TECHNISCHE
UNIVERSITÄT
WIEN
Vienna | Austria

DIPLOMARBEIT

Sequence Reconstruction in Nanopore Sequencing

Ausgeführt am Institut für
Analysis und Scientific Computing
der Technischen Universität Wien

unter der Anleitung von
Prof. Dr. Clemens Heitzinger

durch

CLEMENS ETL
Grohgasse 5-7/1/7, 1050 Wien

Wien, 14.02.2019

Unterschrift Verfasser	Unterschrift Betreuer

Acknowledgements

First of all, I would like to thank my supervisor Prof. Clemens Heitzinger for giving me the opportunity to be a part of this fascinating project and for supporting me with his advice.

Furthermore, I express my gratitude to my colleagues, on whom I could always rely on and with whom I could cooperate well. They were very helpful and motivating.

Also I want to thank my friends for giving me the backing I needed during my whole study time.

Finally, I want to thank my family, especially my parents Hubert and Maria, for their trust, their support and their patience.

Danksagung

Zuerst möchte ich mich bei meinem Betreuer Prof. Clemens Heitzinger bedanken, der mir die Möglichkeit gab, an diesem faszinierenden Projekt mitzuwirken und mich stets mit seinen Ratschlägen unterstützte.

Des Weiteren bedanke ich mich bei meinen Studienkollegen, die sehr hilfsbereit waren, auf die ich mich immer verlassen konnte und mit denen ich gut zusammenarbeiten konnte.

Ein großer Dank gilt auch meinen Freunden, die mir den nötigen Rückhalt während meiner Studienzeit gaben.

Zu guter letzt möchte ich mich herzlich bei meiner Familie bedanken, speziell bei meinen Eltern Maria and Hubert, für ihr Vertrauen, ihre Unterstützung und ihre Geduld.

Abstract

This master thesis is about nanopore sequencing. In this method, a single-stranded DNA oligomer is pulled through a tiny pore. Electric current flows through the pore and is modulated to different degrees by the different bases contained therein. By measuring the current one can draw conclusions on the DNA sequence. This process is called *basecalling*.

The goal of this thesis is to develop, implement and evaluate algorithms for basecalling. Currently this value is about 80% for a single read. By sequencing the same section multiple times an accuracy of over 99% can be reached. In order to develop a *basecaller* bidirectional *recurrent neural networks* are used in this thesis. In addition to their implementation, the optimal hyperparameters, e.g. the size and number of layers, the optimizer, the loss function, etc. are determined.

To train the RNN a training dataset must be created first. For each read, the corresponding section in the reference sequence must be determined in order to assign the actual bases to the read. Since the bases translate the pore at different speeds, it is necessary to first determine from the raw data when a base reaches the pore. Therefore a *break point detection* is applied. This method detects when the current changes significantly. The method used for this work is a window-based break point detection algorithm, which is characterized by its high speed.

The evaluations of the test data obtained have shown that the precision of the developed basecaller does not exceed the precision of the basecaller Metrichor, supplied by Oxford Nanopore. An improvement could be achieved by using a different break point detection algorithm.

Zusammenfassung

In dieser Diplomarbeit geht es um Nanopore-Sequenzierung. Bei dieser Methode wird ein einsträngiges DNA-Oligomer durch eine winzige Pore gezogen. Elektrischer Strom fließt anschließend durch diese Pore und wird durch die darin befindlichen Basen unterschiedlich stark moduliert. Wird dieser Strom gemessen, so kann man anhand der Messwerte auf die DNA-Sequenz zurückrechnen. Dieser Vorgang wird *Basecalling* genannt.

Das Ziel dieser Diplomarbeit ist es, Algorithmen für das Basecalling mit einer möglichst hohen Genauigkeit zu entwickeln, zu implementieren und auszuwerten. Derzeit liegt diese bei ungefähr 80% für einen einzelnen Read. Durch mehrfaches Sequenzieren desselben Abschnittes können Genauigkeiten von über 99% erreicht werden. Um den Basecaller zu entwickeln, werden in dieser Arbeit bidirektionale *rekurrente neurale Netze* (kurz RNN) verwendet. Neben deren Implementierung müssen zusätzlich die optimalen Hyperparameter, wie z.B. die Größe und Anzahl der Schichten, der Optimierer, die Verlustfunktion etc. bestimmt werden.

Um das RNN trainieren zu können, muss zuvor ein Trainings-Datensatz erstellt werden. Dafür muss für jeden Read der zugehörige Abschnitt in der Referenzsequenz ermittelt werden, um die tatsächlichen Basen dem Read zuordnen zu können. Da die Basen mit unterschiedlichen Geschwindigkeiten durch die Pore wandern, muss man zuvor anhand der Rohdaten feststellen, wann eine Base die Pore erreicht. Dazu wird eine *Break Point Detection* durchgeführt. Diese erkennt, wenn sich ein Signal signifikant ändert. Als Methode wurde für diese Arbeit ein *Window-based Break Point Detection*-Algorithmus verwendet, der sich durch seine hohe Geschwindigkeit auszeichnet.

Die Auswertungen der erhaltenen Testdaten haben gezeigt, dass die Präzision des erstellten Basecallers die des von Oxford Nanopore mitgelieferten Basecallers Metrichor nicht übersteigt. Durch die Verwendung eines anderen Break Point Detection-Algorithmus könnte eine Verbesserung erzielt werden.

Contents

Chapter 1

Introduction

1.1 The History of Nanopore Sequencing

In the late 80's David Deamer, a biophysicist, had the idea to sequence DNA by pulling it through a tiny pore and measuring the electric current that flows through the pore, see Figure 1.1. This is the principle of a Coulter counter. Deamer's expectation was that the bases block the pore differently, depending on their shapes and sizes. He believed that when ions pass through the pore at the same time, the ionic is modulated. By measuring the ionic current, one can draw conclusions on the DNA that passes through. In today's nanopores at least five bases influence the measurements at a time. Since DNA consists of four different bases – cytosine, guanine, adenine and thymine – there are 1024 possible arrangements, called pentamers, which require a high precision of the measurements.

The pores were made of a protein isolated from the bacterium Staphylococcus aureus called α-hemolysin. Seven units are arranged in a circle with a hole in the middle. Several thousands of these pores are placed on a membrane, which increases the speed of sequencing. An enzyme, which binds to the end of the single-stranded DNA, is used to control the speed of the strand that passes through. When it is attached to the pore, the DNA is split into two strands and is fed to the pore.

This idea took Deamer and the Harvard University cell biologist Daniel Branton decades of research to develop the first prototype for nanopore sequencing [7, 18, 12]. The difficulty was to find a pore with an appropriate size and charging state to control the speed of the DNA while it passes through and to measure the electric current accurately. Years later the idea was patented by Deamer, Branton and others.

Almost 30 years later this technology was licensed by the company Oxford Nanopore Technologies. They developed a device called MinION, which is able to read DNA, RNA and protein samples and has the size of a modern mobile phone. There are also larger models, such as PromethION and

GridION, which are able to sequence several DNA probes at a time. They all use the same principle. All of these devices can be connected to a computer via USB. The great advantage of this method is its ability to observe reads with a length of up to two million bases. This is impressive compared to conventional chemical methods, which can only decode short stretches of DNA with a length of about 200 bases. As a result, long calculation times can be saved, since piecing short strands together is computationally very complex. It also enables reading DNA sequences with many repetitions and copy-number variations. Moreover, this method is faster than chemical methods.

