Summary

This summary is, to the extent possible, a non-technical description of our work and results.

DNA sequencing is the process of determining the structure of bases for organic material. It is an integral part of the process of understanding the building blocks of all life. Previously, DNA sequencing has been expensive, but new methods have emerged in recent years that, among other improvements, reduce the price significantly. One of these methods is nanopore sequencing, in which DNA is pulled through a membrane while an electrical resistance is measured. This method produces an electrical signal, which, hereafter, is translated into the letters A, C, G, and T, corresponding to the bases adenine, cytosine, guanine, and thymine, respectively. This translation process is referred to as basecalling. Since the electrical signal is complicated for humans to translate, neural networks—a type of machine learning—is used for this task. In fall 2019, we analysed the landscape of basecallers and the methods used for creating those. This work is, therefore, strongly connected to the finding therein. The basecaller Guppy is created by Oxford Nanopore Technologies, who is the creator of the nanopore sequencing device MinION, among others. In our investigation, we found that Guppy, at the time, was the most accurate basecaller—although computational requiring. Since then, however, the same company has achieved similar accuracy with another basecaller named Bonito. Bonito is interesting because its architecture has the potential to increase the speed of basecalling compared to that of Guppy. A faster basecaller can allow experiments for DNA research to be performed faster, making the aspect of time an important subject. However, as is often the case in machine intelligence, it is difficult to get a faster architecture without it negatively impacting the accuracy.

We investigate alterations of Bonito and how these affect the trade-off of speed and accuracy. Our experiments show that most alterations are positively correlated both to the accuracy as well as the processing time. Dilation is an alteration that allows computations of the network to consider larger sparse areas of the electrical signal instead of considering a small dense area. Although it is not a new method, it has not yet been tested for basecalling with Bonito. We find that using dilation allows us to increase the processing speed, while, notably, also increasing accuracy.

One way to improve the accuracy of fast networks is through knowledge distillation, which is a method to train a network, called the student, using knowledge from another existing network, called the teacher. We investigate how knowledge distillation affects networks—or basecallers—with different processing times. In our results, knowledge distillation improves the accuracy on every basecaller it is applied to. We also observe the effect of this result when we apply knowledge distillation on Bonito using itself as a teacher.

Finally, we present five basecallers of which one has higher accuracy, one has faster speed, and three has faster processing speed *and* higher accuracy all compared to Bonito.

Effects of Hyperparameter Tuning and Knowledge Distillation on the State-of-the-Art Basecaller Bonito

Kasper Dissing Bargsteen
kbargs15@student.aau.dk
Aalborg University

Jonatan Groth Frausing
jfraus14@student.aau.dk
Aalborg University

Master's Thesis
mi106f20
11th June 2020

Abstract

Basecalling, similar to many other domains of machine learning, suffers from the problem of having to accept a trade-off between prediction speed and accuracy. Bonito, based on the architecture of QuartzNet, shows similar results to Guppy, which is considered to be state of the art for basecallers. The convolutional architecture of Bonito, however, has the potential to reduce the prediction time markedly compared to the recurrent architecture of Guppy.

This work attempts to provide insight into the effect of tuning the hyperparameters available in Bonito. This effort is made with the focus of improving the speed of predictions without negative impact on the accuracy. In order to alleviate the problem of reduced accuracy in smaller networks, we apply knowledge distillation, which, in other domains, is shown to improve accuracy.

The results of our experiments suggest that dilation, combined with a reduced kernel size, can improve prediction speed and accuracy of Bonito. Additionally, we show that knowledge distillation can improve the accuracy of basecallers. Notably, the most significant improvements are observed on large basecallers. Nevertheless, the results suggest that knowledge distillation should always be applied for any size of basecaller.

Keywords: Basecalling, DNA Sequencing, Hyperparameter Tuning, Knowledge Distillation, QuartzNet, Separable Convolutions, Dilation, Grouping.

Acknowledgements

We would like to thank our supervisors Katja Hose and Thomas Dyhre Nielsen for their academic guidance throughout this project. Furthermore, we would like to thank Albertsen Lab for sharing their data with us along with their explanations of miscellaneous biology subjects and the relevance of the task at hand. Lastly, we would like to thank Mr Seymour for his profound and quick replies. We hope that our efforts regarding this contribution can aid his future work.

Contents

1 Introduction

DNA is the genetic code that underpins all organic life. The process of determining the genetic code of an organism is called *DNA sequencing*. DNA sequencing has played a vital role in understanding our genetic history [1], preventing diseases, and, more recently, helping to track and understand the current global pandemic of COVID-19 [2].

In [3], we investigate the inner workings of State-Of-the-Art (SOA) DNA sequencing techniques. This report builds on top of our findings, and narrows the scope, as will be explained. The SOA DNA sequencing techniques rely on machine learning to function, and modern DNA sequencing is thus at the intersection between biology and machine learning. The details of each field, necessary to understand this report, is thoroughly explained in Section 2 and Section 3. However, a partial summary is included in this introduction to aid the reader in understanding the aforementioned narrowing of scope.

Modern DNA sequencing consists of two primary parts: a *sequencing machine*, which scans organic material and produces a signal as output; and a *basecaller*, which is a program that translates the scanned signal into a DNA sequence. One such sequencing machine is the MinION from Oxford Nanopore Technologies, which "is the only portable, real-time device for [DNA] sequencing," according to their website and our research [3, 4]. The MinION costs $1,000 and has the size of a smartphone [4]. Other existing devices can cost up to $1,000,000 [5]. Being a real-time, i.e. fast, device of its size and price makes it interesting for new types of applications and research. The MinION device is especially interesting for low-budget research or research in the field, perhaps even off-grid in rural areas. Due to the MinION's potential for fast DNA sequencing, we will focus on it and the translation of its signals. However, the basecallers needed to translate, or interpret, the signals from the machine tend to be too slow to keep up with the MinION device, thereby increasing the overall time needed for DNA sequencing. The domain expert and CEO of Albertsen Lab, Mads Albertsen stated in an informal conversation that the basecalling speed is of high importance because it is the bottleneck of DNA sequencing using the MinION device. As argued by [6], this is particularly a problem when using the basecallers on slower, low-budget computers—which, presumably, would be used in low-budget research. Two basecallers are delivered with the MinION, Guppy and Guppy-Fast, of which only the latter is fast enough to keep up with the MinION [6]. However, the speed of Guppy-Fast comes at a high trade-off of reduced accuracy [6, 7]. In essence, there is a need for basecallers for the MinION device, which are both fast and accurate, to enable a truly portable and affordable DNA sequencing setup.

The SOA basecallers are built upon neural networks—a type of machine learning—in order to interpret the complex signals generated by the sequencing devices. In [3], we show that Guppy is the basecaller with the highest accuracy. Since then, a new basecaller, Bonito, has been published by Oxford Nanopore Technologies, which surpasses Guppy on accuracy [8, 9]. It is, however, not faster than Guppy.

Neural networks can be categorised by their architecture—the specific combination, order, and type of their components—and Bonito uses a novel architecture as related to basecalling, namely a *convolutional architecture* [9]. Guppy, along with several other competing basecallers, uses a recurrent architecture [3, 7]. Convolutional architectures, which we will expound upon in Section 3.2.1, have been shown to be faster than recurrent architectures [10, 11, 12]. Since

Bonito is slower than Guppy, it suggests that the novel architecture has not yet reached its full potential. We decide to investigate the potential of Bonito in terms of both speed and accuracy.

Bonito uses a specific convolutional architecture called QuartzNet [9], originally created for automatic speech recognition [13], i.e. the task of transcribing spoken words. Both basecalling and automatic speech recognition involve mapping a sequence of inputs to a sequence of outputs, known as a sequence-to-sequence task, which explains why the same architecture can be used for solving seemingly unrelated problems.

For any given neural network, a number of settings, collectively called HyperParameters (HPs), can be changed or tuned, which can drastically affect its results both in terms of speed and accuracy [14]. HPs can be divided into two groups—namely *model HPs* and *algorithm HPs*. Model HPs are the configurable settings that define the architecture. Algorithm HPs are the parameters specific to how the network learns, which will be further explained in Section 3. Throughout the report, we will use model HPs and HPs interchangeably, whereas we will explicitly state when we are referring to algorithm HPs.

Upon inspection of the HPs used in Bonito, we find them to be nearly identical to those used in QuartzNet. While the tasks of automatic speech recognition and basecalling are similar, it seems unlikely that the same HPs are optimal for both tasks. This observation leads us to believe that the SOA for basecalling can be advanced even further by tuning the HPs in Bonito. Unable to find results of HP-tuning on Bonito, we reach out to its creator, Christopher Seymour, who is the primary basecaller-developer for Oxford Nanopore Technologies. Mr Seymour provides a list of HPs he has experimented with along with the values used for each one. He concludes his message stating that while he is able to increase the speed of Bonito, it always comes at the cost of reduced accuracy.

Having Mr Seymour's initial results in mind, we search for a method, which could counteract the reduction in accuracy he observed for configurations of HPs that yield increases in speed. We find several methods, but Knowledge Distillation (KD) stands out because it explicitly targets fast neural networks [15]. We will further explain KD in Section 3.3.2. Using KD for basecalling is, to the best of our knowledge, unexplored, and, therefore, interesting on its own.

1.1 Problem Statement

On the premise that the speed and accuracy trade-off is of particular importance for basecalling, we decide to investigate the following,

1. How and to what extent can HP-tuning improve the accuracy and/or speed of Bonito?
2. To what degree can KD, combined with the above HP-tuning, positively affect the speed and accuracy trade-off of Bonito?

In the following sections, we will explain the biology and machine learning that forms the foundation of our work.

2 Biology

Parts of this section were originally presented in our previous work, [3], which led to this project. Modifications for clarification have been made. We will, in this section, explain the required biological subjects related to basecalling. In the following pages, we provide an overview of what DNA and DNA sequencing are and a comparison of MinION and its alternatives.

2.1 DNA

DNA encodes the genetic information of an organism as a series of molecules called nucleotides. Each nucleotide contains a *nucleobase*, of which there are four: adenine (A), cytosine (C), guanine (G), and thymine (T). Nucleobases are also simply called *bases*. The specific order of these bases encodes the genetic information itself. DNA is stored in a double helix structure such that the genetic information is contained twice. The two *strands* in the structure are combined with pairs of bases, where A pairs with T, and G pairs with C. Observing a base in one strand, consequently, enables the deduction of the corresponding base in the other strand [16]. Figure 1 shows the characteristic double helix structure of the DNA. The process of understanding DNA has been of great importance for the human species and has enabled us, amongst many other things, to track the history of our evolution [1].

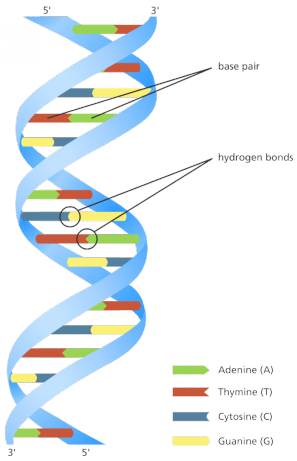


Figure 1: An illustration to show the double helix structure of DNA.
Image credit: Genome Research Limited

2.2 Sequencing

The process of determining the structure of a DNA sample is called DNA sequencing, which was discovered in the 1970s [17]. At a high level of abstraction, modern DNA sequencing consists of two primary steps. First, a sequencing device scans a DNA sample, producing a signal as

output. The signal is, subsequently, processed by basecaller to extract the DNA structure, i.e. the ordering of the bases A, C, G, T.

Devices used for DNA sequencing are evaluated on several parameters, which includes the price of the machine, speed of processing, and its accuracy. Another factor is the length of the continuous fragments of DNA that can be sequenced. A single DNA fragment is called a *read* and its length is referred to as *read length*.

After the signal of each read is processed by a basecaller, a possible next step is *genome assembly* where read length becomes relevant. It is the process of combining the individual reads into one contiguous DNA string. Neighbouring reads have common sub-sequences, which are used in a domino-like fashion to combine them. Consequently, longer reads with longer common sub-sequences improve the quality of the final assembled string [18]. Genome assembly, however, is not in the scope of this project. Nonetheless, the use of genome assembly makes the long read length of the MinION device, which we shall soon describe, an attractive property.

2.2.1 Modern Sequencing Tools

The price required to perform DNA sequencing has been falling steeply for the past decades, a trend unlikely to stop [19]. The *Illumina dye sequencing* system [20] is a sequencing method that uses specially dyed molecules in order to, optically, determine the DNA sequence. This process is time-consuming, and the effectiveness of the chemical process degrades over time, so the maximum read length is in the order of hundreds of bases [21]. As a comparison, E. Coli, a bacteria, has a genome length of 4,600,000 bases [22].

An alternative to the Illumina devices is the *Oxford Nanopore Technologies* (ONT) [23] devices, such as the MinION [24], which are small and cheap devices that enable very long continuous reads of DNA sequences. The two strands of DNA is divided and pulled through an electrically-resistant polymer membrane called a *nanopore* (Figure 2). Since different bases have different molecular structures, they generate different electrical signals when passed through the membrane. This electrical signal is captured by a very sensitive sensor and output as raw data, which can be interpreted by a basecaller in order to recover the original genetic sequence [25].

Table 1 compares two Illumina systems to the MinION device and another nanopore sequencer called PromethION on sequencing speed, read length, price, and error-rate. The MinION device is by far the cheapest and can process the longest reads. It is twice as fast as the Illumina MiSeq, however a lot slower than the two others. The last parameter to consider is the error-rate, where the Illumina systems perform up to 100 times better than the two Nanopore devices [5, 26, 27, 28]. The error-rate is measured on the basecalled signal and consequently depends on the quality of the basecaller. While the price of the MinION is attractive, its error-rate is less so. Improvements to the accuracy of basecalling for MinION is therefore relevant, which is one of the key elements of this project.

2.3 Basecalling

The process of translating the scanned signal from a sequencing device to bases is called basecalling [7], and it is an essential part of DNA sequencing. Basecalling is a *sequence-to-sequence* mapping problem, where the input is a representation of a signal from a sequencing device, and the output is the corresponding sequence of bases. Figure 3 shows a snippet of a signal

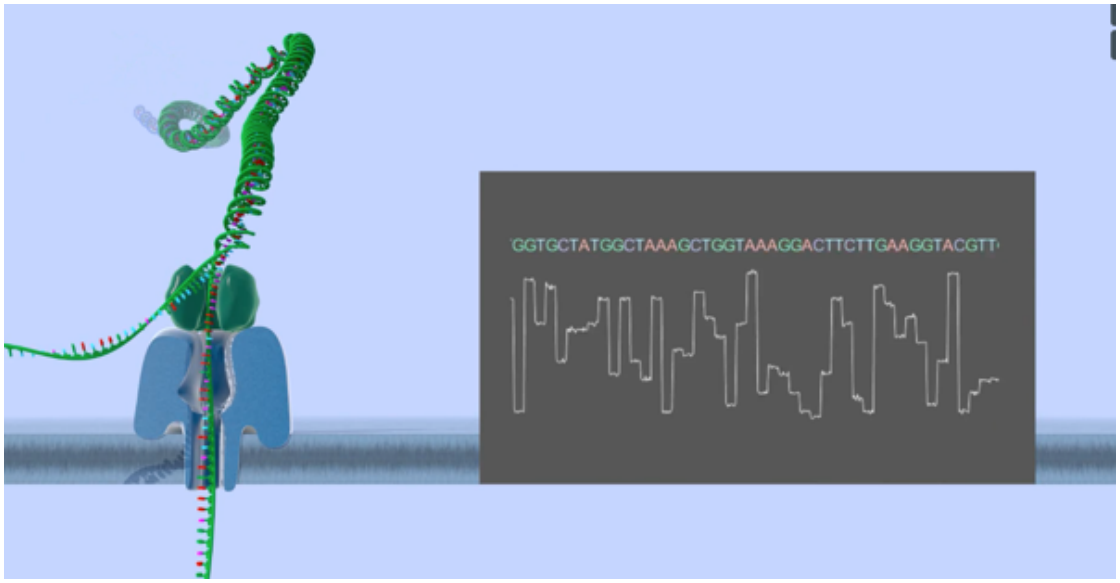


Figure 2: A strand of DNA passing through a *nanopore*, from [23]

Device	**Read per day (Gbp)**	**Read length (bp)**	**Price of machine** ($)	**Error-rate**
Illumina NovaSeq6000	3,600	300	985,000	<0.1
Illumina MiSeq	7	600	99,000	<0.1
Nanopore MinION	15	25,000	1,000	5-10
Nanopore PromethION	10,000	25,000	285,000	5-10

Table 1: Comparison of different sequencing devices [5, 26, 27, 28].

produced by a nanopore along with a sequence of bases, as inferred by a basecaller. The basecallers for different sequencing devices differ in their details. As stated in Section 1, the focus of this project is on basecallers for the MinION device. Hence, any subsequent uses of the word *basecaller* will refer to a basecaller for the MinION device.

