model

The statistical n-gram language model is implemented in Python, and is represented with a Python dictionary where each key is an n-gram, and the value is the n-gram conditional probability. We implemented two versions of the prefix beam search decoder for n-gram language models. The first version was implemented in Python as a single function, and the Bonito basecaller was modified to accept a parameter which switches to use our prefix beam search with language model. The second version is a modification of the existing beam search decoder [23] used by the Bonito basecaller. This version is implemented in Rust and wrapped with PyO3 [24] such that it is callable from the Python code. The Rust beam search decoder was modified to additionally, accept the language model as well as the α and β values as documented in subsection 3-B. This implementation came with an increase in speed as Rust is inherently faster than Python, because it is a compiled and statically typed programming language which allows for the compiler to optimize code further, than interpreted programming languages like Python are able to. Due to the more acceptable run-time of the Rust implementation, we did not use the Python implementation for tests.

B. RNN language model

We used PyTorch to train our RNN language model. Our implementation was based on an implementation of a character-level LSTM [25].

To implement prefix beam search with our RNN language models, we again modified the Rust beam search implementation since we suspected better run-times compared to a Python implementation. The run-time was 4-16 times slower than tests performed using prefix beam search with an n-gram language model, which was acceptable given that the overall complexity is higher for with the RNN language model.

In both Rust implementations, we implemented the application of language model functions in Python and used PyO3 to call Python code from Rust. That means our language models are first loaded in Python, passed to Rust as an object, and every time the language model is applied, a Python function is called from Rust. We chose this approach as none of the authors had experience with Rust, so the more complex code could be handled in Python. We suspect both run-time and memory usage could be reduced, if the language models were exclusively loaded and applied in Rust. For the RNN language model we made some optimizations to reduce the number of calls from Rust to Python.

We first optimized by only computing the hidden state and output once for every beam. Extending a beam requires an application of the language model. For the RNN language models, computing a language model probability always results in a probability distribution over each possible character extension. The distribution is therefore saved temporarily and reused. This saves up to three GRU computations for each beam.

Secondly we observed after filtering by the k best beams after each time-step, the same beams often persisted over a few time-steps. We optimized our code by reusing the language model probabilities if the beams did not change over time-steps, which reduced number of Rust to Python calls. Another way we tried optimize run-time was by batching the hidden state and output computations of the k active beams, but this worsened the run-time.

6. Experiments

This section covers experiments conducted in the paper. Read- and consensus accuracy are used to measure results of the tests. Read- and consensus accuracy are further described in Appendix V. Each test results in a fasta file containing basecalls produced from the testset. To calculate read accuracy, we firstly remove all basecalls obviously containing errors, this mainly includes removing all basecalls having not having lengths between 4000-5000 bases, a common approach when basecalling on DNA with an already known approximate length. This approach only filters $<1\%$ of basecalls and the same filters are applied to each test. Remaining basecalls are then mapped to their respective reference using Minimap2 [26]. These mappings are analysed using the *read_length_identity.py* code produced by Ryan Wick [27] resulting in an individual read accuracy for each basecall. These values are averaged over each genome present in the test set. Consensus accuracy is produced by separating basecalls by their respective DNA reference. Each list of basecalls is clustered using usearch11 [28] to produce a seed for future use. Each basecall is then mapped to this seed using Minimap2, and polished using Racon and the resulting mapping [29]. This Minimap2-Racon procedure is in total done three times per basecall list. The output of the final Racon procedure is used as the final seed for the Medaka software [30] to produce a final consensus sequence from each list of basecalls. Finally the consensus sequence is compared to the reference using blastn [15], producing a consensus accuracy along with a list of errors in the consensus sequence. The approach for reaching a consensus sequences is taken from a demonstation provided by CMC. The original demonstation file is present in our code hand-in. The read accuracy is denoted as Read id. and the consensus accuracy is denoted as Cons. id. in the test results tables.

A. Baseline Tests and Models

To increase our knowledge on the impact of language models and diversity in training data, several QuartzNet models were trained and analysed without the use of language models. The data used for training the models is provided by CMC at Aalborg University, further described in subsection 4-B. As it is unfeasible to train basecallers on all known bacteria, general-purpose basecallers have to succeed in producing high-accuracy basecalls from nanopore data from species not included in the trainingset. While a basecaller-model can be trained to produce high-accuracy reads for a single bacteria, general-purpose basecallers such as the models provided by ONT are able to work on a broader array of data without needing further training.

Test	Avg. Read id.	Avg. Consensus id.
CMC-models	87.88%	96.19%
Standard Bonito	93.51%	99.98%
Standard Taiyaki	92.97%	99.98%

TABLE I: Results of baseline tests

To emulate this we used a k-fold cross-validation approach where 8 models were trained using the bonito architecture, and 5% of reads from 7 different bacteria was used as the trainingset. 5% of reads from an 8th bacteria was used as a testset, resulting in 8 models each with one of the 8 bacteria in the CMC-dataset removed.
Training 8 models this way helped identify potential problems, e.g. if the DNA sequence of a bacteria was too distinctive to identify without prior knowledge of that bacteria. These models were trained with the default hyperparameters in the Bonito code. The hyperparameters and overall training process is described in Appendix II. Models used in tests were selected by choosing the checkpoint with the lowest validation loss during training. Models trained using the data from CMC will be referred to as CMC-models, and are always tested on bacteria unknown to the models (e.g. when a CMC-model is described as being tested on the Bacillus bacteria, or as the Bacillus CMC-model. The CMC-model in question refers to the bonito model trained using 7 of the 8 bacteria present in the CMC-dataset, Bacillus being the one left out). In addition to the 8 models, we also used the pre-trained models provided by the Bonito and Taiyaki basecallers. In this paper we will refer to these as the standard Bonito model and standard Taiyaki model, both are trained on the ONT-dataset containing nanopore data from a broader array of bacteria described in our previous data section. These two models are included as they are the standard models provided with the individual basecaller software. The Taiyaki model is included as a comparison of state-of-the-art basecallers, to show the strength of the Bonito basecaller.

The CMC-models were tested using the bacteria not known to the model individually, while the standard Taiyaki and Bonito models were applied to the entirety of the testset. The 8 CMC-models along with the model will be used in future tests.

1) Baseline Results: The results of the baseline tests can be seen in Table I. For ease of reading results from the CMC-models have been combined in one row. A full list of baseline test results can be seen in Table IV. By examining the individual results of the baseline tests some relevant information can be extracted. When basecalling using the CMC-models, bacteria such as Salmonella and Escherichia receive a higher read accuracy. Using the blastn software suite [15], we can compare references for the two bacteria showing that they share 97% of their bases, resulting in a cross-validation performing better. Bacteria such as Pseudomonas and Lactobacillus receive lower accuracy due

to their DNA sequences sharing at best 88% of bases with other bacteria. Trainingsets for both the Standard Bonito Model and Standard Taiyaki model contain a larger amount of unique DNA sequences resulting in more uniform results for each individual Bacteria. While the Standard Bonito and Taiyaki models produce better accuracies, our CMC-model tests can be used to show the impact of a language model with a smaller sequence model. The standard Bonito model will be used to test the impact of language models on a state-of-the-art basecaller.

B. Language Model Tests

The results of the baseline tests were originally intended for future comparison. However, to ensure that the baseline results were not based on outlier data, the volume of the final testset was increased to 10% of all reads from each bacteria in the CMC-dataset. This was used to replicate the baseline tests and test the effect of using language models when basecalling.