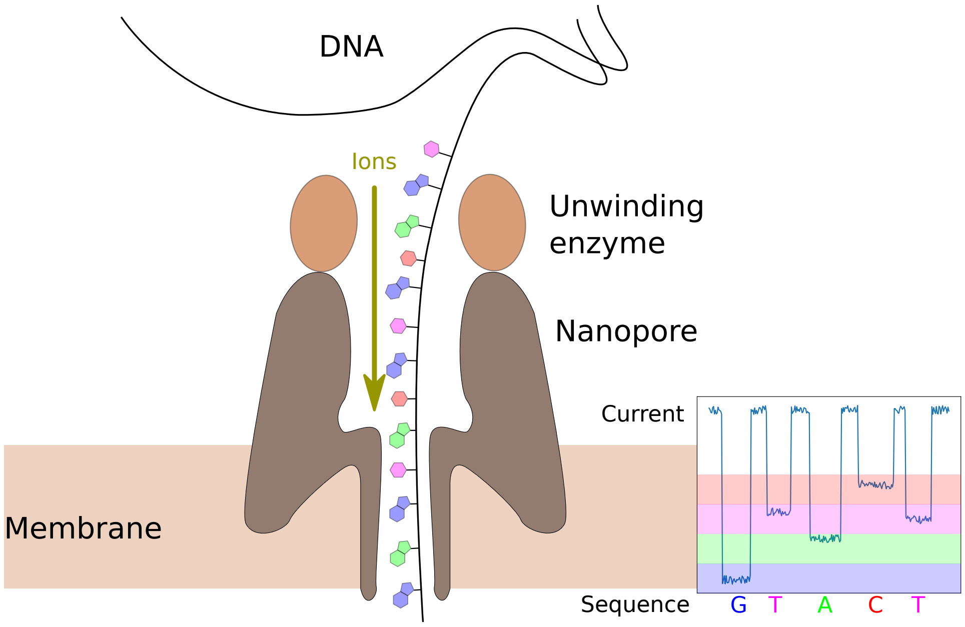

Figure 1.1: Nanopore.

1.2 Recurrent Neural Networks

The average accuracy of a single read is about 80%. Since most sections of the DNA are sequenced multiple times, the accuracy can be increased to over 99% by assembling the reads [9]. Nevertheless, this technology is still under development and will be improved.

The main purpose of this master thesis is to improve the process of converting the electric signal into the actual sequence, called basecalling. There are different approaches for this purpose, such as hidden Markov models. We decided to use an approach from machine learning. More precisely, we used recurrent neural networks (RNN), since they are suitable for varying input lengths. An RNN consists of several neurons, which are connected to each other. During the learning process some of these connections become stronger. The more the RNN is trained, the more accurate it becomes. In order to develop a functioning network, an appropriate training set has to be created. This leads us inter alia to break point detection algorithms.

This thesis is structured as follows:

- **Chapter 2** delivers the theoretical background to create a basecaller. First, *break point detection* algorithms are presented. They are used to determine when a new base has reached the pore. Depending on the software used, the break point detection is already performed during the sequencing. If this is not the case, one has to perform it oneself. In this case the *window-based break point detection* algorithm, which is known for its speed, is used. It is also explained how to create an RNN and the needed training datasets. Finally, the training process and the basecalling are discussed.

- In **Chapter 3** the numerical results are presented. The effects of the various parameter settings are compared in several charts and tables. Also, both the accuracy of the RNN and the functionality of the break point detection were tested.

- **Chapter 4** includes conclusions about the results and provides approaches for further improvement.

Chapter 2

Methods

This chapter presents the theoretical background that is necessary to develop a basecaller.

2.1 Break Point Detection

The electric current in the pore is represented by a raw signal at every point in time during the whole read. The number of measurements per second is defined by the sampling rate. The duration a base is located in the pore is variable, but can be detected since the current changes immediately with each base. For this purpose break point detection algorithms are required. These algorithms search through the raw data for significant changes.

2.1.1 Raw Signal

The raw signal is digitized and therefore needs to be converted to the actual value of the measured current. For this we need the offset, range and digitization, which are stored in the meta data. All these parameters are stored in HDF5 files, together with the raw signal and the break point detection. The digitization represents the number of possible output values. The range is the difference between the smallest and greatest values. By adding the offset to the raw signal, we calculate the actual current as

$$\mathrm{raw}_{\mathrm{real}} = (\mathrm{raw}_{\mathrm{digitized}} + \mathrm{offset}) \cdot \frac{\mathrm{range}}{\mathrm{digitization}}. \tag{2.1}$$

Since there are many reads, where the first base reaches the pore some time after the start of the recording, it is necessary to evaluate the moment when the actual sequencing starts.

2.1.2 Finding the Sequencing Start

As evident from Figure 2.1, the measured current peaks at the very beginning of the read. This occurs due to the fact that no base has yet reached the pore. The value drops immediately when the first base passes through, which can be explained by the resistance of the base. This drop is the *initial event*.

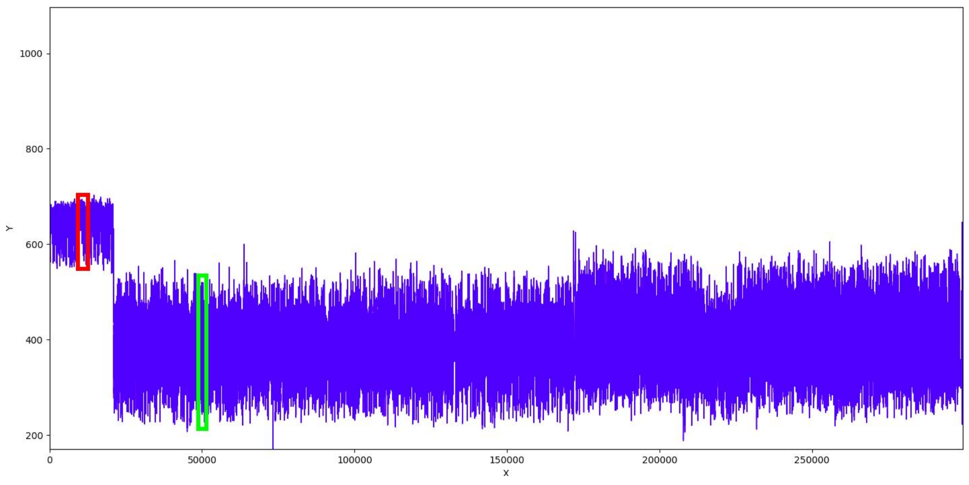

Figure 2.1: Plot of the raw data. The current is higher at the beginning of the read. It drops when the first base passes the pore.

Taking a closer look at the region marked with the red rectangle (Figure 2.2), it becomes obvious that there is no significant signal, but only noise. In the region marked with the green rectangle (Figure 2.3) a lot of jumps in the data can be seen. Those jumps are caused by bases reaching the pore and thereby changing the resistance drastically.

Figure 2.4 shows the possibility of reads which start already with a base in the pore. There is only a short period of noise, as one can see in the region marked in red (Figure 2.5).

Algorithm 1 can be used to find the first event in the raw data. First, the digitized data is converted to the actual current and the whole data is split into 200 equally sized intervals. Afterwards we evaluate the mean of each interval and append those values to the list `mean`. To determine in which interval the jump has occurred, we calculate the average between the maximum and the minimum of the mentioned list and save it as `mid`. Then we search for all intervals that have a greater mean than `mid`. We assume that a jump has happened when the whole read can be split into two regions of intervals, whereby the first region is above `mid` and the second one is below. If this is not the case we assume the read starts from the beginning. In this case, we set $\text{raw}_{\text{start}}$ to zero. Next, we search for the exact position of the initial event. Therefore we take a closer look at the two intervals, where the crossing below `mid` has happened. Then a window with a length of 50 is slid through both of them. Simultaneously we evaluate the mean of this

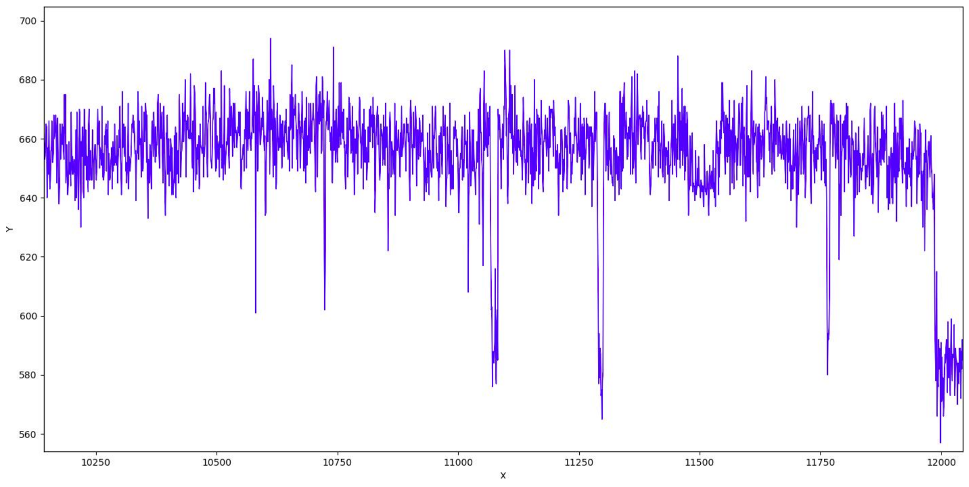

Figure 2.2: Zoom in Figure 2.1, marked by the red rectangle. No base has yet reached the pore. Thus there is only noise.