Figure 3: A snippet of raw signal from an ONT device which corresponds to the DNA sequence *CCGTCCCCCCTTCGCAGTAACACCAAGTACAGGAATATTAACCT*

Basecalling is a non-trivial task due to the complexity of the sequence produced by the MinION device [29]. There are approximately ten times more electric signal measurements than there are bases because the rate at which the sequencing device samples the electric signal is much higher than the rate at which the DNA strand moves through the nanopore. Additionally, the movement speed and the electrical resistance is not only affected by the DNA inside the nanopore, but also by the sequence before and after [29].

2.3.1 Accuracy

There are a number of basecallers that attempt to improve the accuracy of their predictions—i.e. the DNA sequence that they output—by utilising different types of artificial intelligence algorithms [7]. The quality of a basecaller is determined by how well the produced base sequence matches the actual DNA sequence. The accuracy measures used are *read identity* and *consensus identity*. Read identity measure how well one read can be aligned to the expected sequence. The alignment is described using a *CIGAR* string, which is a string over the four characters =(match), *D*(deletion), *I*(insertion), and *X*(mismatch) [30]. Table 2 shows an example of an alignment. Read identity is calculated by dividing the number of matches by the alignment length (length of the CIGAR string).

Expected	`ACGTG  AGGTAT`
CIGAR	`==D=XII======`
Prediction	`AC TATTAGGTAT`

Table 2: Alignment of the reference sequence **ACGTGAGGTAT** and a prediction **ACTATTAGGTAT**

Consensus is used when multiple strands of the same DNA has been basecalled, which is common in specific research settings. The basecalled sequences are combined into a single sequence with majority-rule, of which the consensus identity is then calculated in the same way as read identity. The combination of multiple basecalled sequences into one typically has the benefit of averaging out errors of each basecall. However, this is not always the case. Consider the two basecallers A and B. A has a read identity of 90%, and B reaches 98%. B seems to be the best of the two options. However, if we let the errors in A be completely random, then A's consensus identity will approach 100% as more strands are used. B, on the other hand, could have systematic errors, meaning that it is the same 2% it predicts wrong on every read. A would thereby outperform B since B would not be improved by using consensus. This example shows that both read identity and consensus identity are valid accuracy measures for basecallers. For comparison, the SOA basecallers have read identities between 90% and 95% and consensus identities above 99% [7].

In Section 1, we describe our intention of, among other things, enable field testing by creating a fast basecaller for the MinION device. For field testing, read identity is often more relevant than consensus, because it is hard to gather sufficient amounts of the similar DNA needed for consensus [7, 31]. Hence, we decide to use read identity for the remainder of our research.

In this section, we have covered the essential subjects related to basecalling. In the following sections, we will explain the machine learning specific elements that are required for the scope of this project.

3 Machine Learning

In this section, we will introduce the fundamentals of the machine learning used in the remainder of the report.

3.1 General

The broad category of machine learning used in basecalling is called *supervised learning*. In supervised learning we start out with some *training data*, which is a set consisting of pairs of inputs, $\mathbb{X}$, and targets/labels $\mathbb{Y}$: $\{(\mathbf{x}^1, y^1), \ldots, (\mathbf{x}^N, y^N)\}$, where N is the number of pairs, and $(\mathbf{x}^n, y^n)$ is the n^{th} pair of input and target in the training set. The goal is to find a function which maps each input to its corresponding target value. *Finding* such a function is the *learning* part of machine learning. A common approach to do so is to train a neural network, also referred to as a *model*. To understand the process of building and training a neural network, we describe its primary components and processes in isolation and lastly combine them into a whole.

3.1.1 Perceptron

A *perceptron*, as seen in Figure 4, is the basic unit of a artificial neural network. It takes an input vector, of some size P, from $\mathbb{X}$, $\mathbf{x} = \{x_1, ..., x_P\}$, and uses an internal set of *weights*, $\mathbf{w} = \{w_1, ..., w_P\}$, to compute a single output, denoted z.

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_P x_P = \mathbf{w}\mathbf{x}^T \tag{1}$$

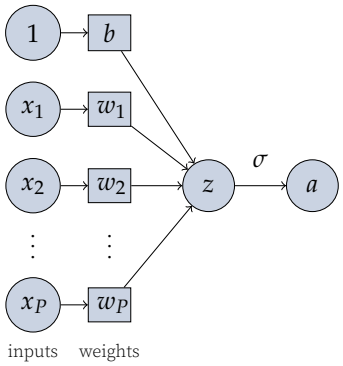


Figure 4: A perceptron with bias b and p inputs and weights, followed by an activation function σ.

It is common to introduce a bias-term, b, that is added, which enables biasing the result in some direction: $z = \mathbf{w}\mathbf{x} + b$. Conveniently, the bias can also be represented by including b in the set of weights and the constant 1 in the inputs, as seen in Equation (2), where $\frown$ means concatenation.

$$\begin{aligned} \mathbf{w} &:= [b] \frown \mathbf{w}, \\ \mathbf{x} &:= [1] \frown \mathbf{x} \end{aligned} \tag{2}$$

Activation function Since a perceptron is a linear function it cannot accurately approximate non-linear functions. An *activation function*, denoted σ, is used to rectify this shortcoming. The activation function is applied on the output, z, of the perceptron and the result is called the *activation*, a. Note the distinction between an activation, a value, and an activation function, which is a function. A common example of an activation function is *ReLU*, as seen in Equation (3).

$$a = ReLU(z) = \begin{cases} z, & \text{if } z > 0 \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

Cost function A *cost function* (also known as a loss function), denoted with a C, is used to measure how poorly the predictions $\hat{y}$, i.e. the output from the network, match the target values from the training data, y. In Figure 4, $\hat{y}$ is equal to a. One common cost function is Mean Squared Error (MSE), as seen in Equation (4), which operates on all N pairs of training data.

$$C_{MSE} = \frac{1}{N} \sum_{n=1}^{N} (\hat{y}^n - y^n)^2 \tag{4}$$

The cost function essentially returns a metric of how far away we are from finding a function that maps the training data correctly. We refer to this metric as the loss, which is used when training the network, as we will explain in the following section.

Training The overarching idea of training a neural network is to change the weights until the network maps a desired function. Interestingly, this is equivalent to minimising the loss from the cost function. One universal approach is to use *gradient descent*, in which the *gradients* of the weights to update the network. The gradients describe the directions in which to change the weights to *maximize* the cost function. Consequently, the gradients are subtracted from the weights. To calculate the gradients, we find the *partial derivatives* of C with respect to each weight—that is, how a change in each weight affects the result of C. Since changes in a weight, w_i, are mediated through z and then a before affecting C, we use the chain rule to calculate the partial derivative as seen in Equation (5). The effect of changes in w_i on C is dependent on how w_i affects z, z affects a, and, finally, how a affects C.

$$\frac{\partial C}{\partial w_i} = \frac{\partial C}{\partial a} \frac{\partial a}{\partial z} \frac{\partial z}{\partial w_i} \tag{5}$$

The gradients $\nabla C = \{\frac{\partial C}{\partial w_1}, \ldots, \frac{\partial C}{\partial w_P}\}$, calculated for all P weights, are then used to update the weights. Knowing which direction to update the weights does, however, not provide any information on how far the gradient continues. We use a *learning rate*, denoted as $\alpha \in]0, 1]$, to scale the gradients and thereby controlling, how much the weights should be changed. Equation (6) shows how the set of weights, $\mathbf{w}$, are updated.

$$\mathbf{w} := \mathbf{w} - \alpha \nabla C \tag{6}$$

3.1.2 Combining Perceptrons into Networks

A single perceptron combined with an activation function, as seen in Figure 4, has a low expressive power, which limits its ability to represent complex functions. In order to increase the expressiveness, multiple perceptrons can be combined into networks with layers, where the output of a given layer becomes the input to the subsequent layer. Each layer is a group of perceptrons and activation functions. In Figure 5, a simple network with three layers is seen. Each layer has a single perceptron and activation function, which is merged into a single circle to simplify the figure. The arrows each represent a weight. While conceptually identical, the first and last layer of any network differ semantically and are therefore called the *input layer* and *output layer*, respectively. Any intermediate layers are called *hidden layers*. In similar fashion, the input to the first layer is often denoted as $\boldsymbol{x}$, but it is conceptually identical to the inputs of the subsequent layers, denoted $\boldsymbol{a}$. To simplify the equations that follow, we use $\boldsymbol{a}$ for all layers. It is also worth noting that the connection between each layer is a function, which makes the whole network a composition of functions. In Section 3.1.1, we describe how to train a single perceptron, now we extend this concept to a network.

Training a Network As described, Figure 5 shows an elementary network with three layers. In this network, the activation a^l for a given layer l is computed based on the activation of the previous layer: $a^l = \sigma(z^l)$, where $z^l = z(\mathbf{w}^l a^{l-1})$ and $a^{l-1})$. The calculation for the gradient of the weight $\mathbf{w}^L$ in layer L is computed the same way as in Equation (5).

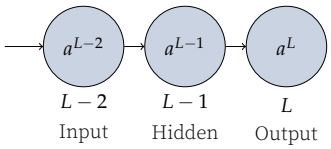


Figure 5: A simple neural network with three layers and only one activation per layer. The arrows represent weights.

We can also update the weight, $\mathbf{w}^{L-1}$, of layer $L-1$, by using the chain rule and the partial derivative of the activation for the layer, a^{L-1}, as computed in Equation (7). By calculating the derivatives backwards, i.e. from the output to the input layer, and storing the intermediate values, a whole network can be updated. The process of calculating the derivatives backwards is called *backpropagation* [32].

$$\frac{\partial C}{\partial a^{L-1}} = \frac{\partial z^L}{\partial a^{L-1}} \frac{\partial a^L}{\partial z^L} \frac{\partial C}{\partial a^L} \tag{7}$$

Equation (7) is a simplified example since most networks have multiple perceptrons per layer. One example of such is shown in Figure 6 in which each vertical group of perceptrons represents a layer. We will now show how the backpropagation algorithm is generalised to work for any network.

We introduce subscripting to indicate which perceptron is being referred to in a given layer l. The activation for perceptron j of layer l is thus denoted as a^l_j. Similarly, we define w^l_{kj} as

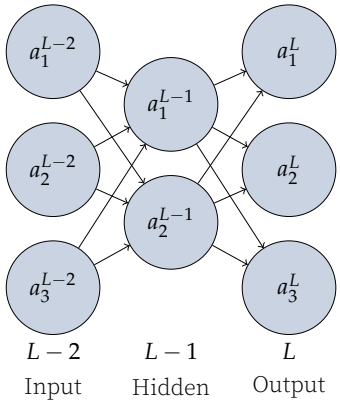


Figure 6: A three-layered network with three inputs, two hidden states, and three outputs

being the weight applied to a_k^{l-1} when calculating a_j^l. We can thereby define:

$$\begin{aligned} z_j^{(l)} &= \sum_k w_{kj}^{(l)} a_k^{l-1} \\ a_j^{(l)} &= \sigma(z_j^l) \end{aligned} \tag{8}$$

Note that a single activation function, σ, is used for all perceptrons to simplify the examples. In real artificial neural networks, it is common to use different activation functions for each layer or perceptron.

Using Equation (8) we can calculate $\frac{\partial C}{\partial a_k^{l-1}}$ by summing over the results of applying the chain rule:

$$\frac{\partial C}{\partial a_k^{l-1}} = \sum_{j=1}^{n_l} \frac{\partial z_j^l}{\partial a_k^{l-1}} \frac{\partial a_j^l}{\partial z_j^l} \frac{\partial C}{\partial a_j^l}, \tag{9}$$

where n_l is the number of neurons in layer l.

With the partial derivative calculated for a given perceptron a_k^{l-1}, its weights can be updated as in Equation (5).

In this section, we covered the essentials of supervised machine learning, including the components and methods needed to build and train a neural network. Although neural networks can be built solely using layers of perceptrons, it is not the only type of layer available.

3.2 Layer Types

In this section, we will delineate the three layer-types used in Bonito, namely convolutional, residual, and batch normalisation.

3.2.1 Convolutions

Convolutional layers have initially been designed to perform well on images, but have recently gained a lot of popularity even in temporal classification—that is, classification in which time

is considered—and similar tasks for which recurrent layers are classically purposed [33, 34]. The convolutional operation, used in convolutional layers, slides a window of *trainable* weights, called a *kernel*, over the input to get an output, known as a *feature map*. In a convolutional context, an input is called a *channel*. Figure 7 illustrates how each of the concepts relate to each other. Convolutions are merely a specific combination of perceptrons, meaning that weights are trained identically to those previously explained in Section 3.1. In the examples that follow, only 1D channels and kernels are used. However, convolutions can be of higher dimensions, e.g. in image classification, where a 2D kernel is generally used. In general, the kernel is of smaller size, i.e. has fewer values, than the channel it operates on. A *stride* determines how much the kernel moves as it slides across the input. Figure 7 shows an example of a convolutional operation, denoted $\star$, of stride 1 with a kernel consisting of the weights w_1 and w_2. Note how a *single* value is produced each time the kernel is applied. Using a kernel size or stride larger than one causes the feature map to be smaller than the input. If this effect is undesired, a common technique is to pad the sides of the input with a value, for example, zero (known as *zero-padding*).

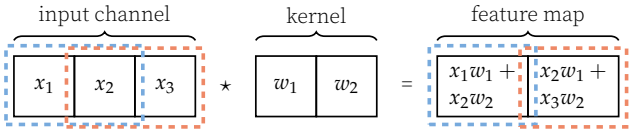


Figure 7: A 1D-convolution, using stride of one, with a single input channel and kernel, producing a single feature map as output.

The computation for each value, f, with index i of a feature map is formally defined in Equation (10), where x is the input channel, w is the weights of the kernel, J the kernel size, and s is the stride.

$$f_i = \sum_{j=1}^{J} x_{j+is-s} w_j, \tag{10}$$

A principal utility of convolutions is their ability to extract new features from the input. In Figure 8, an input signal measured over time is shown, which has a repeating pattern of relative changes inside box A and B. The actual values for the signal are $[3, 2, 3, 1, 3]$ and $[10, 9, 10, 8, 10]$ inside boxes A and B, respectively. It is difficult to recognise the pattern based on the values alone. However, if we apply a convolution with a $[-1, 1]$ kernel and stride 1, which will calculate the amount of positive change between two signal values one time-step apart, the pattern suddenly emerges. The feature map, i.e. the output after applying the convolution operation, is $[-1, 1, -2, 2]$ for both A and B. The pattern mentioned earlier now emerges in both the actual values and the visual representation seen in the bar-chart superimposed on Figure 8. Note that the left and right y-axis are to be used, respectively, for the signal and feature map values.

As mentioned previously, a kernel produces a *single* value in the feature map each time it is applied. This property still holds when there are multiple input channels. The number of input channels is called the *depth*. A 1D kernel of size J applied on an input of size $M \times D$, where D is the depth, will therefore actually have size $J \times D$—the same depth as the input. Thus, the output of convolving a single kernel over any input will always produce an output of depth one.