1) n-gram model experiments: To test our n-gram language model, several informal tests were conducted to find the most appropriate n-gram size. Based on the findings in [12] we produced an n-gram language model for each $4 \leq n \leq 18$ using all reference parts from 7 of the 8 bacteria in the CMC-dataset. We then used the final bacteria references to calculate perplexity as a measurement to find potential sizes of n. While an n-gram size of 8 showed the lowest perplexity of $\sim$3.2, when used for basecalling a small subset containing 30 reads for the final bacteria, the 12-gram language model performed the best with a perplexity of $\sim$3.4. This is most likely due to the reduction in context when only having a history window of 7 in the 8-gram model as opposed to 11 in the 12-gram model.

8 12-gram models were created using the same leave-one-out approach as with the CMC-models. Each model was created using the 48 unique references from the dataset except for references for one bacteria. This was to ensure tests where neither the trained CMC-model nor the language model had prior knowledge of the testset. Laplace smoothing was applied to each language model and all entries in the language model having a probability of 25% were removed to reduce the size of the language model. Informal tests were also conducted using the ONT-dataset to create 12-gram language model. As the ONT-dataset does not include full references, a 12-gram language model was created from the individually mapped references included in the training set. This meant that the n-gram model could potentially have been biased towards bacteria more prominent in the test set. The large 12-gram model produced from the ONT-dataset did not perform well when testing perplexity and had increasing basecalling times due to its size. The increase in size of the ONT-n-gram model was due to more unique 12-grams present in the dataset. As a result no further testing was conducted using n-gram models produced from larger datasets. The α and β values used in the prefix beam search algorithm described in subsection 3-B were then tuned using hyperOpt [31] library with a uniform search space ranging from 0-2 for both the α and β values. The values were tuned for 25 epochs using the *HyperOpt.tpe.suggest* search algorithm. The Staphylococcus CMC-model and testset used in the original baseline tests were used as a validation set for the tuner. The resulting α and β were 0.22765541627585756 and 1.9958819811331983 respectively. Finally the models were tested using the CMC-models from our base tests on the testset containing 10% of reads. Results can be seen in Table II.

2) RNN model experiments: As an RNN language model does not increase in size, we opted to train our RNN language model on the ONT-dataset due to the broader array of bacterial references present in the dataset. This ensured that the RNN model represents a larger corpus and takes advantage of the fixed size of an RNN. The ONT-dataset was passed through the Kraken2 [20] software to identify taxonomi ids for each read. All unidentified reads were removed along with all reads mapping to Listeria or Staphylococcus. The removal of Listeria and Staphylococcus was to ensure that at least two of our test references had never been seen by the language model. Instead of using one instance of each full references as used in the statistical n-gram models, each mapped reference was extracted from the filtered dataset and fed to the RNN language model for training.

The ONT RNN model was trained for 1100 epochs with a batch size of 2048 and a sequence length of 150 bases. The learning rate was started at 0.001 and optimized using the Adam optimizer [32]. The language model training was stopped at 1100 epochs as the validation loss did not decrease further. The perplexity of the ONT language model was calculated to be 2.6 when attempting to predict all Listeria references from the CMC-dataset. The α and β values used in the prefix beam search decoder were then tuned using the model and testset from the baseline tests for the Staphylococcus bacteria. HyperOpt was used with the same search space as in our n-gram tests. The resulting α and β were 0.3669682468345905 and 1.302195478016107 respectively. Finally the RNN language model was tested using both the standard Bonito sequence model and our trained CMC-models from the baseline tests. The testset itself is the same as the one used for testing n-gram models containing 10% of reads from each bacteria part. Results from all tests can be seen in Table II.

C. Results Summary

As can be seen on both our RNN and n-gram language model experiment, applying our language models does not improve read accuracy of the bonito basecaller. Both language models resulted in a decrease in read accuracy when used in conjunction with a basecaller. These result could potentially be caused by the α and β values not having been tuned properly or the language models not modeling the domain properly. The RNN language model could potentially have been trained on fingerprint data as opposed to the full genome data provided by ONT. This was not explored further

Seq. Model	Test Bacteria	No LM Read id.	n-gram Read id.	RNN Read id.	No LM Cons. id.	n-gram Consensus id.	RNN Cons. id.
Std. Bonito	All	**93.55%**	-	93.16%	99.979%	-	**99.996%**
CMC model	Bacillus	**89.7%**	89.44%	89.5%	97.37%	97.4%	**97.51%**
CMC model	Enterococcus	**89.4%**	89.20%	89.36%	96.61%	96.55%	**96.72%**
CMC model	Escherichia	**89.90%**	89.63%	89.66%	98.42%	**98.48%**	98.45%
CMC model	Lactobacillus	**85.14%**	84.85%	84.79%	95.05%	94.94%	**95.12%**
CMC model	Listeria	**88.62%**	88.32%	88.39%	96.48%	96.35%	**96.68%**
CMC model	Pseudomonas	**82.39%**	82.14%	82.05%	92.99%	92.91%	**93.44%**
CMC model	Salmonella	**90.97%**	90.73%	90.73%	98.33%	98.37%	**98.50%**
CMC model	Staphylococcus	**87.08%**	86.90%	87.01%	93.48%	93.82%	**94.40%**

TABLE II: Results from both n-gram and RNN language model tests. The CMC models used in tests are trained using all bacterial references except for the test bacteria. Tests denoted will All as test bacteria are tested on the entirety of the testset.

in this paper, as a larger training set was available through the ONT-dataset proving in our base tests to be capable of producing accurate basecalls of the CMC-dataset reads. Several more tests could have been conducted to further test the decrease in read accuracy, mainly using α and β values tuned further and training individual models for each bacteria used during the baseline tests. Due to the ONT-dataset producing high perplexity n-gram models with increased basecall times further tests were not conducted on n-gram models either. The higher perplexity and slower basecalling times makes n-gram models less optimal when created from datasets containing a broad spectrum of bacterial references. While the read accuracy decrease with the addition of language models, the consensus accuracy did increase by a small margin. Consensus accuracy is commonly used to check for systematic errors in basecallers and an increase in the consensus, by applying a language model we enforce common behaviour in the basecalled DNA sequences that could potentially decrease systematic errors. As show in Table II n-gram language models, trained on a relatively small set of references, can help increase consensus accuracy in some cases, without much increase in basecalling times. An RNN trained on a larger corpus produces similar results, with a slightly higher consensus accuracy. Extracting the result of using the blastn suite to map a consensus sequence to our references shows that without using a language model each consensus sequence contains between 40-100 errors. Applying a language model decreases errors in consensus sequences by 1-10 on average. It should be noted that a small number of consensus sequences contain more errors than its non-LM counterpart. However as can be seen in Table II consensus accuracy is increased overall. Inspecting the output from blastn also shows that consensus sequences both with and without the addition of language models share a majority of their errors. This shows that language models overall do not introduce new systematic errors, but are currently not able to resolve the majority of existing ones. Comparing individual basecalls from the Bacillus_1 part show that the average length of basecalls are closer to the reference length by a margin of 40-90 bases when using language models. Using this knowledge we assume that the β values used to induce a length bias, is not the main cause of decreasing read accuracies.