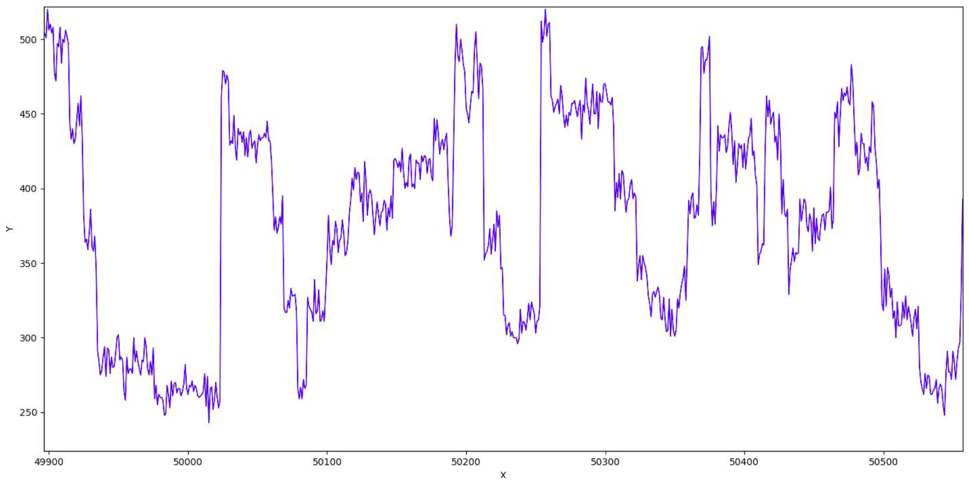

Figure 2.3: Zoom in Figure 2.1, marked by the green rectangle. Every time a new base enters the pore the value changes rapidly.

window. If the mean falls below `mid`, we set $\text{raw}_{\text{start}}$ to the current position plus the window length.

2.1.3 Overview of Break Point Detection Algorithms

There are several break point detection algorithms available. Some of them are listed below [21, 15, 5].

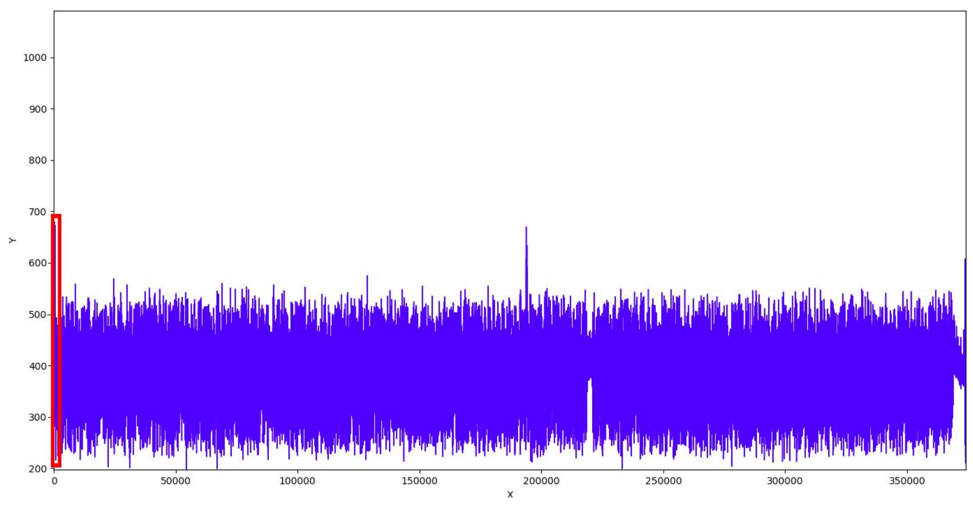
Figure 2.4: Another plot of the raw data. In this case the level of the read seems to be constant.

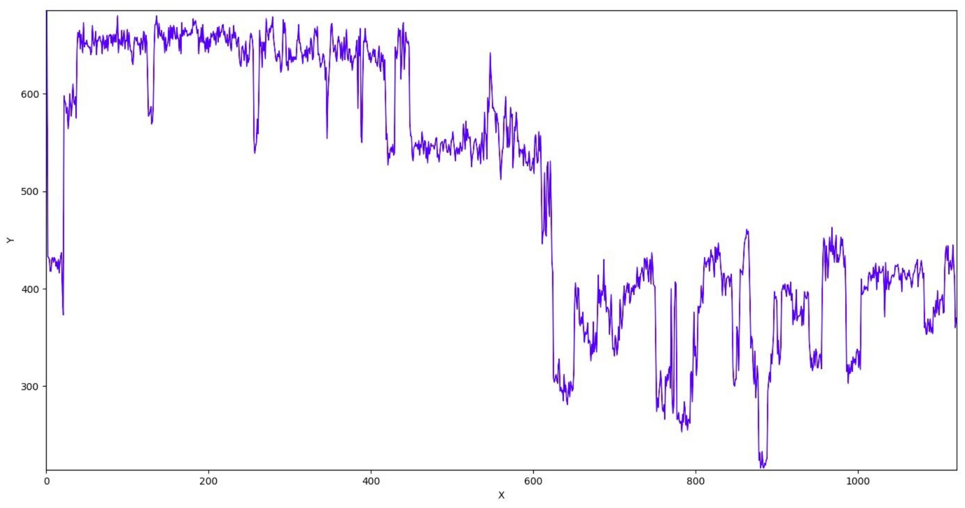
Figure 2.5: Zoom in Figure 2.4. After a short period of noise the first base translocates the pore.

PELT: PELT stands for "Pruned Exact Linear Time" and is used when the total number of break points is unknown, as in our case. The goal of this algorithm is to minimize the sum of the objective function $V(\mathcal{T}, y_{0\ldots T}) = \sum_{k=0}^{|\mathcal{T}|} c(y_{t_k \ldots t_{k+1}})$ and the penalty function $\mathrm{pen}(\mathcal{T}) = \beta|\mathcal{T}|$, where $\mathcal{T}$ denotes the break points, c the cost function and $\beta > 0$ the smoothing parameter. The complexity of this algorithm can be

Algorithm 1 Finding the initial event

```
raw ← $(\text{raw}_{\text{digitized}} + \text{offset})\frac{\text{range}}{\text{digitization}}$
mean ← empty list
for n ← 1 to 200 do
    mean append MEAN OF(raw(length(raw)$\frac{n}{200}, \ldots,$ length(raw)$\frac{n+1}{200}$))
mid ← $\frac{\max(\text{mean})+\min(\text{mean})}{2}$
$\text{raw}_{\text{start}}$ ← 0
fail ← False
for n ← 1 to 200 do
    if $n > 0$ and $\text{raw}_{\text{start}} \neq n - 1$ then:
        fail ← True
    $\text{raw}_{\text{start}} \leftarrow n$
if fail then:
    $\text{raw}_{\text{start}}$ ← 0
else:
    $\text{raw}_{\text{start}} \leftarrow \text{raw}_{\text{start}} \cdot \text{length(raw)}/200$
for $n \leftarrow \text{raw}_{\text{start}}$ to $\text{raw}_{\text{start}} + \text{length(raw)}/200 \cdot 2$ do
    if mean(raw($n, \ldots, n + 50$)) < mid then
        $\text{raw}_{\text{start}} \leftarrow n + 50$
        Leave the for loop
return $\text{raw}_{\text{start}}$
```

reduced to $\mathcal{O}(T)$ by using the pruning rule, which discards certain points from the potential break points.