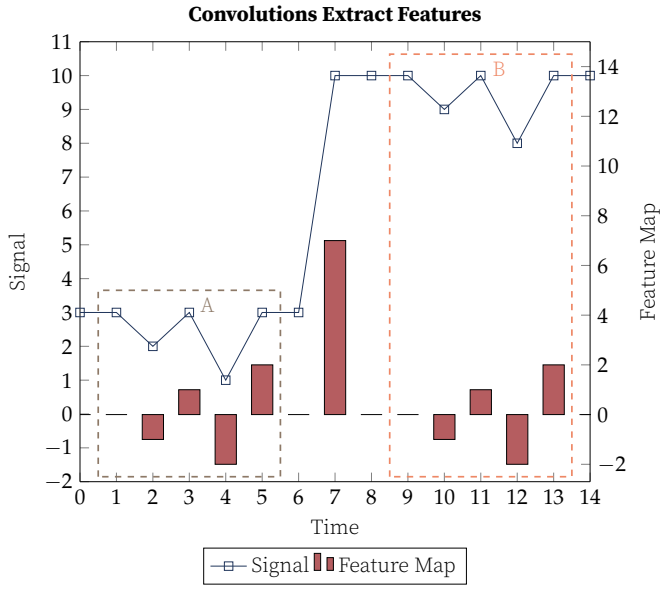


Figure 8: Convolutions as feature extraction. The signal uses the *left* y-axis and the feature map uses *right* y-axis. Both of which are related to time via the x-axis. The signal contains a repeating pattern of relative change inside box A and B. The pattern is clarified through the feature map, shown on the bar-chart.

Figure 9 illustrates a kernel with size and depth two applied on two input channels. Vertical spacing is used on the input channels to indicate that it is *two distinct* channels. By the same logic, no vertical spacing is used for the *single* kernel. We will use spacing in the same fashion for the remaining convolution figures.

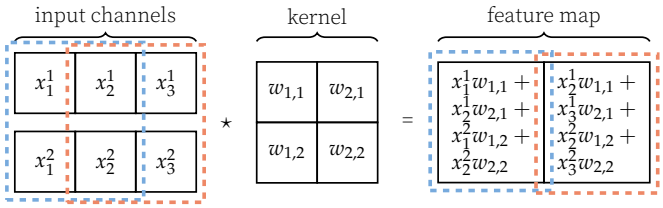


Figure 9: A 1D kernel applied with a stride of one on two channels, which produces a feature map of depth one. x_v^d is the vth input value in the dth channel.

We extend the formalisation from Equation (10) in Equation (11) by including an additional summation over D, the depth of the kernel and the input channels, as well as appropriate indexing for both inputs, $\mathbf{x}$, and weights, $\mathbf{w}$.

$$f_i = \sum_{d=1}^{D} \sum_{j=1}^{J} x_{j+is-s}^d w_{j,d} \tag{11}$$

A single kernel can extract a single type of feature from the data. Often, it is of interest to

extract multiple features, which can be achieved by increasing the number of distinct kernels used. Using K kernels will generate an output of K feature maps—i.e. K outputs of depth one, one for each kernel. In neural networks with multiple convolutional layers in sequence, the feature maps, or outputs, of one layer becomes the input channels for the subsequent layer. *Higher level features* can thus be extracted from features generated by the previous layer. While higher-level features can be difficult to understand through the example in Figure 8, it can readily be understood for images, where features such as lines and curves can be combined into features describing more complex shapes, such as circles and squares.

With an understanding of convolutions in general, we will proceed onto explaining certain extensions to the convolutional operator. Recall that one of our primary intentions is to investigate the effect of tuning different HPs in the basecaller Bonito. The extensions we describe are dilation, grouping, and separability, which are HPs to the convolutional operation. Of the three, only separability has been utilised in Bonito. The subsequent explanations are ordered to help the reader grasp the ideas—as opposed to an ordered by the novelty related to Bonito.

Dilation The first extension is *dilation*. As previously explained, convolutions extract features, where the kernel size determines the region in which a single feature can be extracted. This fact means that if the underlying feature, or pattern, covers a large region, then the kernel has to be large. A larger kernel will have more trainable weights, which will have a negative impact on prediction speed because more computations are needed. [35] shows a different approach, namely *dilated kernels*. Dilation increases the kernel size without adding more parameters. Instead, it spreads the existing parameters out. Dilation can be implemented in several ways, but it is conceptually identical to adding zeroes between existing parameters. Figure 10 shows the effect of dilating a kernel by two, $\mathfrak{D}_2$, visualised using zeros.

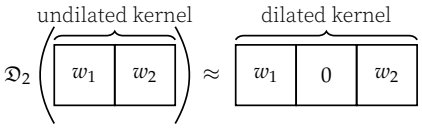


Figure 10: A kernel of size two dilated by two, $\mathfrak{D}_2$, resulting in an effective region of size three. As shown, dilation is conceptually equivalent to interspersing zeros between the weights.

A kernel of size J, where J is the amount of trainable parameters, with a dilation of δ, i.e. $\mathfrak{D}_\delta$, gets the effective region-size of $J + (J-1) * (\delta - 1)$. Dilating by one, therefore, leaves the kernel unaltered.

Grouping The second extension to convolutions is *grouping*, which was first introduced in [36]. When grouping convolutions, each distinct kernel uses only a subset of the input channels, which reduces the computation time needed. More precisely, with g groups, each kernel uses $\frac{1}{g}$ parts of the input channels and the depth of the kernel is equally reduced. Grouping by one is thus equivalent to a regular convolution. While grouping reduces the kernel depth, and thus the number of weights, it has been shown to, sometimes, improve the accuracy of a convolution [37].

The following examples used to explain grouping are simplified by using input channels and

kernels of the same size, which eliminates the need to think about stride—the effect of which is unchanged.

Figure 11 shows a regular convolutional operation with two input channels, two kernels, and no grouping.

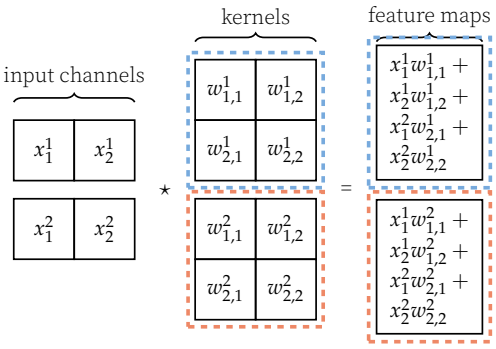


Figure 11: A normal 1D convolution with two input channels, two kernels and two feature maps.

To accommodate the use of multiple kernels, we extend Equation (11) in Equation (12). It is very similar to Equation (11), because the only difference is that we introduce a superscript, m, to reference which kernel is used. m also refers to the feature map generated.

$$f_i^m = \sum_{d=1}^{D} \sum_{j=1}^{J} x_{j+is-s}^d w_{j,d}^m \tag{12}$$

Figure 12 shows a *grouped* convolution with two groups on the same input as used in Figure 11.

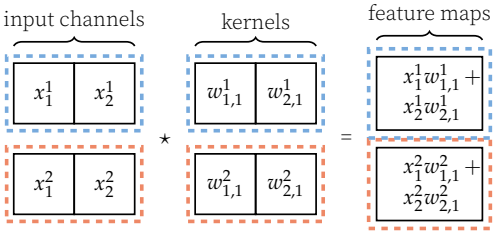


Figure 12: A convolution with *two groups*, where each kernel is applied on half of the input channels.

It would not be possible to use, e.g., three groups in Figure 12 because each kernel would have to use a non-integer subset of the input channels, which is undefined. In fact, the number of groups g has to be a factor of both the number of channels *and* kernels.

Naturally, several grouped convolutions can be used together in a neural network. In this scenario, one downside of using grouping is that each subsequent layer will see a smaller fraction of the original input. To amend this shortcoming, ShuffleNet [38], suggests reordering the output channels for every layer. While it technically is a deterministic reordering, it is called

shuffling.

Separability The third extension to convolutions is *separability*, first introduced in [39]. A convolution can be separated if all of its weights have common factors. This naturally reduces the expressiveness of the convolution, given that not all combinations of weights share common factors. In practice, the separated kernels are used directly to reduce the number of weights in a network, but we will show how they can still represent a normal convolution. In Figure 13 three kernels of size two with common factors are shown. a and b are factors for all the kernels, and r, s, and t are factors for one kernel each.

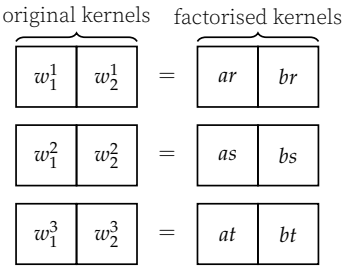


Figure 13: Separable convolution. Kernels that share common factors can be separated.

In Figure 14 a *regular convolution*—i.e. one with no extensions—is shown using the factorised kernels from Figure 13. Each of the three kernels are applied on the single input channel, thereby producing three feature maps.

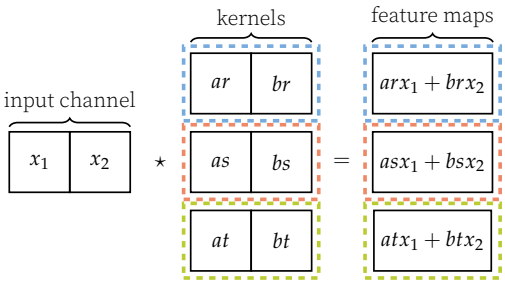


Figure 14: Applying a non-separated convolution with three kernels on an input channel of size two.

The same feature maps can, however, be computed, by separating the kernels by their factors and applying them in sequence. In Figure 15, a kernel with values a, b is applied on the input channel. This operation is called a *depthwise convolution* because the number of kernels used matches the depth of the input. A fact that is not immediately clear from our simplified example, because the depth is one. Interestingly, a depthwise convolution is equivalent to a grouped convolution with an equal amount of groups and input channels. The three kernels with values r, s, and t, respectively, are then applied to the intermediate result in what is known as a *pointwise convolution*. It is called pointwise because it is applied to a single point, or index, across all input channels. Notice that the results in Figure 14 and Figure 15 are identical

due to the distributive law. However, in the latter, there are fewer computations, because the intermediate result $(ax_0 + bx_1)$ is only calculated once. The reduction in computations in this simplified example is minuscule due to the small size and amount of kernels, but in practical use-cases, a significant reduction is achieved.

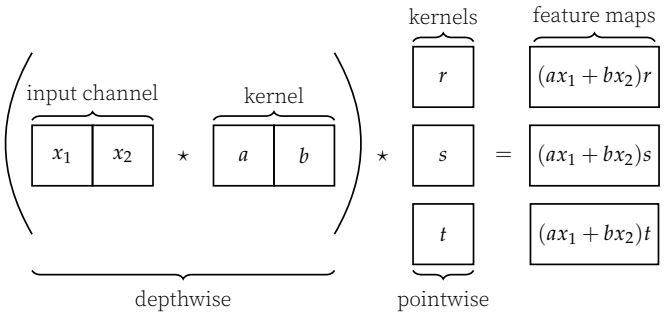


Figure 15: Applying a separated convolution on the same input as in Figure 14, which results in the same output, but fewer computations.

All of the extensions delineated provide different ways to reduce the computational cost and number of weights of convolutions. Furthermore, the extensions can be combined, which allows for an even more considerable amount of experimentation when using convolutions. Moving beyond convolutions, we will now explain the residual layer type.

3.2.2 Residual

In Section 3.1, we present how multiple perceptrons can be combined into layers, and show that adding more layers increases the expressiveness of a neural network, which enables it to approximate more complicated functions. By adding more layers, we should, therefore, be able to improve the accuracy of a network. However, adding layers to a network does not guarantee a higher accuracy, as proven in [40]. In some cases, adding layers to a network makes it unnecessarily complex. The issue can be amended if additional layers can represent the identity function ($i(x) = x$). Although a set of sequential layers might be able to represent the identity function, it is not always easy for it to learn this particular function, because it highly depends on the set of possible inputs. Consider the input x and a set of layers with their function represented by $f(x)$ and arbitrary target function $t(x)$. *If* the target function is the identity function $i(x)$, then every parameter in $f(x)$ must be adjusted to map this correctly, which is difficult. The simple solution to this problem is to introduce a residual connection [40]. A residual connection is made by adding x to $f(x)$, as shown in Figure 16. If the target function is $i(x)$, $f(x)$ is only required to output zero, which is a lot easier. This means that for a target function $t(x)$, $f(x)$ must map the *deviation* from $i(x)$ instead of $t(x)$ itself.

3.2.3 Batch Normalisation

During training the distribution of values in each layer's inputs, or activations, change, because the network is continuously updated. This property makes it difficult for a given layer to learn

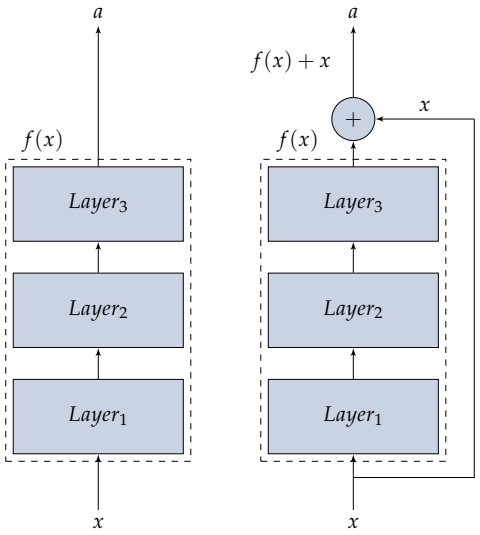


Figure 16: A regular block (left) and a block with a residual connection (right).

because it must account for the changes to the previous layers. A lower learning rate can help alleviate the problem, but it increases the time needed for training. The authors of [41] refer to this problem as *internal covariate shifting* and suggest to continually normalise the activations throughout the network to reduce this effect. By normalisation, they mean to ensure a mean of zero and a variation of one. They call their method for *batch normalisation*, and the novelty is not normalisation on its own—which is a common preprocessing step used in artificial intelligence tasks [42]—but rather the continual application of it throughout the network.

Batch normalisation is applied differently for training and prediction. It uses the same formula for both, as seen in Equation (13), but the values used for mean and variance differ. During training, the mean and variance are calculated based on the activations, $\mathbf{x}$, in the current batch—which is a subset of the data used for training. A running average for each of these two values is stored and used during prediction. Besides mean and variance, a trainable weight, γ, and bias, β, is used. Similar to the argument made in Section 3.2.2, it is preferable if a layer can represent the identity function. By setting $\gamma = \sqrt{variance(\mathbf{x})}$ and $\beta = mean(\mathbf{x})$ we cancel the effect of the layer thus guarantying that we can represent the identity function. To avoid division by zero, batch normalisation uses a small constant ϵ.

$$y = \frac{\mathbf{x} - mean(\mathbf{x})}{\sqrt{variance(\mathbf{x}) + \epsilon}} \times \gamma + \beta \tag{13}$$

The result of using batch normalisation is shown to reduce training time significantly [41].

3.3 Cost Functions

In Section 3.1 we explain the general purpose of cost functions, where we use mean-squared-error as an example. What follows is a delineation of two alternative cost functions: CTC and

KD, which both can be applied to problems.

3.3.1 CTC

This subsection was first presented in our previous work, [3]. Modifications were made for clarification.

In Section 3.1.1, we introduce the concept of cost functions. The choice of cost function relates to what type of problem is being solved. Recall that basecalling is a sequence-to-sequence problem. Some sequence-to-sequence-problems, including basecalling, have an underlying challenge that must be handled. Namely that input and target lengths do not have an exact relation, which means that the size of the output cannot be decided only by looking at the size of the input. *CTC* (Connectionist Temporal Classification) is a method designed to handle sequence-to-sequence problems with this underlying problem [43]. CTC is technically a cost function, but requires the model to output a specific format. With the specific output format, a cost can be calculated and used during training as described in Section 3.1.1. The output from the model required for CTC is *not* the actual DNA sequence, but it can be decoded into the sequence. We will now delineate the following: (1) how to construct a network for basecalling, which can be used with CTC, (2) how to calculate the CTC cost, and (3) how to decode the output of the network to produce a DNA sequence.