7. Conclusion

Our experiments show that attempting to identify patterns in DNA sequences using language models, can help improve consensus accuracy of the Bonito basecaller in some cases. In our experiments, using the pre-trained bonito basecaller model in combination with a trained RNN, we increased the consensus accuracy from 99.979% to 99.996%, the read accuracy however decreased from 93.55% to 93.16%. The current version of our implementation also increases basecalling time substantially. Decoding using our n-gram model doubles the basecalling time due to each beam containing the added operation of looking up language model probabilities.

The RNN model increases basecalling time by between 4-16 times due to the additional complexity. We however suspect these increases to be because of a significant run-time overhead in our prefix beam search implementation from using a combination of Python and Rust. An implementation exclusively in Rust or other high-performance compiled programming language should remove this overhead.

Comparing results to basecalls produced using no language model shows that the β value is most likely not the cause of decreased accuracies, as each basecall produced using language models are closer in length to the reference than those produced without. As the consensus accuracy is increased by introducing language models, we can determine that the decrease in accuracy is not caused by introducing new systematic errors, but rather random ones. These errors are most likely caused by the addition of language model probabilities and the α-value controlling the language model influence. Further tuning of the α value could potentially reduce errors introduced by the language model. As the main purpose of the language model is to only have an effect on the output when the Quartznet model outputs similar probabilities, modifying α to have less impact could reduce random errors, but potentially also reintroduce systematic errors alleviated by the current α-value.

Training an RNN language model using a wider array of species along with performing hyperparameter tuning could potentially resolve this problem. Especially the sequence lengths used for training the RNN could be increased to include more contextual knowledge. Using a sliding window for choosing the training samples instead of simple segmentation could also be used, in order to learn

dependencies between all adjacent samples. During informal testing it was discovered that 12-gram language models produced from a corpus containing many different species would results in much higher perplexity and increased basecalling times. Clustering similar genome references and training language models on individual clusters could potentially alleviate this problem, when used for basecalling a genome similar to the respected cluster. This approach would either require a nanopore sequence to be basecalled multiple times using different language models, or attempting to identify which language model to use before basecalling. This approach would however introduce additional overhead and increase basecalling times substantially.

APPENDIX I
QUARTZNET

QuartzNet is an end-to-end neural acoustic model for automatic speech recognition [9]. The authors claim the network has a near state-of-the-art accuracy, while having fewer parameters than competing models. It is composed of 1D convolutional layers, most of which are 1D time-channel separable convolutional layers, batch-normalization and ReLU activation functions. An overview of the QuartzNet model can be seen in Figure 9. For a 1D convolutional layer with kernel size K, c_{in} input channels, and c_{out} output channels. A 1D convolution has $K \times c_{in} \times c_{out}$ parameters, where K is the kernel size, c_{in} is the input channels, and c_out is output channels. A time-channel separable convolution has $K \times c_{in} + c_{in} \times c_{out}$ parameters split into $K \times c_{in}$ parameters for a depthwise layer and $c_{in} \times c_{out}$ for a pointwise layer. The depthwise layer works on one channel at a time and across K time frames, and a pointwise layer works on all channels for each independent time frame. Figure 2 illustrates the operations of a normal convolution and a time-channel seperable convolution.

The QuartzNet 5x5 model consists a sequence of blocks. With the exception of the last block C_4 all blocks start either a normal convolutional or a time-channel separable convolution layer, followed by a batch normalization layer and ReLU activation function. The model starts block C_1 with a 1D convolutional layer, followed by 5 blocks B_1, B_2, B_3, B_4, B_5. Each block B_i consists of 5 sub-blocks all of which has the same kernel size and output channels. The input of a B block is residually connected to the last sub-block. The B blocks are followed by 3 blocks C_2, $C3$ and C_4 which. The parameters of the 5x5 model can be seen on Table III.

The Bonito architecture is similar to the QuartzNet architecture, but has a small distinction in that Bonito use a stride of 3 in block C_1 while QuartzNet use a stride of 2. QuartzNet also use a dilation rate of 2 in block C_4 where Bonito don't use dilation (i.e. dilation rate of 1).

APPENDIX II
BONITO TRAINING

Bonito models are not trained on full sequences, but splits each reference sequence into smaller variably sized

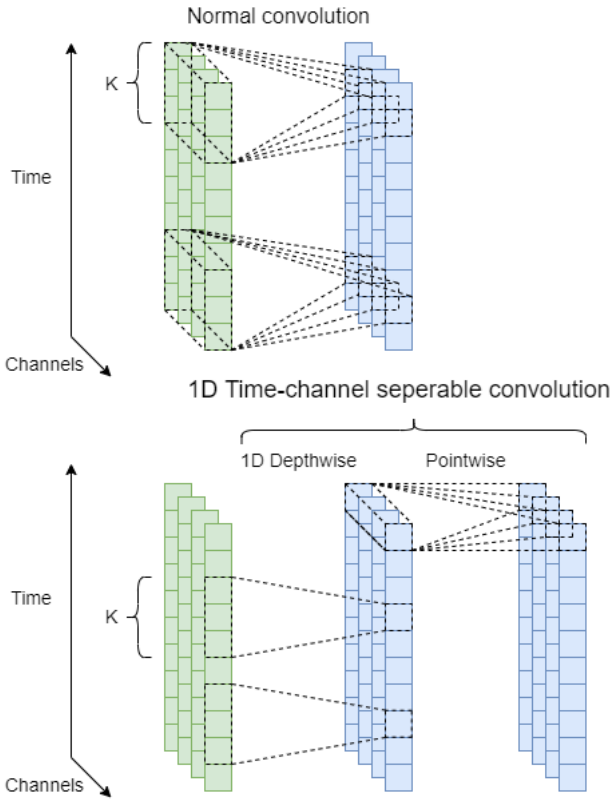

Fig. 2: The top illustration show a normal convolutional layer that operates across K time-steps and all channels. The bottom illustration shows a 1D time-channel separable convolution which splits the operations into two steps. First a 1D depthwise convolution operates across K time-steps. Second a pointwise convolution operates across each time frame independently but across all channels.

Block	R	K	C	Convolution
C_1	1	33	256	Conv
B_1	5	33	256	TCS
B_2	5	39	256	TCS
B_3	5	51	512	TCS
B_4	5	63	512	TCS
B_5	5	75	512	TCS
C_2	1	87	512	TCS
C_3	1	1	1024	Conv
C_4	1	1	$\|\|\Sigma\|\|$	Pointwise

TABLE III: QuartzNet architecture

chunks of size 200-400. In order to split the references into chunks a mapping is required, which shows which parts of the nanopore sequence matches with the reference chunks. The mapping is created between each base in the reference and each measurement in the nanopore sequence using preprocessing functions in the ONT Taiyaki basecaller [33]. We described how these function work in [8].

Nanopore sequences are split into chunks, using the mapping produced by Taiyaki. To allow for batching of training samples all nanopore chunks are padded with zeros, such that all chunks are $400 \cdot max_{mpb}$ long where max_{mpb} denotes the maximum measurements per DNA base.

The bonito models all have 6.7M model parameters,

which is defined by the layer sizes in table III, and overall QuartzNet architecture seen on Figure 1. The models are trained with CTC loss, for 400 epochs, with batch size of 32, learning rate 0.003. 1% of each training dataset was used as a validationset. The model parameters are saved after each epoch, and lowest validation loss epoch is used for testing afterwards.