Binary Segmentation: This algorithm can be used for a fixed number of break points and when the number is unknown. The principle is simple: The first break point $\hat{t}^{(1)}$ is given by

$$\hat{t}^{(1)} := \text{argmin}_{1 \leq t < T-1} c(y_{0 \ldots t}) + c(y_{t \ldots T}). \tag{2.2}$$

This break point splits the whole signal into two intervals, where we repeat this process until a stopping criterion is met, see Figure 2.6. This process has a complexity of $\mathcal{O}(T \log T)$.

Bottom-up Segmentation This algorithm starts by splitting the signal into equally sized intervals. Afterwards some of the intervals are merged gradually, depending on the distance function $d(.,.)$ of two adjoining intervals, whereby $d(.,.)$ is defined as

$$d(y_{a \ldots t}, y_{t \ldots b}) = c(y_{a \ldots b}) - c(y_{a \ldots t}) - c(y_{t \ldots b}). \tag{2.3}$$

A visualization of this process is illustrated in Figure 2.7. This algorithm has a complexity of $\mathcal{O}(T \log T)$ and can also be used for both, known and unknown numbers of break points, depending on the stopping criterion.

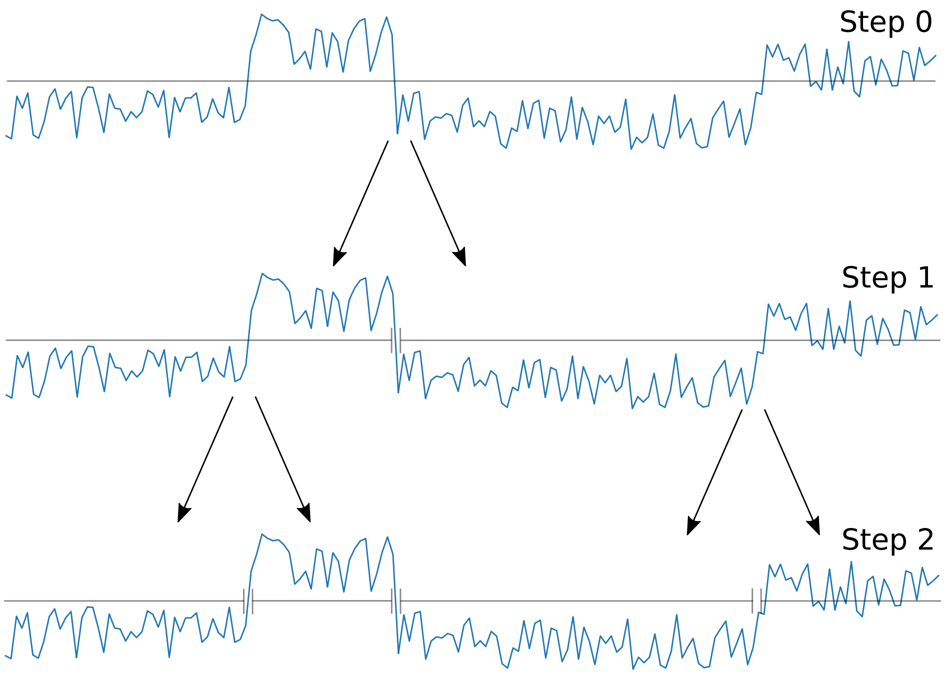

Figure 2.6: Binary Segmentation.

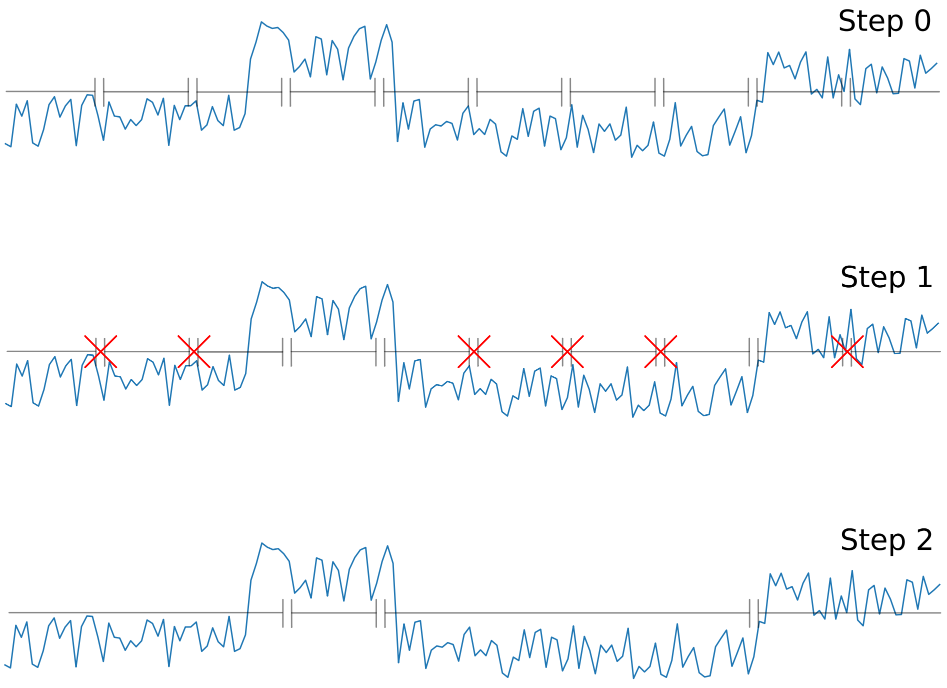

Figure 2.7: Bottom-up segmentation.

2.1.4 Window-based Break Point Detection

Since some reads can be of considerable length, we choose the *window-based break point detection* algorithm, which is the algorithm with the lowest computation time ($\mathcal{O}(T)$), see Figure 2.8. More precisely we use an adaptation of said algorithm (Algorithm 2) with two windows, a short and a long one, suggested by the Deepnano project [1].

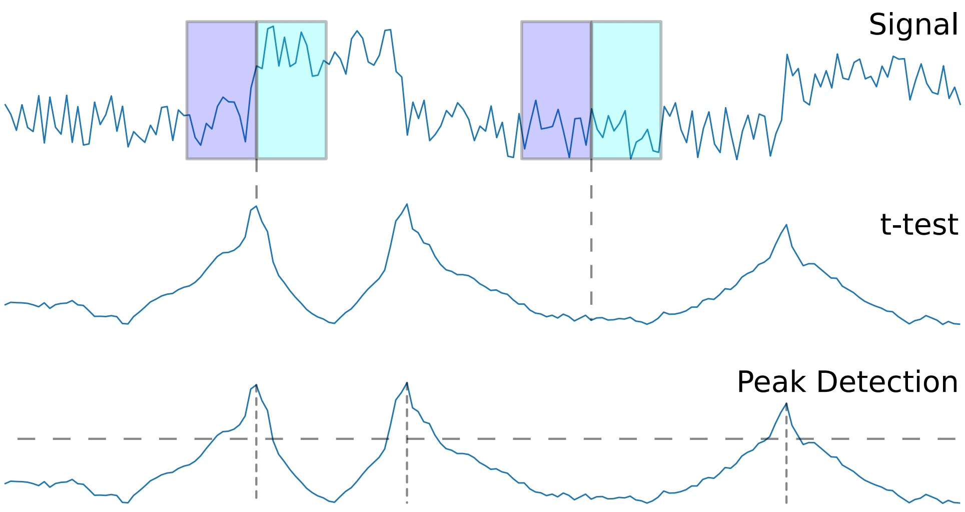

Figure 2.8: Window-based break point detection algorithm.