Constructing the Network In basecalling the input to the model is a signal from a sequencing device corresponding to the input $\mathbf{x} = \{x_1, x_2, \ldots, x_I\}$, where each x_i is a single signal value. The target output sequence is denoted $\mathbf{y} = \{y_1, y_2, \ldots, y_U\}$, where $y_u \in \mathbb{L} = \{A, C, G, T\}$. Recall that $\mathbb{Y}$ is the set of all target outputs in the training data. In regular classification tasks, $\mathbf{y}$ is often a probability distribution over a one-hot encoded vector. This, however, does not work for basecalling due to the problem of not knowing the exact length of output needed. In order to ensure that the prediction can represent the target sequence, the output length must be as long as the longest possible target ($\arg\max_{\mathbf{y}\in\mathbb{Y}} |\mathbf{y}|$). Inevitably, this introduces an issue when the target $\mathbf{y}$ is shorter since the network will output a probability distribution over $|\hat{\mathbf{y}}|$ letters instead of $|\mathbf{y}|$. We can overcome this by, after converting the probability distribution, collapsing consecutive identical characters into a single character, such that a prediction, e.g., $ACCCA$ can be collapsed into its target ACA. This, however, introduces another problem which is detecting when consecutive identical letters should *not* be collapsed. Assuming the same output $ACCCA$, it is unknown whether CCC represents C, CC, or CCC. To overcome this issue, we can add a blank character ($-$) that acts as a divider, thereby extending the output alphabet to $\mathbb{L}_e = \mathbb{L} \cup \{-\}$. With the addition of the blank character to the alphabet, we get an unambiguous sequence by removing repetitions and then blanks from the prediction, as shown in Table 3.

Computing the Cost A naive approach to computing the cost is to use $arg\max$ to decode the probability distribution into a sequence of bases and compare the read identity between that and the target. This method is undesirable since it is not able to distinguish two different probability distributions with the same decoded output.

Instead, *CTC* requires the network to output a matrix, Y, with probability distribution over

Before	**After**
ACCCA	ACA
A–CA	ACA
AC-CA	ACCA
AC-C-CAA	ACCCA

Table 3: Example of using blanks to detect repetitions of bases.

$|\mathbb{L}_e|$ classes, with g columns where $g \geq \arg\max_{\mathbf{y}\in\mathbf{Y}} |\mathbf{y}|$. The columns in the probability distribution are chronological such that the prediction in column t is followed by the prediction in column $t+1$.

A *path* π is a sequence of letters in $\mathbb{L}_e$ with length g. All possible paths are defined by the set $\mathbb{L}_e^g$. The probability of any given path in Y can be expressed by:

$$p(\pi|x) = \prod_{t=1}^{g} p_{\pi_t}^t(Y) \tag{14}$$

where $p_{\pi_t}^t(Y)$ is the probability of the label π_t at time t. $\mathcal{Y}$ represents the set of paths that collapse into y (as shown in Table 3). The aim is to increase $p(\pi|x)\ \forall\ \pi \in \mathcal{Y}$. Finally we can define the conditional probability for a specific sequence of labels l, by summing the probability for all paths that collapse into it:

$$p(l|x) = \sum_{\pi\in\mathcal{B}^{-1}(l)} p(\pi|x) \tag{15}$$

where

$$\begin{aligned} \mathcal{B} : \mathbb{L}_e^g \to \mathbb{L}^{\leq g} &= \text{Function that removes duplicates and blanks.} \\ \mathcal{B}^{-1}(l) &= \text{The inverse function of } \mathcal{B} \text{ and represents the set of all paths} \\ &\quad \text{that collapses into } l. \end{aligned}$$

We use $\mathbb{L}^{\leq g}$ to denote the set of sequences over the alphabet $\mathbb{L}$ with length less than or equal to g. Since $p(l|x) \in [0,1]$ we can define the CTC cost function, that we want to minimise, as $C_{CTC} = 1 - p(l|x)$.

Decoding While being able to calculate the cost is sufficient to train a network, we have yet to convert the probability distribution into a sequence of bases. Formally put, we must decode Y into $\hat{y}$ for a given x. We can get the best path—but not guarantee the best sequence—using a naive approach:

$$\hat{\mathbf{y}} = \mathcal{B}(arg\max_{\pi\in\mathbb{L}_e^g} p(\pi|x)) \tag{16}$$

This will produce a sequence with the most likely character for each time step and then collapse that sequence. This, however, will not guarantee the most probable label sequence. Instead,

we can compute the most probable label sequence using Equation (17). The difference is that we consider every possible path resulting in each label sequence. However, the complexity of examining every path in $\mathbb{L}_e^s$ grows exponentially with the length of the input sequence.

$$\hat{\mathbf{y}} = arg \max_{l \in \mathbb{L}^{\leq s}} p(l|x) \tag{17}$$

Beam search [44] is a compromise between the two methods. It is a breadth-first search algorithm that uses the parameters *beam width* and *threshold* to prune away unpromising paths. The beamwidth controls how many paths to investigate, and the threshold is used to discard paths with low probability. These two parameters balance the trade-off between speed and accuracy. Fortunately, they can be adjusted as needed.

3.3.2 Knowledge Distillation

Knowledge Distillation (KD) is a cost function used to train a network using predictions from one or multiple pre-trained neural networks. We refer to the network being trained as the student and the pre-trained model(s) to at its teacher. KD was originally introduced by Hinton in [15] as a method to distil knowledge from an ensemble of larger teachers. Later, KD has been shown to improve the accuracy of students with identical architecture as its teacher [45] and even improve that accuracy of students larger than their teachers [46]. Furthermore, [46] also shows that students even can benefit from KD using poorly-trained teachers.

Although the student can be trained on the output from the teacher alone, it is common to combine KD with another cost function [47]. Equation (18) shows the cost function used for KD, where $\alpha \in [0, 1]$ is a constant weighing in the cost from a given cost function C_{label}. KL is the Kullback-Leibler divergence of the softmax σ and log-softmax λ. The predictions $\hat{y}^S$ and $\hat{y}^T$—from the student and teacher respectively—are smoothed by a temperature T. We use the generic placeholder for a cost function C_{label} instead of cross-entropy, which is used in [47]. We use C_{label} because cross-entropy is not applicable to sequence-to-sequence problems. For basecalling CTC can instead be used as C_{label}.

$$C_{KD} = (1 - \alpha) C_{label}(\hat{y}^S, y) + \alpha T^2 KL(\sigma(\frac{\hat{y}^T}{T}), \lambda(\frac{\hat{y}^S}{T})) \tag{18}$$

[47] shows that regular KD, as Equation (18), performs better than many proposed extensions, and is a lot simpler to implement.

This concludes the explanation of the machine learning needed to understand the remainder of this report. In the next section, we will describe the Bonito architecture before moving onto our experiments.

4 Model Architecture

The basecaller Bonito uses the QuartzNet block architecture [9], which can be seen in Figure 17. A $B \times R$ QuartzNet model consists of B blocks, each with R sub-blocks. A sub-block has the following layers: 1D depthwise convolution (Section 3.2.1), pointwise convolution (Section 3.2.1), batch normalisation (Section 3.2.3), and ReLU (Section 3.1.1). All sub-blocks in a given block outputs the same number of feature maps. For each block, the input is connected to the last sub-block through a residual connection (Section 3.2.2). The residual connection is passed through a pointwise convolution, to account for differences between the number of input channels and feature maps, and then through a batch normalisation layer. QuartzNet calls the combination of a 1D depthwise convolution and a pointwise convolution for a Time-Channel Separable Convolution (TCSConv in the figure), as the values along the depth are related to time. Except for its name, it is identical to the separable convolutions in Section 3.2.1. QuartzNet includes four extra blocks—one before and three after the central part of the architecture, i.e. the blocks with repetitions. All of the additional blocks consist of a 1D convolution, batch normalisation and ReLU, except for the last, which is a pointwise convolution.

We name the blocks in the architecture to enable easy referencing. B_i is the ith B-block with repetitions. We denote the extra blocks with a C. The layout of a QuartzNet architecture is thus $[C_1, B_1, \ldots, B_B, C_2, C_3, C_4]$, where B_B refers to the last B layer.

In Section 1, we note the peculiarity of Bonito and QuartzNet using the same values for their HPs. We will, in the next section, describe the available HPs in the architecture.

4.1 Model Hyperparameters

Having introduced the general architecture, we will now cover how the HPs explained in Section 3.2.1 are used in Bonito. These HPs have been experimented with by Mr Seymour, the creator of Bonito, and the authors of QuartzNet. However, they limit their experiments to variations in amounts and sizes of the B-blocks, thereby excluding the use of different types of layers and variations to the C-blocks [9, 13].

We mention a list of specific HPs that Mr Seymour has experimented with in Section 1. Each HP on the list will be explained and, subsequently, we will investigate whether any additional elements should be included.

Repeats A natural number that corresponds to the value of R in QuartzNet, i.e. the number of sub-blocks.

Number of Kernels A natural number that determines the number of kernels used in a B-block.

Kernel Sizes The sizes of each kernel in a B-block, also a natural number.

In Table 4, we show the exact values used for each HP of Bonito.

The HPs chosen for tuning by Mr Seymour are logical, given the nature and possibilities of the architecture. One notable absentee from the list, however, is grouping, which we explain in Section 3.2.1. In the QuartzNet paper, grouping is applied to the B-blocks. Grouping is shown to markedly reduce the number of weights in the model and thereby increase the thereof [13]. The

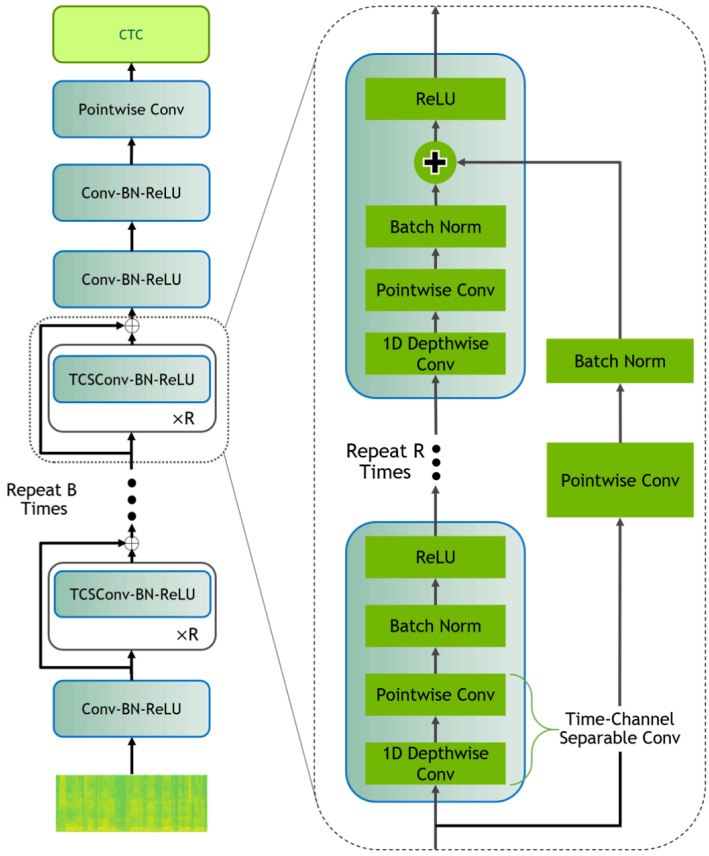


Figure 17: The QuartzNet architecture with B blocks each containing R sub-blocks. Figure is from [13].

Block	R	K	C
C_1	1	33	256
B_1	5	33	256
B_2	5	39	256
B_3	5	51	512
B_4	5	63	512
B_5	5	75	512
C_2	1	87	512
C_3	1	1	1024
C_4	1	1	|labels|

Table 4: Values for Repetitions (R), Kernel size (K), and Channels (C) used in QuartzNet and Bonito.

paper also shows that the speed comes at a relatively low decrease in accuracy for their problem domain, automatic speech recognition. That is, at least with shuffling, which they use for their experiments. With our focus on both speed and accuracy, it seems sensible to investigate the effects of grouping and shuffling as relating to basecalling.

We search for further extensions to convolutions that can positively impact speed and accuracy. Despite our efforts, we only find one sensible extra extension, which is dilation. We introduce dilation in Section 3.2.1, and as we explain, it can increase the effective region in which a kernel operates, without adding additional weights. This effect might be of particular importance to basecalling for a MinION device because the surrounding region of DNA affects the signal produced, as explained in Section 2.3. A wider effective region thus has the potential ability to increase the accuracy without an increase in speed.

Lastly, we must mention that Bonito uses a stride of three for the first C-block instead of two—as it is in QuartzNet. In the following sections, we document our experiments with alternative configurations of the B blocks of Bonito. In these, we will not be experimenting with alternative configurations of the C-blocks. The HPs of the C-blocks will, therefore, be fixed to those of Bonito.

Having defined the architecture of Bonito and its HPs, we will in the next section explain how we, through experiments, aim to gain an insight into the impact of the HPs.

5 Experiments

In Section 1 we state our intent of investigating the effects of HP-tuning and KD on Bonito with both accuracy and speed as metrics. We do so empirically through experiments. Our initial experiment, referred to as Experiment 0 (`Exp0`), turned out to have several flaws, of which we learned a great deal from concerning how subsequent experiments should be executed. While the lessons are valuable, the results are less so, and it has therefore been included in the appendix, see Appendix B. `Exp0` is, in essence, a result of being overly eager and optimistic about the size of our experiments. When relevant, the lessons learned from `Exp0` will be included. In the three following subsections, we explain the experiments that we conducted along with their motivation and contribution. The data used for training the models as well as evaluating the results are described in Section 5.1. Experiments one, two, and three are publicly available on GitHub[1] and Weights and Biases[2].

5.1 Data

The data set consists of eight different bacteria with a total of 480,605 signals produced by the MinION device, where each signal consists of approximately 3,000 to 5,000 reads. We use a subset of 1,000 randomly selected signals where the bacteria Bacillus has been excluded. This subset is further divided into a training and validation set (80% and 20% respectively). While the training set is the data used for training the network, the validation set is used to avoid overfitting. Overfitting is where the model becomes very good at predicting the training set but unable to generalise. This is done by evaluating the loss of the validation set after each epoch of training. At some point during training, the loss for validation will begin to worsen, while the loss of the training set still improves, which means that the network no longer generalises well.

We choose a relatively small subset of signals to make the training of models faster, thereby enabling us to include more variants of models in each experiment. To further reduce the time needed per experiment, we have chosen to use CTC loss as a proxy for the accuracy measure for the models in the first two experiments. Our results from `Exp0` show a strong correlation between CTC loss and accuracy. In the experiments, we normalise prediction time (time) and CTC loss across all of our experiments, on each measure. The normalisation allows results from each experiment to be compared. In `Exp3` we calculate the read identity, defined in Section 2.3.1, using NanoPlot [48] of a subset of 200 signals for Bacillus as well as a random selection of 200 unseen signals and compare it to those of Bonito. We refer to these two data sets as test sets. A test set is used to measure the final accuracy of a neural network. The validation set cannot be used for this purpose, because it is used to stop training when the network performs the best on that data set, thereby intruducing possible bias for this set of data.

5.2 Experiment 1: Random Search

As we mention in Section 1, the values used for HPs in Bonito are identical to those used in QuartzNet [9, 13]. QuartzNet shows the effect on *accuracy* for a subset of the HPs. The effects on

[1] https://github.com/Jgfrausing/basecaller-p10
[2] https://wandb.ai/jkbc/jk-basecalling-v2

speed are, however, not shown. In 4.1 we show the HPs used for tuning by Bonito and QuartzNet. We also add dilation as a novel HP to investigate.

As we shall see, every HP has an extensive range of possible values. Creating and training models for each of all the possible combinations of HPs would yield the most precise insights into their impact on speed and accuracy. Unfortunately, each model takes several hours to train, which makes an exhaustive exploration of the search space infeasible. Instead, this experiment aims at training a representative set of model configurations—i.e. a particular set of values for the HPs— to investigate the interactions between different HPs.

5.2.1 Setup

In `Exp0` changes were made to both the model HPs and the algorithm HPs. The results from `Exp0` showed which values for our algorithm HPs that worked well for a large range of configurations of model HPs. While we gained insights on algorithm HPs, we learned less from the different model HPs. The reason being that we allowed each model HP to change independently for each B-block, thereby creating an enormous search space.

For this experiment, we, therefore, choose to reduce the number of possible configurations by changing each of the different type HPs (i.e. repeats, dilation, grouping, etc.) for every B-block by the same factor. The factors are based on the values used in Bonito, and any deviation is, therefore, a mutation of Bonito. Scaling by a factor reduces the set of possible mutations significantly since only one value for each type is to be changed instead of five—i.e. one for each of the five B-blocks. It is worth mentioning that, similarly to [9, 13], we fix the HPs of all C-blocks to reduce the number of combinations further.

Dilation Since dilation is not explored in Bonito, the default value for every B-block is one. In Section 3.2.1, we explain why dilating by one has no effect. We allow mutations where the dilation can be increased to two and three.