APPENDIX III
TEST RESULTS

Table IV shows each individual bacteria basecalled during the baseline tests. Read-id and Consensus-ids are calculated from the average of reads relating to that bacteria. Individual bacterial parts such as Bacillus_1 and Bacillus_5 might have different average values, but for ease of reading they have been collected into their respective bacteria in the table.

Test	Test Bacteria	Read-id	Consensus-id
Standard Taiyaki	Bacillus	92.67%	99.99%
Standard Taiyaki	Enterococcus	94.09%	99.98%
Standard Taiyaki	Escherichia	92.94%	99.99%
Standard Taiyaki	Lactobacillus	92.82%	99.95%
Standard Taiyaki	Listeria	92.76%	100%
Standard Taiyaki	Pseudomonas	93.02%	100%
Standard Taiyaki	Salmonella	92.99%	99.96%
Standard Taiyaki	Staphylococcus	93.48%	99.97%
Standard Bonito	Bacillus	93.29%	99.97%
Standard Bonito	Enterococcus	93.78%	100%
Standard Bonito	Escherichia	93.81%	99.97%
Standard Bonito	Lactobacillus	93.50%	100%
Standard Bonito	Listeria	93.45%	99.98%
Standard Bonito	Pseudomonas	93.67%	99.98%
Standard Bonito	Salmonella	93.80%	99.97%
Standard Bonito	Staphylococcus	93.51%	99.99%
CMC-model	Bacillus	89.68%	97.50%
CMC-model	Enterococcus	89.77%	96.76%
CMC-model	Escherichia	89.77%	98.40%
CMC-model	Lactobacillus	85.05%	95.06%
CMC-model	Listeria	88.59%	96.47%
CMC-model	Pseudomonas	82.38%	93.06%
CMC-model	Salmonella	90.97%	98.40%
CMC-model	Staphylococcus	87.88%	93.85%

TABLE IV: Individual results from baseline tests

APPENDIX IV
PERPLEXITY

In Natural Language Processing (NLP), perplexity is one way to measure a how well a language model predicts the test data. The intuition behind perplexity is how perplexed the model is when presented with sequences it has never seen before [34]. To calculate perplexity for a language model, a testset of m sequences, $s_1, s_2, s_3, \ldots, s_m$ is used. The joint probability of all sequences is $\prod_{i=1}^{m} p(s_i)$. In a character-level language model the probability of sequence s_i with n characters is $\prod_{j=1}^{n} p(c_j|h)$, where c_j is the jth character, and h is the history window which is previous characters to c_j. Particularly in n-gram models the history window size is $n-1$.

To avoid underflow the log sequence probability can be computed as:

$$log \prod_{i=1}^{m} p(s_i) = \sum_{i=1}^{m} logp(s_i) \tag{10}$$

The perplexity of the language model is calculated as Perplexity $= 2^{-l}$ where l is calculated using Equation 11

$$l = \frac{1}{M} \sum_{i=1}^{m} logp(s_i) \tag{11}$$

M is the total number of characters across all sequences, which normalizes the log probabilities by the total number of characters.

This measure is an indication of, how well a language model can predict sequences in a testset. In our case perplexity was used for evaluating language models before they were combined with a bonito model.

APPENDIX V
BASECALLING ACCURACY

In this domain, there are 2 types of target accuracy; read- and consensus accuracy. Read-accuracy refers to the accuracy of the basecall, for a single read in regards to the reference. Meaning, that a single read is basecalled, and then mapped to the reference sequence. When the mapping algorithm has found the most probable mapping, the basecall is matched against the reference and the accuracy is calculated. Here, there are two types of errors that can influence the accuracy of a read, being random errors and systematic errors. Random errors can come from e.g. noise in the nanopore, a DNA strand that got stuck in the pore.

Consensus accuracy is then the average sequence of the reads. Meaning, that if more than half of the reads for a given sequence reads an ”A” at a given position and some of the reads has a ”G” at that position, then the consensus sequence will contain an ”A” at that position. The consensus accuracy is then how accurate the consensus sequence matches the reference. The consensus sequence should have eliminated any random errors from the reads, and the errors that persists are systematic errors.

APPENDIX VI
PREVIOUS WORK

This appendix contains sections first presented in our first report [8]. These are sections used to define and describe key aspects of the domain, both from a computer science aspect as well as biology aspect. The sections have been modified to correct spelling, and inter-report references have been changed to correctly cite our previous work, along with the referenced section, table and such. These have been marked with ”**”.

A. Recurrent Neural Network

RNN is a type of neural network that uses a node structure like a directed graph, divided into steps on a temporal line. So at each timestep, the input is fed to the internal hidden state of the model, along with the previous hidden state, which gives the model the recurring ability. So each time

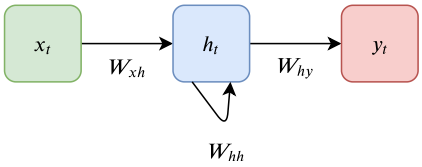

Fig. 3: A simple RNN [8].

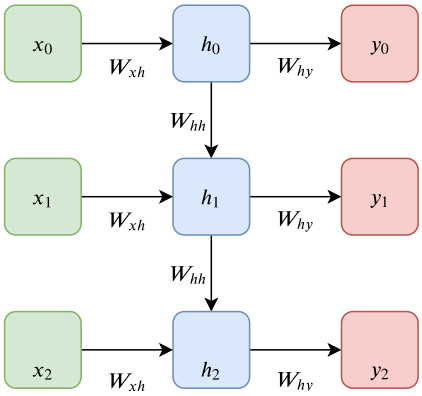

Fig. 4: A simple RNN undfolded [8].

the model reads an input, the hidden state is updated and at some point an output is calculated. A simple version of an RNN is depicted in Figure 3. A better intuition of the temporal properties of an RNN is shown in Figure 4.

Here, it is clear to see that at each timestep, the input is fed to the hidden state, along with the previous hidden state to produce some output. Then, the updated hidden state is passed as input to the next timestep. Note that it is entirely possible to have input, output or both be of variable lengths, which shows the versatility of an RNN model.

1) LSTM: An approach to solving the vanishing gradient problem is the LSTM method. An illustration of an LSTM unit is shown in Figure 5. LSTM introduces an internal memory cell and has three gates which control how much information is used from the three inputs; $\boldsymbol{C}_{t-1}$ the memory cell from last step, $\mathbf{h}_{t-1}$ the last hidden state and $\mathbf{x}_t$ the next input sample. Each LSTM unit computes a candidate memory cell ($\tilde{\boldsymbol{C}}$). The new memory cell ($\boldsymbol{C}_t$) is made by combining the previous memory cell ($\boldsymbol{C}_{t-1}$) and the candidate memory cell. The forget gate ($\mathbf{f}_t$) and input gate ($\mathbf{i}_t$) controls how much information the previous memory cell and candidate cell respectively should influences the creation of the new cell. The output gate ($\mathbf{o}_t$) controls how much of the memory cell is passed to the next hidden state ($\mathbf{h}_t$). All LSTM equations are shown in Equation 12.