The first step of the algorithm is to define the window lengths and thresholds. In the for loop, we set two windows side by side at the beginning of the read. The number of the contained raw data points is equal to the `short_window_length`. Next, we calculate the mean and variance of both windows. In the if condition we run a t-test between those windows. In order to perform the t-test we need to determine the difference between the means and variances. This is an appropriate way to determine whether there is a significant change of the signal. Next, we repeat the same procedure for the long window. Then we use the t-test results to find the break points by calling the *find peak positions* algorithm (Algorithm 3), which we will discuss in the next paragraph. Finally, we evaluate and return the means, variances and lengths of the events.

To determine the locations of the break points, `find_peak_positions` verifies if the t-test exceeds a certain threshold. After initialization the function starts a for loop, which runs through the whole data of the t-test. After one iteration we repeat this process, except for the if condition marked with a comment. This ensures that only one window is detecting a certain break point. The function has two possible states, depending on whether `short_peak_pos` is set to `NO_PEAK_POS`. In this case, the function constantly checks if the t-test exceeds the threshold. Once the threshold is exceeded, `short_peak_val` and `short_peak_pos` are set to the current values. Since short_peak_pos is no longer equal to `NO_PEAK_POS` the else condition is fulfilled. In this state `short_peak_pos` and `short_peak_val` are constantly updated until the latter reaches its maximum. Simultaneously the `short_found_peak` flag is set true. If there is no further increase for the next half `short_window_length`, `short_peak_pos` will be saved as a break point and the function falls back into the initial state.

Algorithm 2 Window-based Break Point Detection

```
Require: short_window_length, long_window_length, short_threshold,
  long_threshold
  for counter ← 1 to length of raw_data − window_length do
      short_window(counter) ← raw_data(counter, ..., counter
                                  +short_window_length)
      mean ← get mean of short_window
      variance ← get variance of short_window
  if counter > short_window_length then
      Δmean(counter) ← mean(counter) − mean(counter
                            +window_length)
      Δvariance(counter) ← (variance(counter) − variance(counter
                            +window_length)) $\frac{1}{\text{window\_length}}$
      t_statistics_short(counter) ← $\left|\frac{\Delta\text{mean(counter)}}{\sqrt{\Delta\text{variance(counter)}}}\right|$
  Repeat for-loop for long_window
  event_positions ← FIND_PEAK_POSITIONS(t_statistics_short,
      t_statistics_long, short_window_length, long_window_length,
      short_threshold, long_threshold)
  Evaluate means, variances and lengths of events
  return position, length, mean and standard deviation of events
```

2.1.5 Window Lengths and Thresholds

Choosing the optimal window lengths and thresholds is essential for an operational break point detection and furthermore for training and sequencing. Our goal is to set the break points so that exactly one base per event goes through the pore as often as possible. Another important factor is to avoid more than two bases per event, since this error cannot be corrected during basecalling. Therefore we have to write an appropriate algorithm to find the ideal window lengths and thresholds, see Algorithm 4. We assume that we know the actual break points of the reads. They can be evaluated by using the break point detection of Metrichor [2]. First, we create a list named `val`. The first two entries define the short and large window length, the last two

Algorithm 3 Find peak positions

```
NO_PEAK_POS ← $-1$
NO_PEAK_VAL ← $10^{100}$
long_mask ← 0
short_peak_pos, long_peak_pos ← NO_PEAK_POS
short_peak_val, long_peak_val ← NO_PEAK_VAL
short_found_peak, long_found_peak ← False
peaks ← empty vector
for $i$ ← 1 to length of short_data do
    val ← short_data[$i$]
    if short_peak_pos = NO_PEAK_POS then
        if val < short_peak_val then
            short_peak_val ← val
    else if val > short_peak_value then
        short_peak_val ← val
        short_peak_pos ← i
    else
        if val > short_peak_val then
            short_peak_pos ← i
            short_peak_val ← val
                    ▷ Don't repeat the following if condition for long-values
        if short_peak_val > short_threshold then
            long_mask ← short_peak_pos + short_window
            long_peak_pos ← NO_PEAK_POS
            long_peak_val ← NO_PEAK_VAL
            long_found_peak ← False
        if short_peak_val > val and
            short_peak_val > short_threshold then
              short_found_peak ← True
        if short_found_peak and
            (i − short_peak_pos) > short_window / 2 then
              peaks append short_peak_pos
              short_peak_pos ← NO_PEAK_POS
              short_peak_val ← val
              short_found_peak ← False
    if i > long_mask then
        Repeat code of For-loop for long-values, except one If-condition
return peaks
```

their thresholds. Then we generate random values for them. In the for loop we load the start and end positions of the raw data. The first four entries of the list `counter` report how many bases have passed the pore between two calculated break points. $\texttt{counter}_1$, $\texttt{counter}_2$ and $\texttt{counter}_3$ stand for zero, one and two or more bases per event. If there are more than two, the surplus bases are written to $\texttt{counter}_4$. $\texttt{counter}_5$ represents the number of events and $\texttt{counter}_6$ the number of bases in total. In every step we change one parameter randomly. Then we take a file sample and evaluate each parameter of `counter` by calling the function `count_bases`. Finally, we calculate a score for our values. If this score reaches a new maximum, we write it to `max_score`, otherwise `val` will be discarded.

Algorithm 4 Window Lengths and Thresholds

```
max_score ← $-100$
val ← $(0,0,0,0)$
$\text{val}_1$ ← random integer in $[3,10]$
$\text{val}_2$ ← random integer in $[7,14]$
$\text{val}_3$ ← random float in $[0,10)$
$\text{val}_4$ ← random float in $[0,10)$
for $q \leftarrow 1$ to number of loops do
    old_val ← val
    raw_start, raw_end ← Load start and end positions in raw data
    counter ← $(0,0,0,0,0,0)$
    if $q \mod 4 = 0$ then
        $\text{val}_1$ ← random integer in $[3,10]$
    else if $q \mod 4 = 1$ then
        $\text{val}_2$ ← random integer in $[7,14]$
    else
        $\text{val}_{q \bmod 4}$ ← random float in $[0,10)$
    for file in file_sample do
        window_lengths ← $(\text{val}_1, \text{val}_2)$
        thresholds ← $(\text{val}_3, \text{val}_4)$
        break points ← BREAK_POINT_DETECTION(file, raw_start(file),
                                raw_end(file), window_lengths, thresholds)
        real_bp ← get_real_bp(file)
        counter ← counter + count_bases(real_bp, break points)
    score ← $\frac{\text{counter}_1}{\text{counter}_4} - \frac{\text{counter}_0}{\text{counter}_4} - 2.5 \cdot \frac{\text{counter}_2}{\text{counter}_4} - 5 \cdot \frac{\text{counter}_3}{\text{counter}_5}$
    if score > max_score then
        max_score ← score
    else
        val ← old_val
return val
```

2.2 Recurrent Neural Networks

Since the reads have variable lengths, an appropriate machine learning algorithm is needed. For this purpose *Recurrent Neural Networks* (RNN) [19, 8] are suitable. The goal of an RNN is to find an accurate function, which transforms the events to the corresponding bases. An RNN consists of an input layer, several hidden layers and one output layer. The length t of each layer is equal to the length of the read, see Figure 2.9.

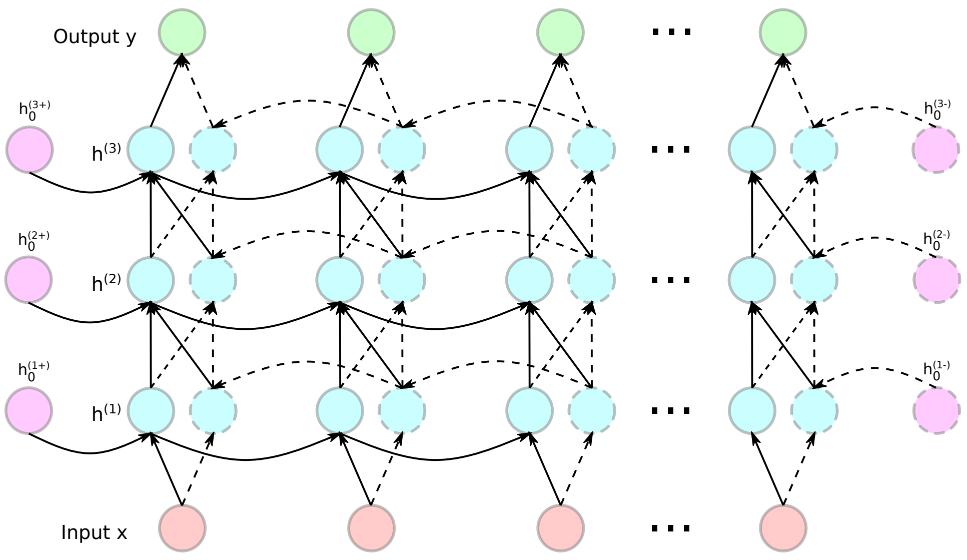

Figure 2.9: Structure of an RNN.