Grouping The second introduction we make to Bonito, is grouping. Similar to dilation, the default value for grouping is one. We allow mutations that use two, four, and eight groups. Additionally, these mutations are combined with and without shuffling.

Kernel Sizes Default values in Bonito is 256 for B_1 and B_2 and 512 for the three other B-blocks. We mutate by scaling using a factor $s \in \{1.3, 1.2, 1.1, 0.9, 0.8, 0.7\}$ on all blocks. Mutations on dilation alone do not change the number of weights, as explained in Section 3.2.1. Therefore, we account for dilation when modifying the kernel size. For a given mutation of kernel size, the size is further adjusted such that its effective size *after dilation* is the same as the size of the kernel with dilation 1. Table 5 shows how a kernel of size seven is adjusted according to dilation.

Number of Kernels Similar to their size, the amount of kernels in each block are also adjusted using the scaling factor s. As explained in Section 3.2.1, the number of kernels has to be divisible by the number of groups. Since one, two, and four are factors of eight, the largest number of groups, we adjust the number of kernels after scaling to be divisible by eight.

Dilation	Kernel						
$\mathfrak{D}_1$	w_1	w_2	w_3	w_4	w_5	w_6	w_7
$\mathfrak{D}_2$	w_1	0	w_3	0	w_5	0	w_7
$\mathfrak{D}_3$	w_1	0	0	w_4	0	0	w_7

Table 5: Adjusting a kernel of size seven with dilation values one, two, and three. The number of weights decrease with a higher dilation, but the effective size of the kernel remains the same.

Repetitions While Bonito uses five repeats for each B-block, QuartzNet have experimented with more and fewer repetitions. We define the set of possible values for mutating repetitions to be $[1, 2, ..., 10]$.

The values for HPs used in Bonito are well-proven to give good results, both given the results of Bonito itself and also the results presented for QuartzNet in [13, 49]. Given this, we consider the above mutations to be a good compromise in order to reduce the number of possible configurations to investigate.

The search space of all combinations of the mutations totals to a little more than nine thousand models. Due to time constraints, we choose to limit the experiment to a random selection of 150 of these configurations. For reliable results, each model is trained until its loss (or cost) stabilises, which takes several hours. The models created from these configurations, naturally differ in complexity. Therefore, they are trained for a maximum of 30 *epochs*, which in `Exp0` has shown to be sufficient for most configurations. One epoch is complete when all of the training data has been seen once. We use *early stopping*, which stops training when the loss has not improved by at least 0.01 over three epochs. Early stopping is used to reduce the workload, given that the configurations on average converge after 20 epochs.

5.2.2 Results

With the setup established, we move to describe and analyse the results.

Pareto set A Pareto set (or Pareto optimal set) can be created from a set of data points in a multidimensional space. A data point is said to be in the Pareto set if another point does not dominate it—that is no other point is better in all dimensions. For our setup, the dimensions are time and accuracy, and the data points are the models. Models not part of the Pareto set will, consequently, have one or more alternatives that are both faster and more accurate. A Pareto set is constructed from the randomly mutated models, as shown in Figure 18. Given that 94 configurations have a shorter running time than Bonito, it is notable that Bonito remains a part of the Pareto set. However, by allowing a slightly worse loss, the running time can be reduced significantly. Similarly, the loss can be reduced markedly with a slight increase in running time.

Feature importance Feature importance is a method to describe the impact a given input feature has when using a machine learning model to predict an output [50]. This method is especially valuable when features can be dependent on others. [51] suggests that feature importance also can be used to get an insight into which HPs have the highest importance.

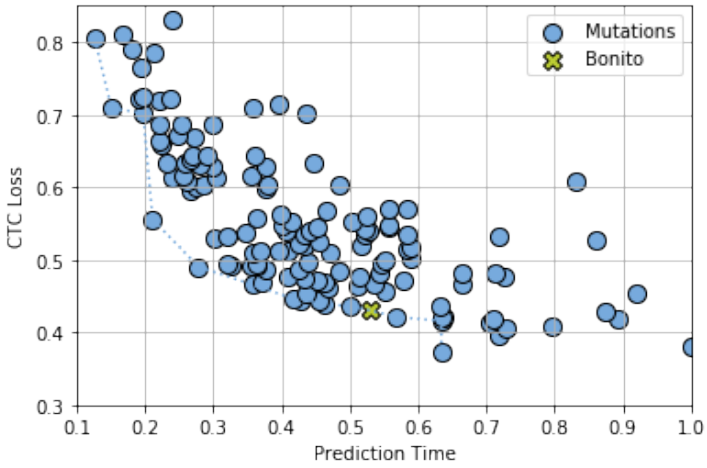


Figure 18: Pareto set defined by the dotted line. The blue dots are mutations. The cross represents Bonito. CTC Loss and Prediction time is normalised according to the highest value for each measure across all of our experiments.

The method uses a random forest regressor, which is an combination of *decision trees* [52]. A decision tree is, similar to a neural network, a machine learning method used to predict a value given an input. It uses a tree structure, where each node makes a branching decision on its input. Nodes starting from the top layer, each node in the tree is constructed by taking the best split of a training set using the most decisive feature. The best split is the one that reduces a specified error the most (e.g. mean squared error).

As defined in [50, 51], we train a random forest regressor using the HPs as features and the loss, representing the accuracy, of the networks as targets. We disregard speed because we cannot combine speed and accuracy into one measure without selecting a single trade-off. By the definition of decision trees, important HPs (a.k.a decisive features) will on average be positioned higher in each of the trees in the regressor. Hereafter, we randomise the values for a specific HP h for all networks. If h is important, it will, therefore, be difficult for the regressor to make correct predictions. However, if h is of less importance, then the regressor will retain more of its correctness. By doing this for all HPs, we can order them according to their importance.

Figure 19 shows the importance that each HP has according to the loss along with its correlation. It suggests that increasing *kernel size*, amount of *kernels*, and *repetitions* yields a decrease in loss. The decrease of loss is no surprise given that increases in each of these HPs result in models with more weights. *Grouping* and *shuffling* is shown to have an impact on the loss. It is, however, difficult to know if this is the case for all group sizes. Lastly, the figure suggests that *dilation* is of low importance and has no correlation to the loss.

Importance is, as mentioned, measured according to accuracy. Figure 19 implies that dilation and groups are of low importance to accuracy. However, we are still interested to see if there are *any* configurations where dilation and grouping can provide a valuable contribution

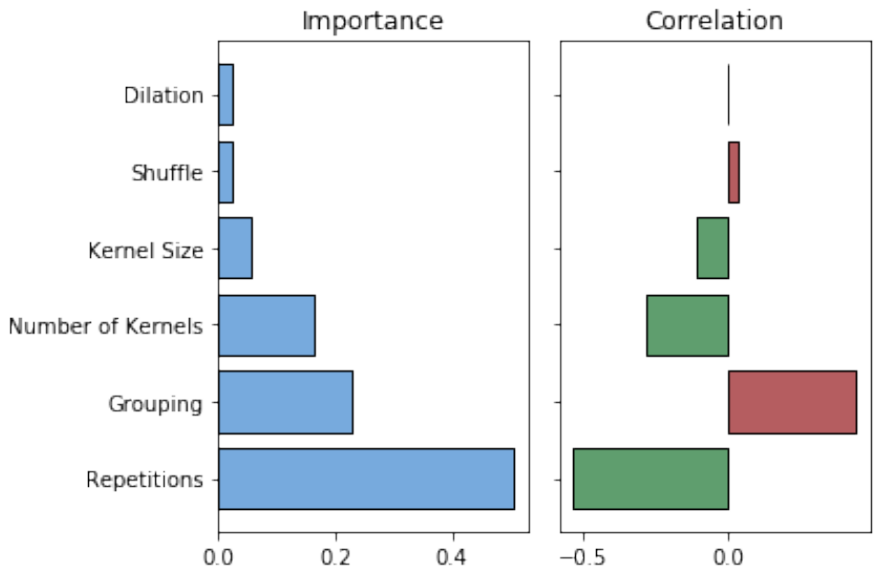


Figure 19: Importance and correlation of each HP to loss. Positive correlation means a higher value for that HP results in a higher loss—and is therefore marked by red.

when speed is also considered. For this purpose, the 14 configurations in the Pareto set are interesting to investigate, since they are good trade-offs between time and accuracy. Of those 13 configurations (excluding Bonito) only four does not use dilation, which suggests that dilation could provide some value when not only accuracy is considered. Grouping, on the other hand, is only represented two times in the Pareto set, and both of those use two groups.

Summary This experiment contributes with the following:

1. The random set of HP configurations has yielded a Pareto set that includes Bonito.
2. The importance and correlation measure suggests that dilation and grouping will not reduce the loss.
3. The configurations of the Pareto set hints that dilation and using two groups can be of value when considering both loss and prediction time.

The importance measure falls short when both time and loss is considered. In order to gain a better understanding of how each HP affects the trade-off independently, we will in `Exp2` apply a more systematic approach to investigate the individually impact of the HPs further.

5.3 Experiment 2: Grid Search

Results from `Exp1` show that increasing the number of weights in Bonito can yield a decrease in loss. This increase is not useful when aiming to reduce prediction time. We show the impor-

tance of each HP according to the loss, a proxy for accuracy, but the impact on the time/accuracy trade-off is difficult to interpret. Here we perform a *grid search* over each HP and view the results in terms of speed and loss. A grid search is simply a complete search over a given range of values.

5.3.1 Setup

In order to enable comparison of these results and those of `Exp1`, the setup is almost identical. The only difference being that the HPs are changed independently, and we investigate all values in the specified ranges, as is the nature of a grid search. All combinations of dilation and kernel sizes are, furthermore, tested, for the reasons explained in `Exp1`.

5.3.2 Results

The results of the grid search allow us to create plots that show the trade-off between loss and speed for each HP. We choose to relegate the results for *Number of kernels* and *Repetitions*, to Appendix C, since they exhibit the same patterns as in `Exp1`.

Dilation and Kernel Sizes One of the most interesting results is the dilation and kernel size experiment. Recall that the kernel sizes are scaled to effective kernel sizes using dilation, which means that experiments with dilation larger than 1 have smaller kernels. The formula for computing the effective kernel size is shown in Section 3.2.1. Figure 20 shows the impact of different combinations of dilation and kernel sizes have on time and accuracy. As expected, the prediction time is, in general, lowered when applying dilation because we also reduce the kernel size. However, the two lowest values for loss are achieved by using dilation. So, as we theorised in Section 4.1, dilation can, in combination with a reduced kernel size, replace larger non-dilated kernels while yielding similar or lower loss at a faster speed. The *can* is emphasised since variation in the loss for the results indicate that proper tuning is essential. Also noteworthy is the position of Bonito, which is dominated by nearly other configurations, suggesting that the values in Bonito were not ideal.

Grouping The results of grouping and shuffling are shown in Figure 21, where two patterns emerge. The first is that the loss increases with the number of groups, which is consistent with the findings from `Exp1`. Intriguingly, two groups with shuffling seem to be a direct improvement over Bonito. The second pattern is that shuffling reduces the prediction time, which is odd because it is an additional operation. One possible explanation is that the shuffling implementation moves the data in memory according to the new shuffled ordering. The rearranged data can, therefore, potentially be a better match for further processing. However, the intricate low-level GPU implementations make it hard to know for certain with further investigations.

Summary The additional runs result in an updated Pareto set, as seen in Figure 22. Most

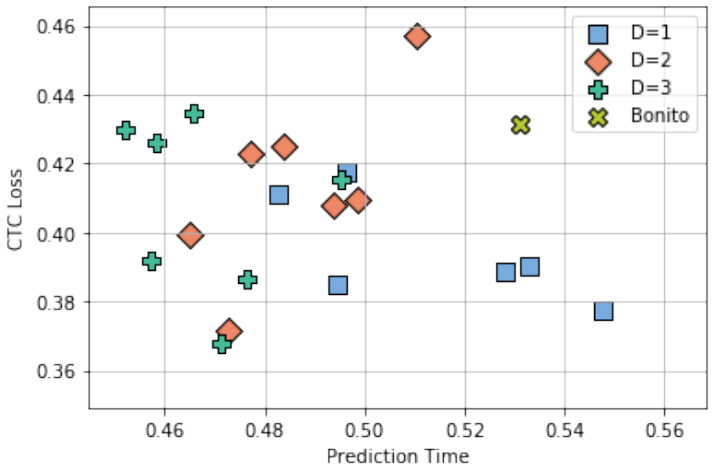


Figure 20: Various kernel sizes with dilation $D \in \{1, 2, 3\}$. Bonito, which has dilation of 1 is also shown.

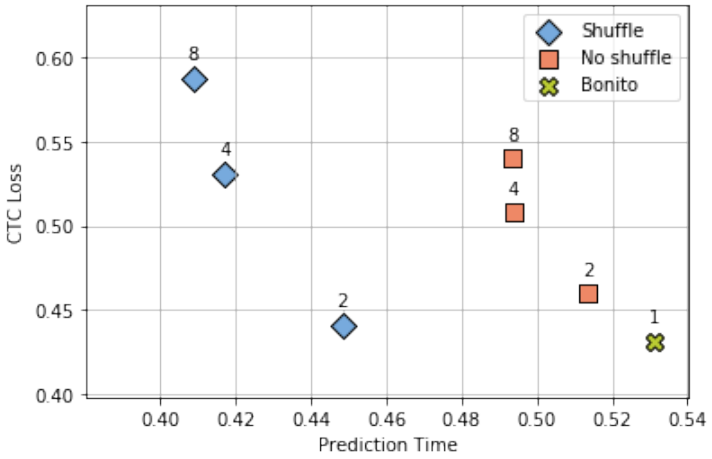


Figure 21: The results of using grouping and shuffling on Bonito. The number of groups are written above each marker. The legend explains the colours and markers used.

HPs can be adjusted to improve either speed or accuracy at the cost of the other. Dilation combined with adjusted kernel size, however, is shown to reduce both measures. The experiments with dilation is highlighted with **D**s in Figure 22.

The named models in Figure 22 are selected to represent the Pareto set for additional experimentation concerning KD in the following section.

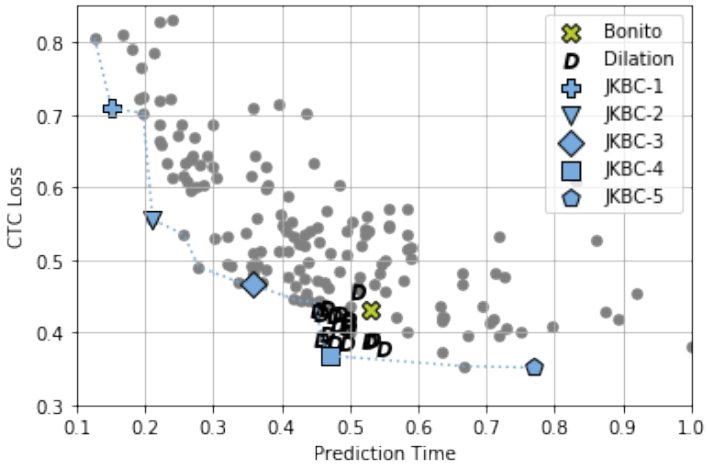


Figure 22: Updated Pareto set. The dotted line refers to the Pareto set for the joint experiments. **D** are dilated models. The named models are a selected subset which are to be used in `Exp3`.

5.4 Experiment 3: Knowledge Distillation

In the final part of `Exp2`, we select five configurations that represent the Pareto set well. The configurations and Bonito will be trained with KD in this experiment. KD has a number of algorithmic HPs which need to be set. We conduct experiments to find the best values for the HPs and subsequently uncover the effect of KD with respect to basecalling.

5.4.1 Setup

We explain the algorithmic HPs for KD, α and T, in Section 3.3.2. In [47], a grid search of three values for both of these is conducted on one model. The difference in accuracy for these $3^2 = 9$ different configurations is less than one per cent. Given that we are interested in applying KD to six models, a grid search for all values is deemed to be too time-consuming. Instead, we divide the selection of these values into two steps. First, we select a value for T, which is then used when searching for a value for $alpha$.