$$
\begin{aligned}
\mathbf{f}_t &= \sigma(\boldsymbol{W_{hf}}\mathbf{h}_{t-1} + \boldsymbol{W_{x}f}\mathbf{x}_t + \mathbf{b_f}) \\
\mathbf{i}_t &= \sigma(\boldsymbol{W_{hi}}\mathbf{h}_{t-1} + \boldsymbol{W_{x}i}\mathbf{x}_t + \mathbf{b_i}) \\
\tilde{\boldsymbol{C}}_t &= tanh(\boldsymbol{W_{hC}}\mathbf{h}_{t-1} + \boldsymbol{W_{xC}}\mathbf{x}_t + \mathbf{b_C}) \\
\boldsymbol{C}_t &= \mathbf{f}_t\boldsymbol{C}_{t-1} + \mathbf{i}_t\tilde{\boldsymbol{C}}_t \\
\mathbf{o}_t &= \sigma(\boldsymbol{W_{h}o}\mathbf{h}_{t-1} + \boldsymbol{W_{xo}}\mathbf{x}_t + \mathbf{b_o}) \\
\mathbf{h}_t &= \mathbf{o}_t tanh(\boldsymbol{C}_t)
\end{aligned}
\tag{12}
$$

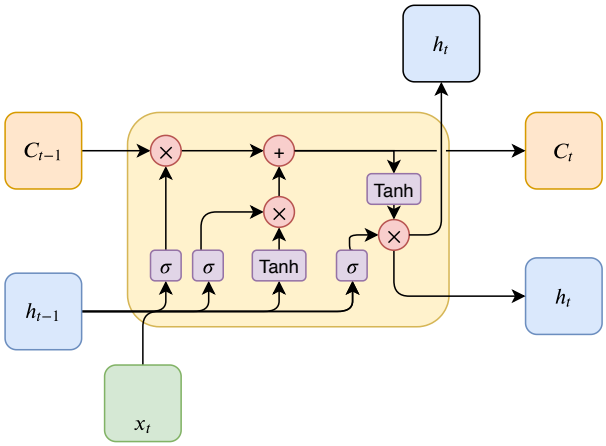

Fig. 5: An LSTM unit

2) Gated Recurrent Unit: GRU is a simpler version of LSTM and can be seen in Figure 6. It only has two gates and no memory cell. The update gate ($\mathbf{z}_t$) behaves similarly to the forget and input gate of an LSTM. It decides what information to forget from the previous hidden state, and what new information to add from the input, and puts this info into a new candidate hidden state $\tilde{\mathbf{h}}_t$. The reset gate ($\mathbf{r}_t$) decides how much information of $\mathbf{h}_{t-1}$ should be kept and how much should be replaced $\tilde{\mathbf{h}}_t$. All GRU equations are shown in Equation 13.

$$
\begin{aligned}
\mathbf{z}_t &= \sigma(\boldsymbol{W_{hz}}\mathbf{h}_{t-1} + \boldsymbol{W_{xz}}\mathbf{x}_t) \\
\mathbf{r}_t &= \sigma(\boldsymbol{W_{hr}}\mathbf{h}_{t-1} + \boldsymbol{W_{xr}}\mathbf{x}_t) \\
\tilde{\mathbf{h}}_t &= tanh(\boldsymbol{W_{rh}}\mathbf{r}_t + \boldsymbol{W_{xh}}\mathbf{x}_t) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t)\mathbf{h}_{t-1} + \mathbf{z}_t\tilde{\mathbf{h}}_t
\end{aligned}
\tag{13}
$$

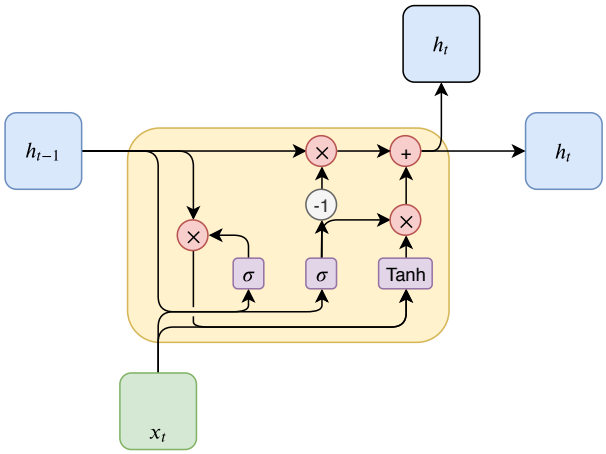

Fig. 6: A GRU unit

B. Connectionist Temporal Classification

CTC is used to derive an objective function in sequence-to-sequence models, where the input is shorter than the output, and where an alignment between the input and output has to be inferred [17]. It is often used in speech

recognition because the amount of samples in audio is higher than the amount of words or phonemes to be recognized. In conjunction with a many-to-many RNN, it can be seen as an alternative to a encoder-decoder model (see **[8, sec. 4.2.1]). It is used in the nanopore basecallers Albacore and Chiron(see **[8, Appendix. A]).

The CTC layer is the top layer, in nanopore CTC models.

To train a CTC model the CTC layer accepts a sequence of probability distributions $\boldsymbol{Y} = (\mathbf{y^1} \dots \mathbf{y^T})$ over the alphabet $L' = L \cup \{-\}$, where $L = \{A, C, T, G\}$ in the context of nanopore sequencing, and a corresponding target sequence $\mathbf{z} = (z_1, z_2, ..., z_U)$ [35]. In nanopore sequencing $\mathbf{z}$ is the reference base sequence. The probability distributions $\boldsymbol{Y}$ can be generated by an RNN with a softmax function, that accepts the input sequence $\mathbf{x}$. Figure 7 shows an illustration of probability distributions derived from an RNN with a softmax function.

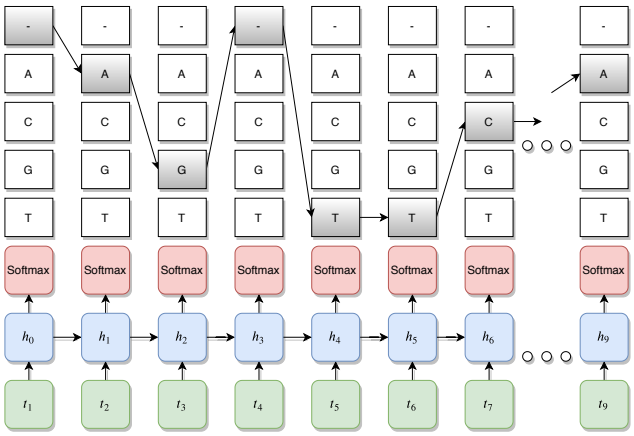

Fig. 7: An illustration of outputs of a RNN with softmax function. The letters -,A,C,T,G represents a probability distribution where the most probable letter is marked. The arrows describe the most probable path. Note only some timesteps are shown.

The rest of this section use notation and equations derived from Graves et al. [35]. The probability generated by each activated output unit k at time t is denoted as y_k^t. The number of output units correspond to the size of L', in this case $K = |L'| = 5$. The activation of the unit to $-$, called the blank character, is the stay state. The stay state is used when multiple adjacent measures of $\mathbf{x}$ correlate the same base z_u.