Input layer: The input vectors $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_t$, are transferred to the input layer. In our case one vector contains the mean, standard deviation and length of an event.

Hidden layer: A typical RNN has multiple hidden layers $\boldsymbol{h}^{(j)}$. Every hidden state $\boldsymbol{h}_t^{(j)}$, also called neuron, represents a vector with a certain size. First, we describe a one directional RNN. In this case to obtain $\boldsymbol{h}_t^{(j)}$ we need a function f_j that depends on the neuron of the previous layer $\boldsymbol{h}_t^{(j-1)}$ and the previous neuron in the current layer $\boldsymbol{h}_{t-1}^{(j)}$. For the first hidden state in each layer there is an initial state $\boldsymbol{h}_0^{(j)}$, which is a parameter of the model. For the function f_j we use the *Long Short-Term Memory* unit (LSTM).

Definition 2.1 (Long short-term memory (LSTM)) *Let $\boldsymbol{x}_t \in \mathbb{R}^d$ be the input vector of the LSTM unit, $\boldsymbol{h}_t \in \mathbb{R}^h$ the output vector, W_f, W_i, W_o, $W_c \in \mathbb{R}^{h \times d}$ and U_f, U_i, U_o, $U_c \in \mathbb{R}^{h \times h}$ the weight matrices, and $\boldsymbol{b}_f$, $\boldsymbol{b}_i$, $\boldsymbol{b}_o$, $\boldsymbol{b}_c \in \mathbb{R}^h$ the bias vectors, where d denotes the size of the previous layer, and h the current layer size. The gates and the cell state are defined as*

- *forget gate* $f_t = \sigma(W_f \boldsymbol{x}_t + U_f \boldsymbol{h}_{t-1} + \boldsymbol{b}_f)$,
- *input gate* $i_t = \sigma(W_i \boldsymbol{x}_t + U_i \boldsymbol{h}_{t-1} + \boldsymbol{b}_i)$,
- *output gate* $o_t = \sigma(W_o \boldsymbol{x}_t + U_o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o)$,
- *cell state* $c_t = f_t \circ c_{t-1} + i_t \circ \tanh(W_o \boldsymbol{x}_t + U_o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o)$,

where $\circ$ denotes the element-wise product and σ the sigmoid function. Finally, the output vector $\boldsymbol{h}_t$ is given by

$$\boldsymbol{h}_t = o_t \circ \tanh(c_t).$$

The weight matrices U and W, the bias vectors $\boldsymbol{b}$ and the initial state $\boldsymbol{h}_0$ are also parameters of our network and will be trained.

Since the output vector is also influenced by the following outputs, we need a bidirectional RNN. Therefore two output vectors will be calculated, one in forward and one in backward direction. The concatenation of those yields the input vector of the next layer.

Output layer: Ideally, each input event leads to exactly one base, but in some cases more than one base passes through the pore during one event. Thus we need at least two outputs. It is also possible, that there are three or more bases, but this happens very rarely and for this reason was left unattended. For every output there are five possibilities: A, C, G and T denote the four bases and N denotes no base. In case of one base per event, the base in question is allocated to the second output. The probabilities for the four bases, respectively blank, are calculated by the softmax function

$$P(\text{out}_i^{(k)} = q) = \frac{\exp(W_i \circ \boldsymbol{h}_i^{(n)} + \boldsymbol{b}_i)}{\sum_{i=1}^{5} \exp(W_i \circ \boldsymbol{h}_i^{(n)} + \boldsymbol{b}_i)}, \tag{2.4}$$

where $\boldsymbol{h}_i^{(n)}$ is the output vector of the last hidden layer, $k \in \{1, 2\}$, $W \in \mathbb{R}^{d \times 5}$ and $b \in \mathbb{R}^5$. W and b are also trainable parameters.

The RNN returns two 5-dimensional vectors for each event, where every element stands for the probability of one base, or a blank.

2.3 Preparation of Training Data

To train our model a set of training data is necessary. The reads of those data sets are usually stored in HDF5 files, containing the raw data, break point detection and output of the Oxford Nanopore Technologies' basecaller Metrichor. We use this output to determine the region of the reference sequence to which the read belongs to. For that, we use the aligner Minialign [3] (Algorithm 5).

The output at the beginning and ending of the read is often inaccurate, which is why it cannot be aligned. Therefore Minialign returns the start and

Algorithm 5 Prepare dataset

```
for all reads do
    $(\text{read\_start\_pos}, \text{read\_end\_pos}, \text{ref\_start\_pos},$
      $\text{ref\_end\_pos}, \text{cigar\_string}) \leftarrow$ MINIALIGN(read, reference sequence)
    scale events of read to normal distribution $\mathcal{N}(0, 1)$
    Data append LOAD_DATA(read, read_start_pos, read_end_pos,
                          ref_start_pos, ref_end_pos, cigar_string)
split Data into trainData, validationData and testData
return events with the corresponding bases
```

end position in the read and in the reference, as well as the accuracy and the cigar string. The cigar string describes how the read aligns with the reference. It consists of a series of sets, containing a number followed by one letter, whereby the letter denotes an operator and the number stands for the quantity of those. Here a short example: `49S3M1D3M1D8M1I23M1D4M1D9M1D6M...`

The meaning of the operators are shown in Table 2.1.

Operator	Description
D	Deletion: the nucleotide is present in the reference, but not in the read.
H	Hard clipping: the clipped nucleotides are not present in the read.
I	Insertion: the nucleotide is present in the read, but not in the reference.
M	Match: can be either an alignment match or mismatch. The nucleotide is present in the reference.
S	Soft clipping: the clipped nucleotides are present in the read.

Table 2.1: Meaning of the cigar string operators.

2.3.1 Finding the Initial Event

In the next step, we will use this string to allocate each event to the corresponding bases of the reference. Since the basecaller of Metrichor sometimes returns two or no bases, the start and end position of the output is not the same as the one of the events. Thus, first we have to find the starting event (Algorithm 6).

Along with the mean, standard deviation and length, there are also a few more bits of information about every single event. They are stored in an array of the HDF5 files. Most of them affect the hidden Markov model that Metrichor works with and are thereby irrelevant for our problem. However, there are also useful ones such as the column `move`, which shows us how many bases a single event Metrichor is outputting. According to this, we have to go over each event and sum up their moves, until the sum reaches the start and end position of the read. At this point it must be stated that Metrichor returns pentamers, but we only return single bases. Thus if we use the base in the middle of the pentamer as output, we have to shift the read position by two.

Algorithm 6 Find starting event

```
eventPos ← 2
foundStart ← False
for $n$ ← 1 to length of events do
    eventsPos ← eventPos + events[move]($n$)
    if eventPos $\geq$ read_start_pos and not foundStart then
        event_start_pos ← $n$
        foundStart ← True
    if eventPos $\geq$ read_end_pos then
        event_end_pos ← $n$
        break
return event_start_pos, event_end_pos
```

2.3.2 Allocating the Corresponding Bases to the Events

Now that we found the appropriate events, we can allocate them to the bases of the reference sequence (Algorithm 7). Therefore the output of Metrichor becomes obsolete and we continue using only its moves. We always store two bases per event in the array read. Additionally we add a list to each base, an explanation for that will follow in the next section. At first we set pointers for the events, reference and cigar string to the right position (concerning the cigar string: to simplify the code, we assume n letters in a row instead of a number n followed by a letter, e.g. `MMMMDD` instead of `4M2D`). Now we have to go over every single event. Depending on their moves, there are three different cases:

Case 1: move = 0**:** In this case we store two N's.