Selecting T In `Exp1`, we use early stopping meaning that training terminates when the decrease in validation loss is below a fixed threshold. We continue to use early stopping with the same criteria, to ensure that models trained with KD are not training longer than the ones

without KD. This choice means that, although the cost function used for training is another, we still use CTC loss to determine when to stop training. The idea of using early stopping based on CTC works as long as reducing KD loss also reduces CTC loss—if not, it will likely result in faulty early stopping. Consequently, a suitable temperature value, T, is not only one that produces good models, but also one where CTC loss is reduced proportionally to the KD loss. We acknowledge that this combined criterion can mean that the optimal temperature for producing good models is not selected. We make a grid search with the values $T \in \{1, 4, 7, 10, 13, 16, 19\}$ on Bonito and fix $alpha$ to one. From these trials, we see that for $T \geq 4$, the difference between KD loss and CTC loss is nearly constant after just two epochs (Figure 32). Similar to the results in [47], we see that T does not affect the overall accuracy to a large degree. With our previously stated purpose in mind, we choose $T = 4$, as it seems to be the best value for our criteria.

Selecting α Having selected a value for T, we can experiment with multiple values for α across the six models on which we apply KD. All of the selected configurations are trained with a grid of $\alpha \in \{0.25, 0.5, 0.75, 1.0\}$. The teacher predictions are generated using Bonito.

Figure 23 shows the results of training the six model configurations using the four α values. We highlight each original model trained without KD as well as their best configuration of the four α values. In Table 6 the HPs used for each model is presented.

As we can see, KD, along with appropriate values for T and α, provide a decrease in the overall loss. Surprisingly, even Bonito can benefit from being trained on predictions from itself.

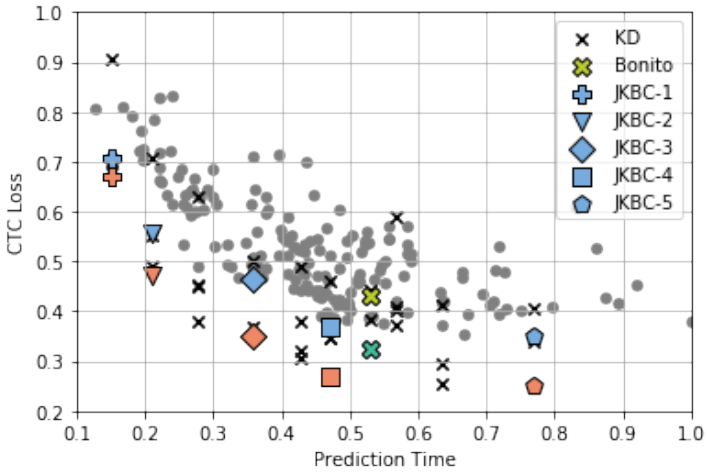


Figure 23: Improvements of applying KD. The nodes with alternative colour are the best outcome of applying KD. Crosses represents additional runs where KD has been applied. Figure 31 is a box plot for the accuracy of different alpha values. It suggests that $\alpha \in [0.25, 0.5]$ yields the best results for our setup.

5.4.2 Results

For the final evaluation of accuracy of our best models, we use two different data sets. Neither of which has ever been used for training or validation. *Test Set 1* (TS1) consists of signals from the same seven species of bacteria as used in both the training and validation sets. *Test Set 2* (TS2) only contains signal from the bacteria Bacillus—the species excluded from both the training and test sets. Beam search (Section 3.3.1) converts the probability distribution of each model into a sequence of bases, which is then aligned to the reference (Section 2.3.1). Table 6 shows the accuracy of each model on each test set. There is a gap of around four per cent accuracy between the two data set for all models. However, the accuracy on TS1 remains valuable, given that some basecallers are trained on taxon-specific data [7].

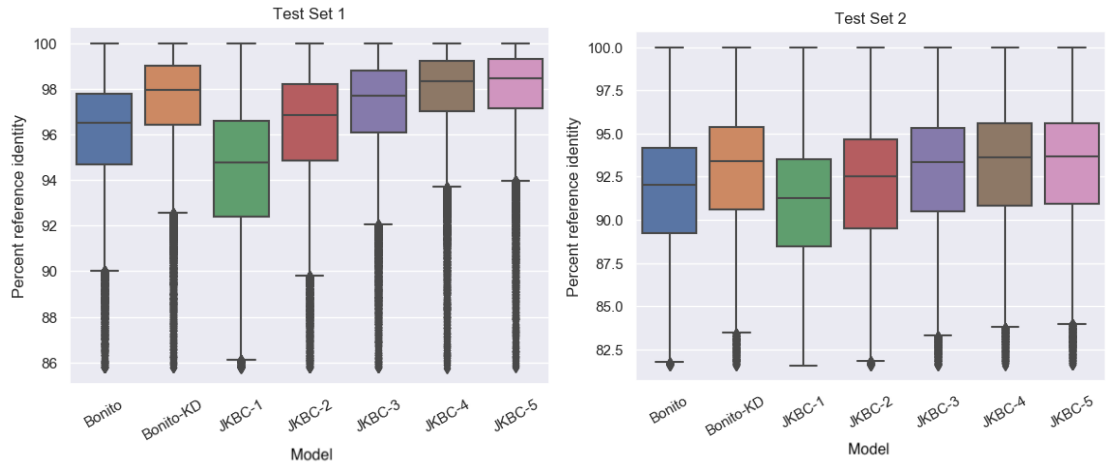
In `Exp1` and `Exp2`, we use CTC loss as an indicator for the overall accuracy of a given model. By this measure, Bonito, trained with KD, ranks third overall among the highlighted models. We see that this ordering of the models is the same when measuring accuracy on both TS1 and TS2. This reconfirms our findings from `Exp0` that showed a strong correlation between CTC loss and accuracy.

Figure 24 is a box plot of the five JKBC models and Bonito trained with and without KD. Here we see that the slower models, JKBC-4 and JKBC-5, outperforms the others on accuracy. JKBC-3, which is 63% faster than Bonito, has similar accuracy, which is also shown in Table 6.

	Bonito	Bonito-KD	JKBC-1	JKBC-2	JKBC-3	JKBC-4	JKBC-5
Time (Normalised)	0.549	0.549	0.150	0.249	0.349	0.488	0.748
Median Identity % (TS1)	96.5	97.9	94.6	96.8	97.7	98.3	98.4
Median Identity % (TS2)	92.0	93.4	91.2	92.4	93.3	93.6	93.6
Kernel size (scale)	1	1	1.2	0.7	1.3	1.2	1
# of kernels (scale)	1	1	0.8	0.7	1	1.1	1.3
Groups	1	1	2	1	1	1	1
Shuffle	-	-	False	-	-	-	-
Dilation	1	1	2	3	3	3	1
# of repetitions	5	5	1	3	3	5	5

Table 6: A comparison of selected models from the Pareto set along with Bonito. Time is normalised similar to previous results. The median identity in percentage is shown for Test Set 1 (TS1), all bacteria except Bacillus, and Test Set 2 (TS2), just Bacillus.

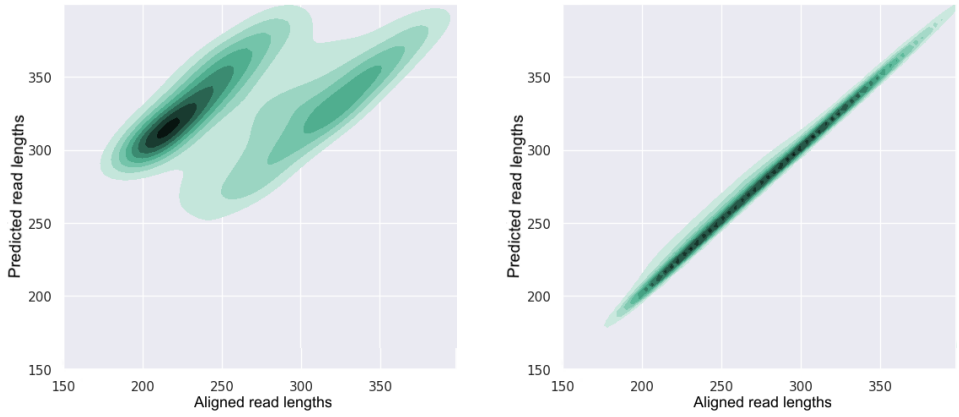
Impact of KD It is clear from Figure 23 that KD improves the accuracy. While any increase in accuracy is valuable, it is also interesting to reach insights into *how* KD affects the models trained. Figure 25 shows one of the best clues we found while trying to extract insights. It shows the relationship between the length of the predicted read length and the length of the *aligned read*—the part of the predicted DNA sequence which could be aligned to the reference sequence. While the figure only shows the difference between Bonito and Bonito-KD, the same pattern emerged for all of our models. Models without KD output long sequences for most of



(a) TS1 (b) TS2

Figure 24: A boxplot of the accuracy (Percent Identity) for selected models on Test Set 1 (left) and Test Set 2 (right). The outliers are hard to distinguish, due to the large number of signals used to produce the box plots.

which only two thirds can be aligned to the reference. On the other hand, models *with* KD tend to output shorter sequences, of which nearly all of the sequence can be aligned. In short, KD improves both the accuracy of the output length and the accuracy of the individual bases.



(a) Bonito (b) Bonito-KD

Figure 25: Output length, y-axis, vs aligned length for Bonito and Bonito-KD for TS1.

Summary KD has been applied on Bonito as well as five selected models in the Pareto set. These models were selected such that they represent different time and accuracy trade-off. We find that every model can be improved by applying KD given appropriate values for the algorithm HPs alpha and temperature.

6 Discussion

Our primary contribution in this project has been the investigation of the impact on speed and accuracy of tuning different HPs on the state-of-the-art-basecaller Bonito. Additionally, we have explored how KD, combined with well-selected HPs, can further improve the accuracy.

Our results suggest that we can divide the HPs into two categories based on their effect. The HPs either: (1) Improve either accuracy or speed at the cost of the other; or (2) improve both accuracy and speed. HPs in (1) are the number of repetitions, kernels, and groups. We achieve prominent results with the combination of reducing kernel size and adding dilation, which suggests that this combination is in the second category, (2). Furthermore, our results show that using KD, with adequately selected values for temperature and alpha, can improve the overall accuracy of existing well-performing models. With this, we accommodate the need for faster and more accurate basecalling as stated to be important in Section 1.

Our contribution builds on empirical results from training an extensive set of Bonito-variants both with and without KD. The quality of the results is, therefore dependent on the data and the method, which we will discuss first. Subsequently, the experiments and results will be examined.

Data In our experiments, a relatively small training set is used to allow for more experiments to be run. The size of a training set is positively correlated with the training time for a model. A small training set, however, increases the risk of overfitting—i.e. getting results that do not generalise well to different data sets. Despite the risk, Table 6 and Figure 24 show that the selected models perform well on test set 1, which contains the same species as used in the training data. The accuracy is nearly five per cent lower for test set 2, which consists of the Bacillus bacteria; a species hitherto unseen by the models. This discrepancy suggests that the difference between signals for different species is vast and challenging to capture, which is also the case presented in [7] and [29]. Whether the discrepancy would be even larger on animal or plant DNA—i.e. DNA from different biological kingdoms—is something we leave for future experiments. Should these future experiments show that our results do not generalise well, we suggest to redo `Exp1` and `Exp2` with the relevant data. Additionally, it would be of great value to see whether our observed patterns on the HPs would change for a very heterogeneous data set, for example, one including species from several biological kingdoms.

CTC Loss as a Proxy for Accuracy We choose to use CTC Loss on the validation set as a proxy for accuracy (read identity) for all results, except for the ones, including the test sets. The choice is made on the basis that read identity, from Section 2.3.1, is too time-consuming to compute. Using read identity would, therefore, markedly increase the evaluation time of the trials in our experiments. We acknowledge that this choice introduces the risk of us selecting sub-optimal models for further considerations due to the iterative nature of our experiments. We accept that our results for all experiments potentially could have been improved if we had used read identity instead. Despite this caveat, we still believe that dilation, combined with a reduced kernel size, can improve accuracy and reduce prediction time. Furthermore, the models, which are

shown in Table 6, are still a valid contribution even if models exhibiting better trade-offs exist. For similar experiments, especially ones focused on improving basecaller or of non-iterative nature, we recommend the usage of CTC loss as a proxy for accuracy.

Time Needed for Experiments Our contribution builds on empirical results from training an extensive set of Bonito-variants both with and without KD. To ensure the validity of our results, we train each model until its loss stabilises, which takes between three and six hours. Each model is trained on its own GPU, of which we had several available. The combined computation time for our experiments is slightly over 80 days. We could, possibly, have reduced the total computation time needed by training each model for a shorter timer, i.e. for fewer epochs. However, the number of epochs needed is unclear. An experiment showing the relationship between accuracy measured after each epoch and the final accuracy would be beneficial in deciding this matter. Naturally, one other reason for the large computation time needed for our experiments is that we provide insights into the combination of HP-tuning and KD. Especially the HP-tuning experiments are time-consuming. Based on the contribution in this report, we recommend future experiments to focus on *either* HP-tuning or KD.

Hyperparameters An inherent problem with HP-tuning is the problem of choosing the ranges of values to use. The problem is exacerbated when multiple HPs are tuned at the same time because the search space quickly increases in size. To combat this issue, we deploy a scaling technique with discrete values in a limited range. The scaling technique uses the values from Bonito, and therefore also QuartzNet, as the basis. Neither Bonito nor QuartzNet provide any reasoning for, or show any experiments that explain, the values they use for the HPs. It is, therefore, likely that even better values for the HPs exist, which requires further experiments.

Similar to our scaling technique, we choose to apply the same dilation and grouping to each B-block, where it is could be worth investigating the impact of doing these mutations differently on individual blocks. One interesting idea, conducted by [53], is to increase the number of groups by a power of two with the depth of the network, which is shown to work well on smaller networks.

We consider that experiments with higher dilation than three can be of interest since our results suggest that dilation, in many cases, is positively correlated with accuracy. The fact that dilation improves accuracy suggests that long-range dependencies of the DNA signal are of higher importance than local differences. These dependencies become apparent when comparing JKBC-3 to Bonito-KD. Their accuracy on both test sets are very similar; however, JKBC-3 uses only 60% of the time for its predictions. The reduction in speed is caused by fewer repetitions and a smaller number of trainable weights in the kernels.

Although grouping works as a trade-off between speed and accuracy, our results suggest that tuning of other HPs, such as repetitions and kernels, yields a better trade-off. These observations align with the results from QuartzNet [13]. This observation is further supported by grouping only being represented once in the Pareto set of each experiment. Albeit, this might be because of the exact implementation of grouping in the machine learning framework that we used.

Bonito uses a stride of three on the first C-block, which reduces the length of probability distributions to a third of the input. KD requires the outputs of teacher and student to have the same dimensions, which means that a teacher would have to be trained for every variation of stride used by students. To limit the scope of `Exp3`, we decide to use only one teacher, having one specific stride. We have, therefore, no experiments regarding the effects of stride, which could be interesting to conduct, especially for non-KD related experiments. Both an increased and decreased stride would be worth investigating in our opinion.

Recall that the QuartzNet architecture can be described by $B \times R$. Here, R refers to how many times each sub-block in each of the B B-blocks are repeated. During our experiments, we show the results of different values of R, the *repetitions* HP; however, we do not experiment with B-blocks. In QuartzNet experiments are made with 5, 10, and 15 B-blocks, where each B-block is repeated one, two, and three times respectively. The difference between a 10×5 network and a 5×10 network is that a residual layer is added after each B-block. This difference means that the prior has twice as many residual layers. We propose that further experiments could investigate the impact of different amount of B-blocks.

We deemed that `Exp0` had too many configurations of HPs for us to make any general claims on their impact regarding model HPs. Nonetheless, we selected the algorithmic HPs based on this experiment. Given that the selected values for these generally performed well, we consider them to be viable choices. However, we do admit that there is a chance that other selection of these could lead to different results, though we consider significant changes to be unlikely.

Knowledge Distillation KD has proven to be a powerful tool to improve the accuracy of models, given appropriate temperature and alpha values. While we only scrape the surface of KD, we consider these results as valuable pointers to the potential of KD in relation to basecalling. Future work could focus on exploring the vast amount of extensions to KD [47, 54], and especially extensions with additional focus on KD for sequence-to-sequence problems [55, 56].