It can also be explained as the probability of the current being measure over the middle of two bases of the same type. In practice this is hard to distinguish, which we will go in depth with in **[8, sec. 4.7]. The conditional probability of an arbitrary path π through the generated CTC network is:

$$p(\pi|\mathbf{x}) = \prod_{t=1}^{T} y_{\pi_t}^t \tag{14}$$

Note that all paths has to start on a state in t_1, and go through exactly one state for each time-step, in sequential order, and end in a t_T state. We can use this to define a distribution over the set of all possible paths generated from x of length T sequences over the alphabet L':

$$p(\pi|\mathbf{x}) = \prod_{t=1}^{T} y_{\pi_t}^t, \forall \pi \in L'^T \tag{15}$$

Next we can define a many-to-one map $B : L'^T \mapsto L^{\leq T}$, where $L^{\leq T}$ is the set of possible labels over the original alphabet L, with length less or equal to T. This is done by collapsing the characters generated by a path, by removing all blank characters and repeated characters. For example $(TT - C - AA - A - A - -) \rightarrow (TCAAA)$.

B is used to define the conditional probability of a labelling sequence $\mathbf{l} \in L^{\leq T}$ as the sum of all the probabilities of all the path that generate the labelling $\mathbf{l}$:

$$p(\mathbf{l}|\mathbf{x}) = \sum_{\pi \in B^{-1}(\mathbf{l})} p(\pi|\mathbf{x}) \tag{16}$$

The output of the classifier should be the most probable labelling sequence:

$$h(\mathbf{x}) = \underset{\mathbf{l} \in L^{\leq T}}{\operatorname{argmax}}\, p(\mathbf{l}|x)$$

However to find the most probable labelling sequence one needs to compute the probability of every possible labelling sequence, which the number of is exponential to the input sequence length; $|L'|^T$. The length of a nanopore read is up to 50kbp, which makes it infeasible to compute the probability of each labelling sequencing. Therefore a heuristic is used instead. The simplest heuristic is to use the most likely path. The probability of most the likely path π_{max} is trivial to compute, as it is the product of the highest probability at each timestep:

$$p(\pi_{max}|\mathbf{x}) = \prod_{t=1}^{T} max(\mathbf{y^t}) \tag{17}$$

An illustration of selecting the most probable path is shown on Figure 7.

Chiron use beam-search as its heuristic function, which requires it to maintain a list of the W most probable sequences over L up to timestep i [36]. To find the next base, W is used to compute a distribution over transitions from timestep t_i to $t_i + 1$, and selects each base from the transition with highest probability.

The dataset used by Graves et. al. [35] consisted of speech data, and their model could label audio frames with silence(blank characters) with high probability. They used this as their heuristic to split the audio samples, making their maximum input length (e.i. variable T) shorter, in turn making calculating the most probable labelling viable. To calculate the most probable labelling effectively they proposed using a modified forward-backward algorithm [35]. For nanopore data sequencing there does not exist a heuristic that can be used to split the signals into smaller sequences. Therefore the forward-backward algorithm is not viable to produce prediction on nanopore data.

Although it isn't practical to produce predictions with the forward-backward algorithm, when the length of input $\mathbf{x}$ is

large, it can still be used effectively during training. In the next section we will explain the forward-backward algorithm.

1) Training with the CTC Forward-Backward Algorithm: The cost function of a CTC network can be defined as:

$$J(\boldsymbol{W}) = -\sum_{(\mathbf{x},\mathbf{z})\in S} ln(p(\mathbf{z}|\mathbf{x})) \tag{18}$$

where S consists of all training pairs $(\mathbf{x}, \mathbf{z})$. This cost function is a Negative Log-Likelihood (NLL) function, which is derived from the principle of maximum likelihood estimation. This is also called maximum likelihood training; *"The aim of maximum likelihood training is to simultaneously maximise the log probabilities of all the correct classifications in the training set"*(Graves et al. 2006) [35]

During training CTC places some restrictions on the paths. First it removes letters that are not in the target alphabet. On Figure 8 the letter G has been removed because it is not in the target sequence TCAA. Secondly it restricts the order of the sequence according to target sequence, as also can be seen on Figure 8. Finally blank characters are added between each character of the target sequence, and as the first and last state. The arrows show the available paths through graph defined by the restrictions. The sum of all paths defined by this network describe the probability of Equation 16. Any path described in Figure 8 will always go through the characters in the reference sequence in the correct order. Three other rules are needed to handle the blank character:

1) The first state can either be the first blank or the first base.
2) The last state can either be the last blank or the last base.
3) Blanks can be skipped if the from state and to state types are not equal.

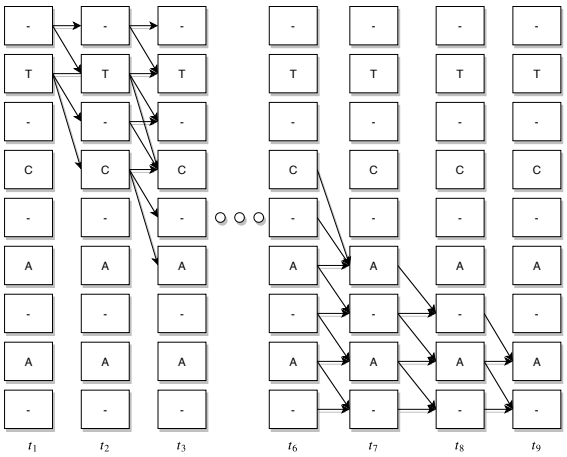

Fig. 8: A CTC..., with reference sequence TCAA.

On Figure 8 it is possible to transition directly from T to C and from C to A, and it is not possible to go directly from the first A to the second A, as is required by the third rule.

With the possible paths defined, the forward-backward algorithm can effectively calculate Equation 16.

For a target labelling sequence $\mathbf{l}$, let us define the forward variable $\alpha_t(s)$ to be the sum of all sub-paths that start at time 1 and end on character s at time t. At timestep 1 the sub-path is allowed to start on a blank or the first character of l. Formally the $\alpha_1(s)$ is:

$$\begin{aligned}\alpha_1(1) &= y^1_b \\ \alpha_1(2) &= y^1_{\mathbf{l}_1} \\ \alpha_1(s) &= 0, \forall s > 2\end{aligned}$$

At all timesteps $t > 1$ $\alpha_t(s)$ is:

$$\alpha_t(s) = \begin{cases} \bar{\alpha}_t(s) y^t_{\mathbf{l}'_s} & \text{if } \mathbf{l}'_s = b \text{ or } \mathbf{l}'_{s-2} = \mathbf{l}'_s \\ (\bar{\alpha}_t(s) + \alpha_{t-1}(s-2)) y^t_{\mathbf{l}'_s} & \text{otherwise} \end{cases} \tag{19}$$

where

$$\bar{\alpha}_t(s) := \alpha_{t-1}(s) + \alpha_{t-1}(s-1) \tag{20}$$

From this definition we can see that $\alpha_t(s)$ is recursively defined for all $t > 1$. The backward variable $\beta_t(s)$ is defined similarly:

$$\begin{aligned}\beta_T(|\mathbf{l}'|) &= y^T_b \\ \beta_T(|\mathbf{l}'| - 1) &= y^T_{\mathbf{l}_{|\mathbf{l}|}} \\ \beta_T(s) &= 0, \forall s < |\mathbf{l}'| - 1\end{aligned}$$

$$\beta_t(s) = \begin{cases} \bar{\beta}_t(s) y^t_{\mathbf{l}'_s} & \text{if } \mathbf{l}'_s = b \text{ or } \mathbf{l}'_{s+2} = \mathbf{l}'_s \\ (\bar{\beta}_t(s) + \beta_{t+1}(s+2)) y^t_{\mathbf{l}'_s} & \text{otherwise} \end{cases} \tag{21}$$

where

$$\beta_t(s) := \beta_{t+1}(s) + \beta_{t+1}(s+1) \tag{22}$$

The product of the forward and backward variables at a given character s and time t is the probability of all paths that go through character s at time t. Due to this for any t we can sum over characters s to get all paths:

$$p(\mathbf{l}|\mathbf{x}) = \sum_{s=1}^{|\mathbf{l}'|} \frac{\alpha_t(s)\beta_t(s)}{y^t_{\mathbf{l}'_s}} \tag{23}$$

We can use Equation 23 in the cost function in Equation 18 to get an efficient way of computing the cost function.