Case 2: move = 1**:** Again we store an N for output 1, but for output 2 there are another three cases to consider.

Case 2.1: cigar_string[cigPos] = M**:** The next base in the reference will be allocated.

Case 2.2: cigar_string[cigPos] = D**:** At this point the list, that has already been mentioned before, becomes relevant. The reference pointer

Algorithm 7 Allocating the Corresponding Bases to the Events

```
event_pos ← event_start_pos
ref_pos ← ref_start_pos
cig_pos ← 0
if move($n$) = 0 then
    read append [$N$, []]
    read append [$N$, []]
else if move($n$) = 1 then
    read append [$N$, []]
    if cigar_string[cigPos] = $M$ then
        read append reference[refPos]
        refPos ← refPos + 1
    else if cigar_string[cigPos] = $D$ then
        noDels ← number of D's until next M in cigar_string
        dels ← reference[refPos, refPos + noDels]
        refPos ← refPos + noDels
        cigPos ← cigPos + noDels
        read append [reference[refPos], dels]
        refPos ← refPos + 1
    else if cigar_string[cigPos] = I then
        read append [$N$, []]
    cigPos ← cigPos + 1
else
    for $n$ ← 1 to 2 do
        if cigar_string[cigPos] = $M$ then
            read append reference[refPos]
            refPos ← refPos + 1
        else if cigar_string[cigPos] = $D$ then
            noDels ← number of D's until next M in cigar_string
            dels ← reference[refPos, refPos + noDels]
            refPos ← refPos + noDels
            cigPos ← cigPos + noDels
            read append [reference[refPos], dels]
            refPos ← refPos + 1
        else if cigar_string[cigPos] = $I$ then
            read append [$N$, []]
        cigPos ← cigPos + 1
return read
```

jumps to the next base that has a match. The skipped bases are added to the list and if possible, they can later be inserted to the output.

Case 2.3: cigar_string[cigPos] = I: No base will be allocated. Finally, the pointer has to be set to the correct positions.

Case 3: move = 2: Similar to case 2, but this time we make the distinction for output 2 as well.

Finally, we insert the deleted bases from our lists to the output (Algorithm 8). Before doing that we add the bases without the lists to a string called `line`. Now let us take a closer look at one base and its list. There are two key aspects to consider: First, we only change N's into bases, and secondly, we only change them if there is no other base in between. Otherwise we would mix up the order of the output. Thus, we count the N's between the current and the last base in the read. In case of having more blanks than bases that should be inserted, we add them to the second output first. Those are the positions in the read that have an even number. The determination of these positions takes place in the first inner for loop. Then we take this position array to insert the deleted bases to the right spots of the line. Finally, we split the line into sets of two letters and write them to the output.

Algorithm 8 Creating the dataset

```
for all Events $e$ in read do
    line append $e[0]$
    if $e[1]$ is not empty then
        num $\leftarrow$ numbers of $N$s between $e$ and last output which is no $N$
        pos $\leftarrow$ empty list
        for $j \leftarrow 1$ to num do
            if $j =$ length of $e$ then
                break
            if $j < \frac{num}{2}$ then
                pos append $i - 1 - i \mod 2 - j \cdot 2$
            else
                pos append $i - 2 + i \mod 2 - (j - \frac{num}{2}) \cdot 2$
        sort pos
        for $j \leftarrow 1$ to length of pos do
            line[pos[$k$]] = read[$i, 1, j$]
for $n \leftarrow 0, 2, 4, \ldots,$ length of read $- 2$ do
    Data append [line[$n$], line[$n+1$]]
return Data
```

2.4 Training

2.4.1 Optimizer

After having prepared our dataset, we can finally train our network. In order to do that we use Python and the Keras library with the Tensorflow backend. First, we have to determine the hyperparameters, such as the learning rate, decay, size and number of hidden layers etc. Furthermore we need an appropriate loss function and optimizer. An overview of the most commonly used optimizers is given below.

Stochastic Gradient Descent: In order to optimize the RNN stochastic gradient descent (SGD) [6] uses the gradient of the loss function f. In each training step, the parameters $\boldsymbol{x}$ are updated by

$$\boldsymbol{x}_{t+1} := \boldsymbol{x}_t - \eta \nabla f(\boldsymbol{x}_t), \tag{2.5}$$

where η denotes the learning rate. If η is small enough the loss gets smaller with each step.

Momentum: Momentum [20] adds another term to the equation which uses the previous update to prevent the learning process from oscillations.

$$\boldsymbol{x}_{t+1} := \boldsymbol{x}_t - \eta \nabla f(\boldsymbol{x}_t) + \alpha(\boldsymbol{x}_t - \boldsymbol{x}_{t-1}), \tag{2.6}$$

where α denotes the coefficient of momentum. This extra term enlarges the step size when the direction of the gradient stays the same and reduces it when the direction changes. It is called momentum due to its analogy to the momentum in physics.

Nesterov Accelerated Gradient Descent: Nesterov accelerated gradient descent (NAG) [17, 16] changes the second term of (2.6) to $\gamma \nabla f(\boldsymbol{x}_t + \alpha(\boldsymbol{x}_t - \boldsymbol{x}_{t-1}))$. By using the momentum for the gradient, Nesterov approximates the next position of the parameters. This method improved the convergence rate from $\mathcal{O}(1/t)$ to $\mathcal{O}(1/t^2)$.

Adagrad: Unlike SGD, Adagrad [11] uses different learning rates for every parameter $x_{t,i}$. These learning rates are updated at every time step τ by using the outer product matrix

$$G_t := \sum_{\tau=1}^{t} g_\tau g_\tau^T, \tag{2.7}$$

where $g_\tau = \nabla f(\boldsymbol{x}_\tau)$. The update of $\boldsymbol{x}_t$ is given by

$$x_{t+1,i} := x_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}. \tag{2.8}$$

The ϵ is added to avoid division by zero. Adagrad's weakness is that the learning rate decreases with every time step and eventually becomes infinitesimally small.

Adadelta: Adadelta [24] fixes Adagrad's problem by restricting the number of past gradients used to some fixed size w. Therefore, the factor G_{ii} is replaced by the running average $E[g^2]_t$, which is defined by

$$E[g^2]_{t+1} = \gamma E[g^2]_t + (1-\gamma) g_{t+1}^2. \tag{2.9}$$

Since $\sqrt{E[g^2]_t + \epsilon}$ is equal to the root mean squared (RMS) error criterion of the gradient, the update of $\boldsymbol{x}$ can be written as

$$x_{t+1,i} := x_{t,i} - \frac{\eta}{\text{RMS}[g]_t} g_{t,i}. \tag{2.10}$$

Adam: In addition to evaluating the RMS of the gradient $\boldsymbol{v}_t$ like Adadelta, Adaptive Moment Estimation (Adam) [13] also calculates the exponentially decaying average of the gradient $\boldsymbol{m}_t$. $\boldsymbol{v}_t$ and $\boldsymbol{m}_t$ are recursively defined by

$$\boldsymbol{m}_{t+1} := \beta_1 \boldsymbol{m}_t + (1-\beta_1) g_{t+1}, \tag{2.11}$$
$$\boldsymbol{v}_{t+1} := \beta_2 \boldsymbol{v}_t + (1-\beta_2) g_{t+1}^2. \tag{2.12}$$

Since $\boldsymbol{v}_t$ and $\boldsymbol{m}_t$ are initialized as vectors of zeros their values are too small at the beginning and therefore have to be corrected by

$$\hat{\boldsymbol{m}}_t := \frac{\boldsymbol{m}_t}{1-\beta_1^t}, \tag{2.13}$$
$$\hat{\boldsymbol{v}}_t := \frac{\boldsymbol{v}_t}{1-\beta_2^t}. \tag{2.14}$$

$\hat{\boldsymbol{m}}_t$ and $\hat{\boldsymbol{v}}_t$ are then used for updating $\boldsymbol{x}$ by

$$x_{t+1,i} := x_{t,i} - \frac{\eta}{\sqrt{\hat{v}_i} + \epsilon} \hat{m}_{t+1,i}. \tag{2.15}$$

Adamax: Adamax replaces $\hat{\boldsymbol{v}}_t$ from the Adam optimizer by a vector $\boldsymbol{u}_t$, which is defined as

$$\boldsymbol{u}_{t+1} := \max(\beta_2 \cdot \boldsymbol{u}_t, |g_{t+1}|). \tag{2.16}$$

Since $\boldsymbol{u}_t$ is not biased to zeros, a bias correction is not necessary.