In order to reduce the scope of the experiments, we have only used a single model as a teacher—namely Bonito. We choose to do this since the teacher predictions can come from an identical network or networks of smaller sizes, as shown in [46]. However, better results could likely be achieved with another teacher. In the original, seminal KD paper, the author presents the teacher to be an ensemble of models [15]. To improve accuracy, we suggest combining multiple models into an ensemble that is used as the teacher instead. Our results suggest that larger networks profit more from KD than smaller ones. It could be of interest to investigate if this is specific to the KD method and teacher that we have used, or if it generalises to other combinations. Lastly, we are curious about born-again networks, which are networks trained with themselves as teachers [46]. Our results show that Bonito-KD, which is trained with the output of Bonito, gains similar improvement as other networks of similar size. JKBC-5, as a born again network, could have further improvements to its accuracy—we leave this to future experiments as well. Intriguingly, the outputs of born-again networks can be used to train new born-again networks recursively. The degree to which recursive born-again training is beneficial to accuracy could also be investigated.

7 Conclusion

In this research, we aimed to investigate how and to what extent HP-tuning combined with the use of KD on Bonito impacts the trade-off between speed and accuracy—a trade-off, which is essential in basecalling.

Our thorough experiments examine an array of HPs, of which the effect of dilation on basecalling is hitherto undocumented. Similarly, the impact on speed is unexplored for most of the HPs. The results suggest that most HPs in Bonito improve *either* its accuracy or speed. However, we find that combining dilation with a reduced kernel size yields a positive effect on *both* accuracy and speed. Dilation, therefore, is a prominent HP to consider when time is an essential factor.

Utilising the results, we present multiple models that outperform Bonito on speed, accuracy, or even both. Those models illustrate the potential of investigating alternative HPs and their combinations. A subset of models representing sound trade-offs, as defined by the Pareto set, is, subsequently, trained with KD to investigate its impact. The results show a substantial increase in accuracy—with no difference in prediction time—for all models in our experiment. Even Bonito trained with itself as a teacher—also known as a born-again network—yields an increase in accuracy. KD improves not only a basecaller's ability to predict each base correctly but also their ability to, correctly, predict the *length* of the output.

As stated in Section 6, our models outperform Bonito. This result is significant, given that Bonito is SOA in basecalling. The improvements over Bonito on both time and accuracy are relevant, as they comply with the importance of both in relation to basecalling, as described in Section 1 and confirmed by the domain expert Mr Albertsen.

It is important to emphasise that these results might be specific to the data used for our experiments. Additionally, we cannot make any claims about whether the effects observed translates to tasks outside the domain of basecalling. Notably, the experiments conducted are not specifically tailored to the data set nor the domain. Thus, we suggest that one or more of the experiments can be applied when faced with a similar task where both time and accuracy are of importance.

Index

References

[1] M. Stoneking. “Mitochondrial DNA and human evolution”. In: *Journal of Bioenergetics and Biomembranes* 26.3 (June 1994), pp. 251–259. ISSN: 1573-6881. DOI: 10.1007/BF00763097. URL: https://doi.org/10.1007/BF00763097.

[2] Y.-R. Guo, Q.-D. Cao, Z.-S. Hong, Y.-Y. Tan, S.-D. Chen, H.-J. Jin, K.-S. Tan, D.-Y. Wang, and Y. Yan. “The origin, transmission and clinical therapies on coronavirus disease 2019 (COVID-19) outbreak–an update on the status”. In: *Military Medical Research* 7.1 (2020), pp. 1–10.

[3] M. P. Felix Gravila Jonatan Groth Frausing. *Building a basecaller for Oxford Nanopore Technologies devices*. URL: https://bargsteen.com/basecalling/reports/mi905e19.pdf (visited on 02/01/2020).

[4] O. N. Technologies. *MinION sequencing*. URL: https://nanoporetech.com/products/minion. (accessed: 04.05.2020).

[5] Illumina. *Illumina sequencing platforms*. URL: https://www.illumina.com/systems/sequencing-platforms.html (visited on 12/06/2019).

[6] V. Boža, P. Perešíni, B. Brejová, and T. Vinař. “DeepNano-blitz: A Fast Base Caller for MinION Nanopore Sequencers”. In: *bioRxiv* (2020). DOI: 10.1101/2020.02.11.944223. eprint: https://www.biorxiv.org/content/early/2020/02/12/2020.02.11.944223.full.pdf. URL: https://www.biorxiv.org/content/early/2020/02/12/2020.02.11.944223.

[7] R. R. Wick, L. M. Judd, and K. 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.

[8] O. N. Technologies. *New research algorithms yield accuracy gains for nanopore sequencing*. URL: https://nanoporetech.com/about-us/news/new-research-algorithms-yield-accuracy-gains-nanopore-sequencing?utm_content=120296379&utm_medium=social&utm_source=twitter&hss_channel=tw-37732219 (visited on 03/10/2020).

[9] O. N. Technologies. *Bonito: Convolution Basecaller for Oxford Nanopore Reads*. URL: https://github.com/nanoporetech/bonito. (accessed: 04.05.2020).

[10] J. Bradbury, S. Merity, C. Xiong, and R. Socher. “Quasi-recurrent neural networks. arXiv 2016”. In: *arXiv preprint arXiv:1611.01576* ().

[11] D. Askan. *Dropout effects on Convolutional vs. Recurrent Neural Networks*. URL: http://www.doganaskan.com/blog/posts/cnnvsrnn.html (visited on 05/19/2020).

[12] mrgloom. *Deep Benchmark*. URL: https://github.com/mrgloom/Deep-Benchmark-List (visited on 05/19/2020).

[13] 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”. In: *arXiv preprint arXiv:1910.10261* (2019).

[14] M. Feurer and F. Hutter. "Hyperparameter Optimization". In: *Automated Machine Learning: Methods, Systems, Challenges*. Ed. by F. Hutter, L. Kotthoff, and J. Vanschoren. Cham: Springer International Publishing, 2019, pp. 3–33. ISBN: 978-3-030-05318-5. DOI: 10.1007/978-3-030-05318-5_1. URL: https://doi.org/10.1007/978-3-030-05318-5_1.

[15] G. Hinton, O. Vinyals, and J. Dean. *Distilling the Knowledge in a Neural Network*. 2015. arXiv: 1503.02531 [stat.ML].

[16] B. Alberts, A. Johnson, J. Lewis, D. Morgan, and M. Raff. *Molecular Biology of the Cell*. 6. ed. W. W. Norton & Company, 2014. ISBN: 9780815344322.

[17] A. J. Griffiths. *DNA sequencing*. In: *Encyclopædia Britannica*. July 2012. URL: https://www.britannica.com/science/DNA-sequencing (visited on 11/21/2019).

[18] J. U. Adams. *DNA Sequencing Technologies*. URL: https://www.nature.com/scitable/topicpage/dna-sequencing-technologies-690/ (visited on 11/21/2019).

[19] E. R. Mardis. "A decade's perspective on DNA sequencing technology". In: *Nature* 470.7333 (2011), pp. 198–203. ISSN: 1476-4687. DOI: 10.1038/nature09796. URL: https://doi.org/10.1038/nature09796.

[20] I. Illumina. *Sequencing Technology*. URL: https://emea.illumina.com/science/technology/next-generation-sequencing/sequencing-technology.html (visited on 11/21/2019).

[21] M. Meyer and M. Kircher. "Illumina Sequencing Library Preparation for Highly Multiplexed Target Capture and Sequencing". In: *Cold Spring Harbor Protocols* (2010). URL: http://cshprotocols.cshlp.org/content/2010/6/pdb.prot5448.

[22] F. Blattner, G. 3. Plunkett, C. Bloch, N. Perna, V. Burland, M. Riley, J. Collado-Vides, J. Glasner, C. Rode, G. Mayhew, J. Gregor, N. Davis, H. Kirkpatrick, M. Goeden, D. Rose, B. Mau, and Y. Shao. "The complete genome sequence of Escherichia coli K-12". In: *Science* (Sept. 1997). ISSN: 1453-62. DOI: 10.1126/science.277.5331.1453.

[23] O. N. Technologies. *Nanoporetech Community*. URL: https://nanoporetech.com (visited on 11/21/2019).

[24] O. N. Technologies. *MinION*. URL: https://nanoporetech.com/products/minion (visited on 11/21/2019).

[25] O. N. Technologies. *How it works*. URL: https://nanoporetech.com/how-it-works (visited on 09/30/2019).

[26] 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 12/06/2019).

[27] O. N. Technology. *Minion brochure*. URL: https://nanoporetech.com/sites/default/files/s3/literature/minion-brochure-oct19.pdf (visited on 12/06/2019).

[28] A. Vilella. *Next Generation Sequencing*. URL: https://docs.google.com/spreadsheets/d/1GMMfhyLK0-q8XkIo3YxlWaZA5vVMuhU1kg41g4xLkXc/edit#gid=1569422585 (visited on 12/06/2019).

[29] P. Spealman, J. Burrell, and D. Gresham. “Nanopore sequencing undergoes catastrophic sequence failure at inverted duplicated DNA sequences”. In: *bioRxiv* (2019). DOI: 10.1101/852665. eprint: https://www.biorxiv.org/content/early/2019/11/23/852665.full.pdf. URL: https://www.biorxiv.org/content/early/2019/11/23/852665.

[30] JEFworks. *Cigar Strings For Dummies*. URL: https://jef.works/blog/2017/03/28/CIGAR-strings-for-dummies/. (accessed: 0t.05.2020).

[31] T. Charalampous, H. Richardson, G. L. Kay, R. Baldan, C. Jeanes, D. Rae, S. Grundy, D. J. Turner, J. Wain, R. M. Leggett, et al. “Rapid diagnosis of lower respiratory infection using nanopore-based clinical metagenomics”. In: *BioRxiv* (2018), p. 387548.

[32] R. Hecht-Nielsen. “Theory of the backpropagation neural network”. In: *Neural networks for perception*. Elsevier, 1992, pp. 65–93.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[34] S. Bai, J. Z. Kolter, and V. Koltun. “An empirical evaluation of generic convolutional and recurrent networks for sequence modeling”. In: *arXiv preprint arXiv:1803.01271* (2018).

[35] F. Yu and V. Koltun. “Multi-scale context aggregation by dilated convolutions”. In: *arXiv preprint arXiv:1511.07122* (2015).

[36] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[37] Y. Ioannou, D. Robertson, R. Cipolla, and A. Criminisi. “Deep roots: Improving cnn efficiency with hierarchical filter groups”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1231–1240.

[38] X. Zhang, X. Zhou, M. Lin, and J. Sun. *ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices*. 2017. arXiv: 1707.01083 [cs.CV].

[39] L. Sifre and S. Mallat. “Rigid-motion scattering for image classification”. In: *Ph. D. thesis* (2014).

[40] K. He, X. Zhang, S. Ren, and J. Sun. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[41] S. Ioffe and C. Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[42] H. W. Werntges. “Partitions of unity improve neural function approximators”. In: (1993).

[43] 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. Pittsburgh, Pennsylvania, USA: ACM, 2006, pp. 369–376. ISBN: 1-59593-383-2. DOI: 10.1145/1143844.1143891. URL: http://doi.acm.org/10.1145/1143844.1143891.

[44] B. Lowerre and R. Reddy. *THE HARPY SPEECH UNDERSTANDING SYSTEM*. 1977.

[45] T. Furlanello, Z. C. Lipton, M. Tschannen, L. Itti, and A. Anandkumar. *Born Again Neural Networks*. 2018. arXiv: 1805.04770 [stat.ML].

[46] L. Yuan, F. E. H. Tay, G. Li, T. Wang, and J. Feng. *Revisit Knowledge Distillation: a Teacher-free Framework*. 2019. arXiv: 1909.11723 [cs.CV].

[47] F. Ruffy and K. Chahal. "The State of Knowledge Distillation for Classification". In: *arXiv preprint arXiv:1912.10850* (2019).

[48] W. D. Coster. *NanoPlot*. URL: https://github.com/wdecoster/NanoPlot.

[49] J. Li, V. Lavrukhin, B. Ginsburg, R. Leary, O. Kuchaiev, J. M. Cohen, H. Nguyen, and R. T. Gadde. "Jasper: An end-to-end convolutional neural acoustic model". In: *arXiv preprint arXiv:1904.03288* (2019).

[50] J. Howard. *FEATURE IMPORTANCE, TREE INTERPRETER*. URL: http://course18.fast.ai/lessonsml1/lesson4.html.

[51] Wandb. *Find The Most Important Hyperparameters In Seconds*. URL: https://www.wandb.com/articles/find-the-most-important-hyperparameters-in-seconds.

[52] Tin Kam Ho. "Random decision forests". In: *Proceedings of 3rd International Conference on Document Analysis and Recognition*. Vol. 1. 1995, 278–282 vol.1.

[53] T. K. Lee, W. J. Baddar, S. T. Kim, and Y. M. Ro. "Convolution with logarithmic filter groups for efficient shallow CNN". In: *International Conference on Multimedia Modeling*. Springer. 2018, pp. 117–129.

[54] Y. Tian, D. Krishnan, and P. Isola. *Contrastive Representation Distillation*. 2019. arXiv: 1910.10699 [cs.LG].

[55] Y. Kim and A. M. Rush. "Sequence-level knowledge distillation". In: *arXiv preprint arXiv:1606.07947* (2016).

[56] M. Huang, Y. You, Z. Chen, Y. Qian, and K. Yu. "Knowledge Distillation for Sequence Model." In: *Interspeech*. 2018, pp. 3703–3707.

[57] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[58] N. Qian. "On the momentum term in gradient descent learning algorithms". In: *Neural Networks* 12.1 (1999), pp. 145–151. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(98)00116-6. URL: http://www.sciencedirect.com/science/article/pii/S0893608098001166.

[59] T. Tieleman and G. Hinton. *Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning. 2012.

[60] I. Loshchilov and F. Hutter. *Decoupled Weight Decay Regularization*. 2017. arXiv: 1711.05101 [cs.LG].

[61] J. Snoek, H. Larochelle, and R. P. Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 2951–2959. URL: http://papers.nips.cc/paper/4522-practical-bayesian-optimization-of-machine-learning-algorithms.pdf.

Appendices

A Optimisers

Optimisers are used for updating the weights in a neural network during back propagation. In Section 3.1.1, we introduce gradient descent, which is one of the simplest optimisers. Optimisers affect the training time needed for a network [57]. Additionally, certain optimisers can help "getting out of" *local minima*, *local maxima*, and *saddle points*, which are areas in the *landscape* of the cost function with gradients of zero, as shown in Figure 26. When using gradient descent, updating the weights with gradients of zero will cause no change to their value. The learning thereby halts. A non-convex cost function, or any non-convex function for that matter, can have

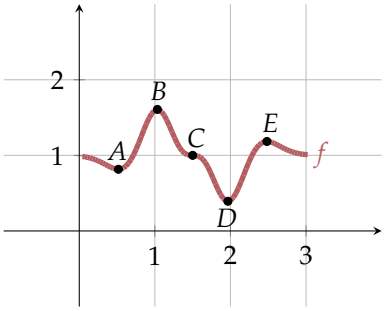


Figure 26: Illustration of an arbitrary cost function f. A and E are local minima and maxima respectively. B and D are global maxima and minima. Lastly, C represents a saddle point.

multiple local minima and maxima. The lowest minimum is called the *global minima*, which is what we, ideally, want to find, hence the need to "get out of" the other local minima or maxima.

In order to clarify the relationship between the optimisers, we show gradient descent in Equation (19) and changes to it are highlighted in different colours when introduced.

In gradient descent, seen in Equation (19), a weight at time t, w_t, is updated based on the weight from the previous time step, w_{t-1}, a learning rate, α, and a *step size*, s_t. The step size is the partial derivative for the weight w_{t-1} relative to the cost function C_t.

$$\begin{aligned} w_t &= w_{t-1} - \alpha s_t \\ s_t &= \frac{\partial C_t}{\partial w_{t-1}} \end{aligned} \tag{19}$$

A.1 Gradient Descent with Momentum

Gradient descent can be altered by adding a *momentum*. The momentum is an exponentially weighted moving average. One way to understand momentum is to view the weight as a particle on top of the cost function's landscape. With gradient descent, we directly change the position of the particle, but with momentum, we change its velocity [58].