C. Nanopore Basecalling

Basecalling is a sequence-to-sequence mapping problem [2, p.44]. Nanopore basecalling can either be performed on raw signal data, or on event-based data, where sections of reads referering to the same k-mers are split into "events" each representing a potential k-mer. As described in **[8, sec. 2.1] strands containing bases are pulled through the nanopore to obtain reads, however reads are not always accurate. A DNA strand can get stuck in the nanopore resulting in too many measurements of the same k-mer. A single base can also be skipped due to the DNA strand moving too quickly through the

nanopore. Finally an interference in the electrical current running through the nanopore can result in the measured current not properly representing the k-mer. All of these errors are categorized as random errors as they occur at random when sequencing DNA. These error can result in a incorrect base label when basecalling, however they can usually be eliminated by making use of the consensus sequence. The consensus sequence is a sequence of the most probable base at each position, obtained by comparing several basecalled sequences. The quality of a basecaller is measured by looking at its read_id and assembly_id. Read_id is the basecall of a single read compared against a reference, whereas the assembly_id is the quality of a consensus sequence measured against a reference.

There are currently a lot of different basecallers being used in the industry. Some of them, such as Chiron[36] or Scrappie[37], have an open licence while others such as Metrichor[38] are proprietary software. Basecallers usually make use of RNNs, **[8, app. A] taken from Mahdieh Abbaszadegan's master thesis on "An Encoder-Decoder Based basecaller for nanopore sequencing" [2, p.48-49]. The table has been modified to not include the library and language for each basecaller, since this was not deemed necessary for this analysis. Additionally information regarding the Guppy architecture has been updated based on our research in this report (see **[8, Ch. 5]). As can be seen in the table, the median read_id is relatively low for all basecallers, with none of them being above 90% this is partly due to the occurrence of errors when sequencing. However systematic errors can also occur. These are errors not usually mitigated by the consensus. These errors can occur when a basecaller is not properly trained to identify a given set of bases in a sequence, or if the architecture does not properly model data. Seeing as the assembly_ids are all above the 99% mark, an interesting area to be explored more, could be raising the median read_id.

References

[1] Oxford Nanopore Technologies, "Oxford nanopore technologies website." https://nanoporetech.com/, 2020. Accessed: 2020-09-03.

[2] M. Abbaszadegan, *An Encoder-Decoder Based Basecaller for Nanopore DNA Sequencing*. YorkSpace, York University Torronto, Ontario, 2019. https://yorkspace.library.yorku.ca/xmlui/handle/10315/36268.

[3] Oxford Nanopore Technologies, "March 2020 accuracy update from r&d." https://community.nanoporetech.com/posts/march-2020-accuracy-update?fbclid=IwAR2rjN_i8Jtf4BG0utVJ2ZJkWAMtm_G9_1_exkygQUkI5g1Bk6lQTYqVn2U, 2020. Requires Login, Accessed: 2020-09-03.

[4] C. Brown and R. Dokos, "Accuracy update." https://community.nanoporetech.com/posts/accuracy-update, 2019. Requires Login, Accessed: 2020-09-03.

[5] Oxford Nanopore Technologies, "Flappie - flip-flop basecaller for oxford nanopore reads." https://github.com/nanoporetech/flappie, 2020. Accessed: 2020-09-03.

[6] Oxford Nanopore Technologies, "Analysis solutions for nanopore sequencing data." https://nanoporetech.com/nanopore-sequencing-data-analysis, 2020. Accessed: 2020-09-03.

[7] Oxford Nanopore Technologies, "Bonito - convolution basecaller for oxford nanopore reads." https://github.com/nanoporetech/bonito, 2020. Accessed: 2020-09-03.

[8] A. C. M. Larsen, C. A. Knudsen, and M. N. Hansen, *An analysis of the flip-flop architecture and Taiyaki training tool*. Aalborg University, january 2020.

[9] S. Kriman, S. Beliaev, B. Ginsburg, J. Huang, O. Kuchaiev, V. Lavrukhin, R. Leary, J. Li, and Y. Zhang, "Quartznet: Deep automatic speech recognition with 1d time-channel separable convolutions," *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2020. http://dx.doi.org/10.1109/icassp40776.2020.9053889.

[10] K. Hwang and W. Sung, "Character-level incremental speech recognition with recurrent neural networks," *CoRR*, vol. abs/1601.06581, 2016. http://arxiv.org/abs/1601.06581.

[11] S. Srivastava and M. S. Baptista, "Markovian language model of the dna and its information content," *Royal Society Open Science*, vol. 3, p. 150527, Jan 2016. https://arxiv.org/abs/1510.02375.

[12] W. Liang, "Segmenting dna sequence into 'words'," 2012. https://arxiv.org/abs/1202.2518.

[13] C. Seymour, "Bonito accuracy update on twitter." https://twitter.com/iiseymour/status/1230086141166211073, 2020. Accessed: 2020-21-04.

[14] C. Brown, "Clive Brown CTO update | NCM 2019." https://youtu.be/fFceCr4O284?t=1533, 2019. Accessed: 2020-09-03.

[15] National Center for Biotechnology Information, U.S. National Library of Medicine, "Blast: Basic local alignment search tool." https://blast.ncbi.nlm.nih.gov/Blast.cgi, 2020. Accessed: 2020-09-03.

[16] A. Maas, Z. Xie, D. Jurafsky, and A. Ng, "Lexicon-free conversational speech recognition with neural networks," in *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, (Denver, Colorado), pp. 345–354, Association for Computational Linguistics, May–June 2015. https://www.aclweb.org/anthology/N15-1038.

[17] B. Raj, "S18 lecture 14: Connectionist temporal classification (ctc)." https://www.youtube.com/watch?v=c86gfVGcvh4.

[18] A. Y. Hannun, A. L. Maas, D. Jurafsky, and A. Y. Ng, "First-pass large vocabulary continuous speech recognition using bi-directional recurrent dnns," 2014. https://arxiv.org/abs/1408.2873.

[19] D. Jurafsky and J. H. Martin, "Speech and language processing - chapter 3 n-gram language models." https://web.stanford.edu/~jurafsky/slp3/3.pdf, 2019. Accessed: 2020-09-05.

[20] D. Wood, J. Lu, and B. Langmead, "Improved metagenomic analysis with kraken 2." https://doi.org/10.1186/s13059-019-1891-0, 2019. Accessed: 2020-09-03.

[21] S. M. Karst, R. M. Ziels, R. H. Kirkegaard, and M. Albertsen, "Enabling high-accuracy long-read amplicon sequences using unique molecular identifiers and nanopore sequencing," *bioRxiv*, 2019. https://www.biorxiv.org/content/early/2019/05/28/645903.

[22] yourgenome.org, "What is a dna fingerprint?." https://www.yourgenome.org/facts/what-is-a-dna-fingerprint, 2016. Accessed: 2020-09-03.

[23] Oxford Nanopore Technologies, "Blitzing fast ctc beam search decoder." https://github.com/nanoporetech/fast-ctc-decode, 2020. Accessed: 2020-09-03.

[24] Open-Source, "Pyo3—rust bindings for python." https://pyo3.rs/v0.10.1/, 2020. Accessed: 2020-09-03.

[25] F. Paulin, "Character-level lstm in pytorch." https://www.kaggle.com/francescapaulin/character-level-lstm-in-pytorch, 2019. Accessed: 2020-06-07.

[26] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, pp. 3094–3100, 05 2018. https://doi.org/10.1093/bioinformatics/bty191.

[27] R. wick, "Basecalling-comparison analysis scripts." https://github.com/rrwick/Basecalling-comparison/tree/master/analysis_script, 2019. Accessed: 2020-09-03.

[28] R. C. Edgar, "Search and clustering orders of magnitude faster than BLAST," *Bioinformatics*, vol. 26, pp. 2460–2461, 08 2010. https://doi.org/10.1093/bioinformatics/btq461.

[29] R. Vaser, I. Sovic, N. Nagarajan, and M. Sikic, "Fast and accurate de novo genome assembly from long uncorrected reads," *Genome Research*, 2017. http://genome.cshlp.org/content/early/2017/01/18/gr.214270.116.abstract.
[30] Oxford Nanopore Technologies, "Medaka." https://nanoporetech.github.io/medaka/, 2019. Accessed: 2020-09-03.
[31] J. Bergstra, D. Yamins, and D. D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, p. I–115–I–123, JMLR.org, 2013. https://dl.acm.org/doi/10.5555/3042817.3042832.
[32] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization." https://arxiv.org/abs/1412.6980, 2014.
[33] Oxford Nanopore Technologies, "Taiyaki." https://github.com/nanoporetech/taiyaki, 2020. Accessed: 2020-09-03.
[34] D. Jurafsky, "Language modelling - introduction to n-grams." https://web.stanford.edu/class/cs124/lec/languagemodeling2015.pdf, 2015. Accessed: 2020-06-07.
[35] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks," in *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, (New York, NY, USA), p. 369–376, Association for Computing Machinery, 2006. https://doi.org/10.1145/1143844.1143891.
[36] H. Teng, M. D. Cao, M. B. Hall, T. Duarte, S. Wang, and L. J. M. Coin, "Chiron: translating nanopore raw signal directly into nucleotide sequence using deep learning," *GigaScience*, vol. 7, 04 2018. https://doi.org/10.1093/gigascience/giy037.
[37] Oxford Nanopore Technologies, "Scrappie." https://github.com/nanoporetech/scrappie, 2020.
[38] Oxford Nanopore Technologies, "Metrichor." https://metrichor.com/, 2020.

GLOSSARY

ASR Automatic Speech Recognition. 1

CMC Center for Microbial Communities. 5–9, 11
CNN Convolutional Neural Network. 1
CTC Connectionist Temporal Classification. 2–4, 6, 12–14

GRU Gated Recurrent Unit. 5, 6, 12

LSTM Long Short-Term Memory. 5, 6, 12

NLL Negative Log-Likelihood. 14
NLP Natural Language Processing. 11

ONT Oxford Nanopore Technologies. 1, 2, 5–10

RNN Recurrent Neural Network. 1, 5, 6, 8, 9, 11–13, 15

SMS Single Molecule Sequencing. 1

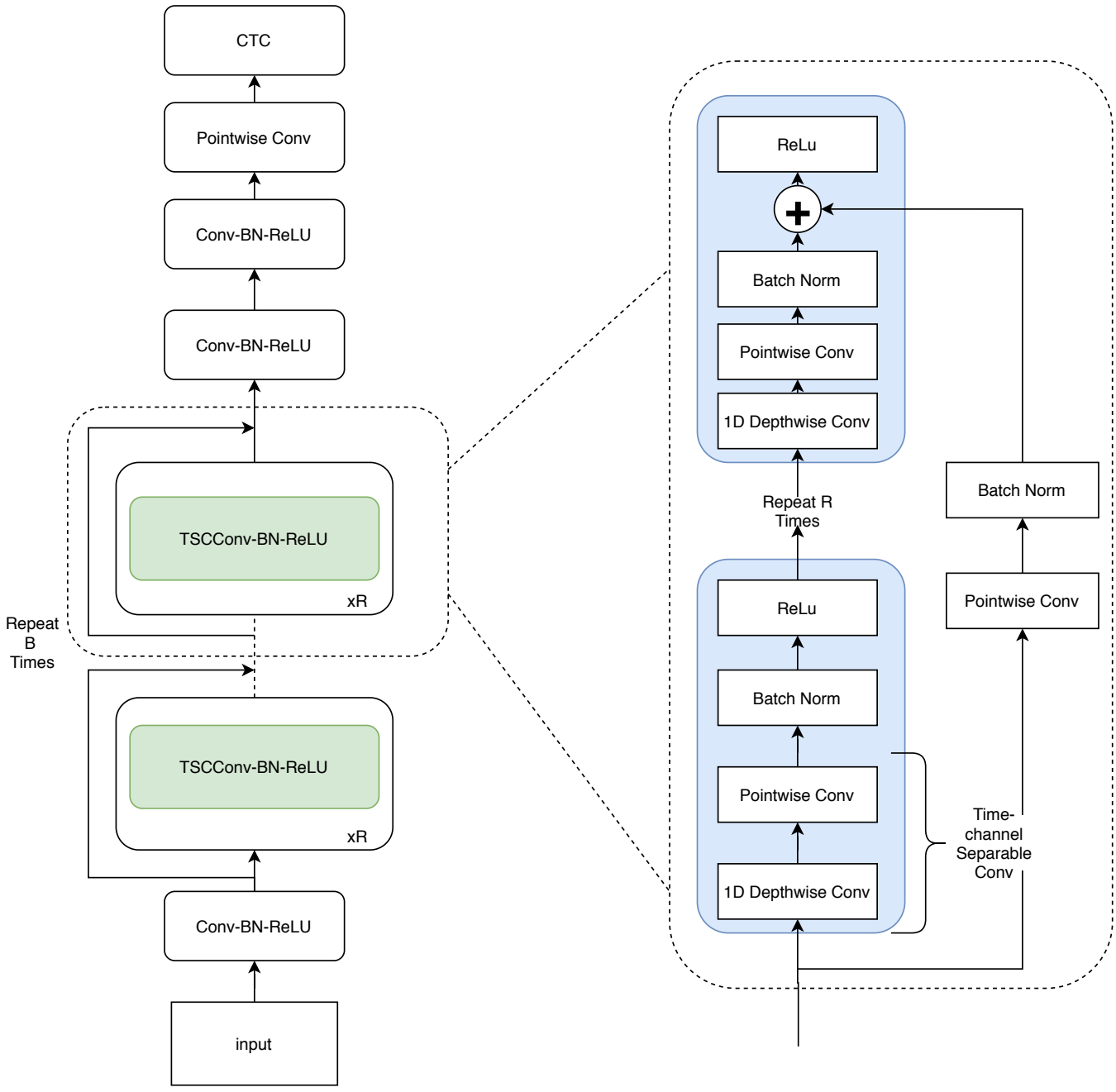

Fig. 9: The QuartzNet Architecture [9]