Changes from Equation (19) are highlighted in red. The step size using momentum at time t, s_t^m, is different from s_t, as used in Equation (19), in that it is dependent on the previous step sizes as well as the current gradient. β^m controls how quickly the averages decay, i.e. how far back you look in order to determine the moving average.

$$\begin{aligned} w_t &= w_{t-1} - \alpha s_t^m \\ s_t^m &= \beta^m s_{t-1}^m + (1-\beta^m)\frac{\partial C_t}{\partial w_{t-1}} \end{aligned} \tag{20}$$

A.2 RMSProp (Root Mean Square Propagation)

The landscape of the cost function can vary to a great degree, with some areas having smooth, gradual changes, while others will look more like mountain ranges. Recall that the gradients describe the changes in the cost function's landscape. During training, the magnitudes of the gradients can change abruptly, dependent on the topography of the landscape. The abrupt changes make it challenging to find a good local minimum [59]. While the learning rate, α, can help to temper the sudden changes of gradients, a constant value will not work for all situations. Geoffrey Hinton created *RMSProp* in an attempt to solve this problem of having to choose a single learning rate [59]. RMSProp works by dividing the learning rate by an exponentially weighted moving average of the *squared gradients*. This method results in changes to the weights that, to some degree, can ignore large, sudden changes in gradients. Changes from Equation (19) are highlighted in blue. Similarly to Equation (20), we use a factor, here β^r, to determine the decay of the moving average.

$$\begin{aligned} w_t &= w_{t-1} - \frac{\alpha}{\sqrt{r_t}} s_t \\ r_t &= \beta^r r_{t-1} + (1-\beta^r) s_t^2 \end{aligned} \tag{21}$$

A.3 Adam

Adam is a combination of momentum and RMSProp [57]. It changes the effective learning rate as in RMSProp (Equation (21)), highlighted in blue, but it also uses a momentum to calculate the step size. The step size comes from Equation (20) and is highlighted in red.

$$w_t = w_{t-1} - \frac{\alpha}{\sqrt{r_t}} s_t^m \tag{22}$$

A.4 AdamW

Most, if not all, of the popular machine learning libraries, had an error in their implementation of Adam, which affected the results when Adam was used in combination with *weight decay* [60]. Weight decay is a type of regularisation—i.e. a method that helps to reduce overfitting. The problem was that the weight decay was applied prior to the calculation of the momentum- and RMSProp-parts, which reduced its regularising effects. *AdamW* is simply an implementation of

Adam in which the application of weight decay is postponed until the very end, and, as such, Equation (22) is not altered [60].

B Experiment 0: Bayesian Hyperoptimisation

This experiment, our first, proved to have flaws, which cause its results to be of lesser value. Nonetheless, we include it here, in the appendix, because the experiment provides insights used for subsequent experiments. The two primary flaws of this experiment are (1) the size of the search space we attempt to explore, and (2) the approach with which we explore it. We will examine the flaws in order and lastly show how the algorithm HPs, i.e. the HPs related to training, are selected for our remaining experiments.

B.1 Size of the Search Space

Bonito has with our additions, as explained in Section 4.1, six HPs that can be tuned for each B-block. In this experiment, we allow the HPs for each block to change independently of the models HPs in the other blocks. Since we are testing every combination of a B-layer with all other possible combinations of all other B-layers, the total number of combinations is extremely large. Equation (23) shows the calculation for the total amount of combinations for this setup.

$$T_{combinations} = \prod_{L} \prod_{HP} V_{HP} \tag{23}$$

where L is the five blocks, HP refers to the six different HPs, and V_{HP} is the size of the range for each HP as described in Section 5.2.1. The total amount of combinations for this setup is $1.4e^{20}$. We run 6,000 combinations in this experiment, which is not nearly enough to uncover any tendencies in how the HPs affect each other. Additionally, it is complicated to see the effect of changing a given HP, because one block may scale it up when another scales it down.

To amend the shortcomings of this experiment, we decide to apply the same value for each HP across all blocks in `Exp1`, thereby drastically decreasing the size of the search space.

B.2 Bayesian Hyperoptimisation

To explore the search space we use *Bayesian HyperOptimisation* (BHO). BHO, as explained in [61], is a method to make informed selections of HPs based on previous experiments. Generally, it works by trying to predict what combinations of HPs will maximise or minimise a given metric—e.g. accuracy or loss. BHO excels because it uses a small amount of time to select the configuration compared to how long it takes to train the network. The paper states that BHO outperforms both expert and random selection of HPs. In order to find suitable configurations of HPs, we use BHO for the 6,000 runs. However, we learn that using BHO is counter-productive given that the scope of the project is to investigate the impact—both positive and negative—of each HP. Additionally, BHO uses a single measure for its informed selections, and it is unclear how speed and accuracy can be combined into a single measure that could be used for our research.

To amend this flaw, we, therefore, use a random selection of configurations instead of BHO in `Exp1`.

B.3 Algorithm HPs

Alongside the model HPs, we also test algorithm HPs in this experiment. While we deem the 6,000 configurations to be too few to see tendencies for model HPs, we believe they can be used to see good values for the algorithm HPs. Since the exploration of tendencies in algorithm HPs are out of the scope of this project, we need values that likely will work well for our of our remaining experiments. The following are the relevant algorithm HPs along with the values we choose:

Learning Rate The purpose of a learning rate is described in Section 3.1.1. We find 0.001 to work well in our experiment.

Scheduler A scheduler is used to make changes to the learning rate during training. While it can contribute to a better model, our experiments suggest that when using a scheduler, a lot more epochs are required. To reduce the amount of time need to train each model, we choose not to use a scheduler.

Optimiser As explained in Appendix A, optimisers affects how weights are updated. Our experiments show good results with AdamW, which is also proven to work well for several other domains [57, 60].

Weight decay Weight decay is a regularisation method—i.e. a method that helps to reduce overfitting—which works by multiplying the weights by a positive number less than one on every update. The experiments suggest that a weight decay of around 0.1 yields good results.

Drop out Drop out is, similar to weight decay, a regularisation method. Our experiments suggest that drop out has little to no impact on the overall accuracy, and it, furthermore, slows down training time. This tendency is likely because we use batch normalisation, which allegedly reduces the need for further regularisation [41]. We, therefore, choose not to use drop out.

C Additional figures

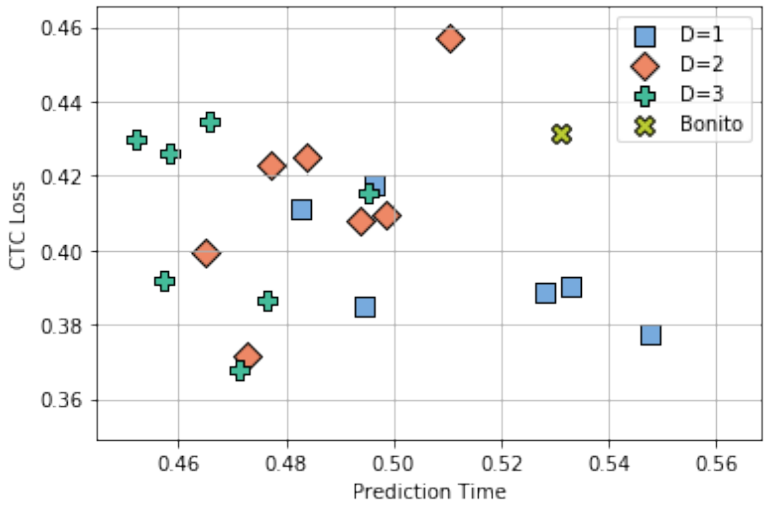


Figure 27: Dilation and Kernel

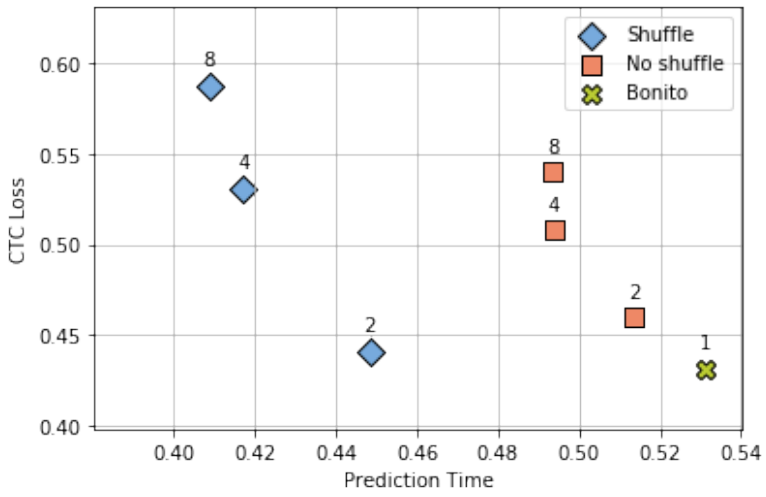


Figure 28: Grouping and Shuffle

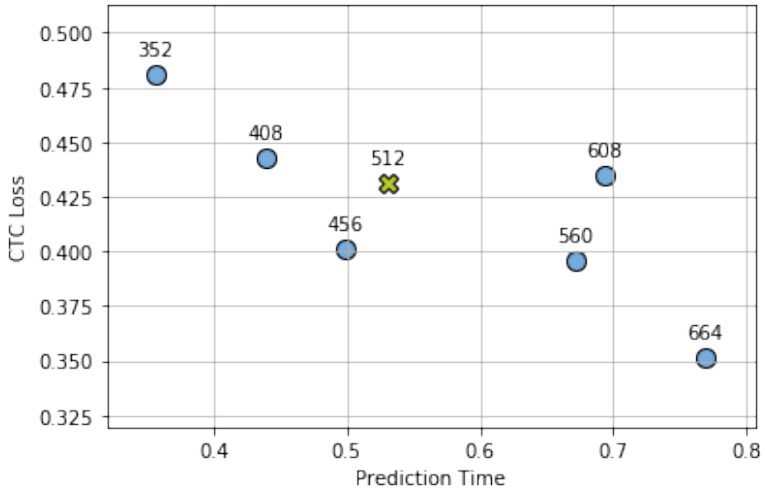


Figure 29: Number of kernels

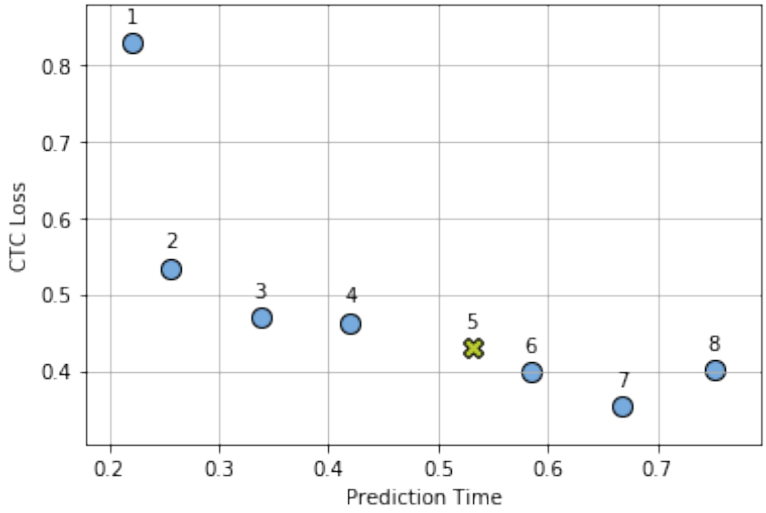


Figure 30: Number of repetitions

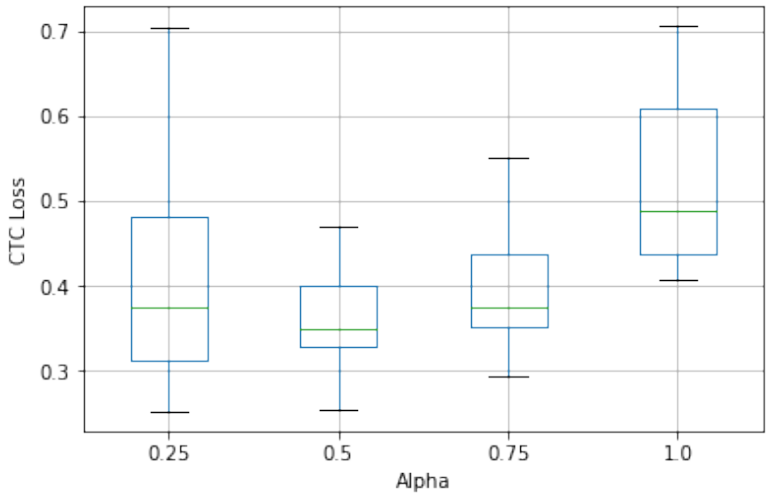


Figure 31: Box plot of the distributing of CTC loss for different values of α

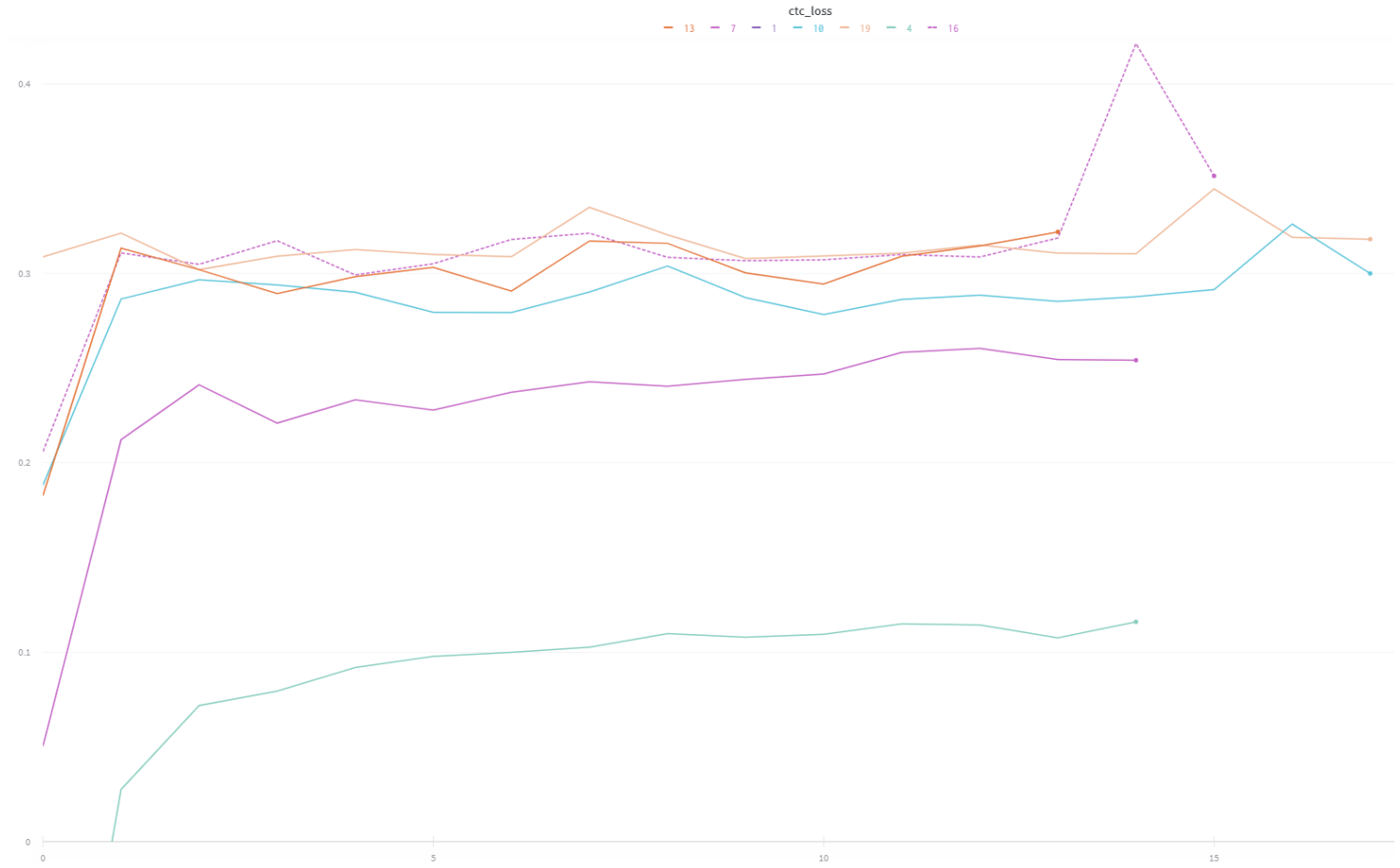


Figure 32: Difference between KD loss and CTC loss on each epoch for different values for temperature