Nadam: Nesterov-accelerated Adaptive Moment Estimation (Nadam) [10] extends the Adam optimizer by using NAG. For this purpose g_t is replaced by $\gamma \nabla f(\boldsymbol{x} + \alpha \Delta \boldsymbol{x})$. The update of $\boldsymbol{x}$ is then given by

$$x_{t+1,i} := x_{t,i} - \frac{\eta}{\sqrt{\hat{v}_{t+1,i}} + \epsilon} (\beta_1 \hat{m}_{t+1,i} + \frac{(1-\beta_1) g_{t+1,i}}{1-\beta_1^{t+1}}). \tag{2.17}$$

2.4.2 Loss Function

During training, binary cross entropy turned out to be by far the best loss function. The only difference to the categorical cross entropy is the fact that it can only categorize two classes. The binary cross entropy function f is defined as

$$f(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\frac{1}{N} \sum_{n=1}^{N} y_n \log(\hat{y}_n) + (1 - y_n) \log(1 - \hat{y}_n), \tag{2.18}$$

where $\boldsymbol{y}$ denotes the predicted and $\hat{\boldsymbol{y}}$ the actual output.

2.4.3 Hyperparameters

The layer size and the number of hidden layers have a major impact on the accuracy. If the size is too small the network cannot cover the complexity of the problem. On the other hand, too large a size leads to overfitting and longer computation time. Thus we have to try out several sizes to optimize the accuracy and computation time. Changing the learning rate, β_1 and β_2, as well as using learning rate decay have no positive influence on the precision. Due to the huge amount of training data overfitting is no problem. This is the reason why regularizers are not only unnecessary, but also slow down the learning process.

2.4.4 Building the Model

In the following code lines, written in Python, one can see the structure of the model. First, we transfer the layer size, the number of layers, the regularizer and the input size to the model. The input size is equal to three, since we only use the mean, standard deviation and length of the events. In line 15 the shape is set to `(None, input_size)`, where `None` means that one dimension is variable. Next, we add the layers to the model. Each layer, except the last one is a bidirectional LSTM-layer. If `layer_num` is smaller than one, the program will return an error. The first layer depends on the input, every other layer depends on the previous one. Finally, there are two output layers, which are time distributed, since the input length is variable. The output size equals five, whereby the first four ones represent the bases and the last one a blank. Softmax is used as our activation function.

```
def build_model(layer_size,layer_num,reg,input_size):
  inputs = Input(shape=(None, input_size))
  layer = []
  layer.append(Bidirectional(LSTM(layer_size,
      return_sequences=True,
      kernel_regularizer=regularizers.l2(reg)),
      input_shape=(None,input_size))(inputs))
  for n in range(layer_num-1):
    layer.append(Bidirectional(LSTM(layer_size,
      return_sequences=True,
      kernel_regularizer=regularizers.l2(reg)))(layer[-1]))
  out_layer1 = TimeDistributed(Dense(5, activation="softmax",
      kernel_regularizer=regularizers.l2(reg)),
      name="out_layer1")(layer[-1])
  out_layer2 = TimeDistributed(Dense(5, activation="softmax",
      kernel_regularizer=regularizers.l2(reg)),
      name="out_layer2")(Concatenate()([layer[-1], out_layer1]))
  return Model(inputs=inputs, outputs=[out_layer1, out_layer2])
```

2.4.5 Training the Model

To simplify the code, we use a generator for the training and validation. The generator is constructed by calling

```
training_generator = DataGenerator(args.subseq_size,
  trainPackage,32)
```

and defined as

```
class DataGenerator():
  def __init__(self,subseq_size,data_package,batch_size):
    self.subseq_size = subseq_size
    self.data_package = data_package
    self.batch_size = batch_size.
```

`subseq_size` defines how many events are used per training step. This number is essential for the speed, as well as the `batch_size`. If it is too low, the gradient may point in the wrong direction. If it is too high, each step will take longer. When the generator is called, three empty lists will be created. `Data_X` contains the input, whereas `Data_Y1` as well as `Data_Y2` contain the corresponding outputs one and two. The number of sequences added is equal to the batch size. In the for loop we choose a random sub-sequence from a random read. Next, we add the events and bases from the sub-sequence to the corresponding lists. After transferring the lists to numpy-arrays we return them.

```
def base_transformation(self,out):
  new_out = [[0,0,0,0,0] for i in range(len(out))]
  for m,n in enumerate(out):
    new_out[m][n] = 1
  return new_out

def __next__(self):
  while True:
    Data_X = []
    Data_Y1 = []
    Data_Y2 = []
    for n in range(self.batch_size):
      file_nr = np.random.randint(0,
        len(self.data_package.files)/2)
      if len(self.data_package['arr_%d' % (file_nr·2)]) <
        self.subseq_size:
        continue
      event_nr = np.random.randint(0,
        len(self.data_package['arr_%d' % (file_nr*2)])-
        self.subseq_size)
      Data_X.append(self.data_package['arr_%d' % (file_nr*2)]
        [event_nr:event_nr+self.subseq_size])
      Data_Y1.append(self.base_transformation(
        self.data_package['arr_%d' % (file_nr*2+1)]
        [event_nr:event_nr+self.subseq_size,0]))
      Data_Y2.append(self.base_transformation(
        self.data_package['arr_%d' % (file_nr*2+1)]
        [event_nr:event_nr+self.subseq_size,1]))
    Data_X = np.array(Data_X)
    Data_Y1 = np.array(Data_Y1)
    Data_Y2 = np.array(Data_Y2)
    return (Data_X,[Data_Y1,Data_Y2])
```

The generator is called when we train our model.

```
  history.append(model.fit_generator(generator=
      training_generator,initial_epoch=n,epochs=n+1,
      steps_per_epoch=args.steps_per_epoch,
  class_weight=class_weight,validation_data=validation_
      generator,validation_steps=2,verbose=1))
```

To save our model, we stop the training by setting `epochs` to `n+1`.

2.5 Sequencing

For the last step, the sequencing, we load the model we trained before. For using our own detection we have to use a different model than the detection of Metrichor as its break point detection is different from ours. In this case we have to apply the break point detection to the raw data first. At this point it is important to use the same window lengths and thresholds we used during the training. As a next step, we use the prediction function from Keras to apply our model to the events. Then we have to translate the output to the sequence. Thereby we evaluate the maximum of output 1 and 2 of every event, convert it to the corresponding bases and remove the blanks. Finally, we write the filenames and sequences to a FASTA file.

Chapter 3

Numerical Results

In order to develop the basecaller with the highest possible accuracy we need to evaluate the optimal hyperparameters. In this chapter we therefore compare the obtained results for each of them. Moreover, we will go into detail on the outputs of the break point detection algorithms.

3.1 Recurrent Neural Network

To train the RNN, the E. coli data set [4] was used. This data set contains over 50,000 reads, hence it provides enough data to work with. Moreover the E. coli bacteria has almost no mutations and is therefore perfectly suited for training. In order to reach accurate results several options had to be tested, such as different loss functions, optimizers, layer size, etc.

In the following sections those options are compared by using the loss and the accuracy of the validation data for the outputs 1 and 2.

3.1.1 Loss Function

Choosing an appropriate loss function had the biggest impact on the results. Figure 3.1 and Figure 3.2 show the accuracies of the available loss functions of Keras. `Binary cross entropy` had by far the highest accuracy, whereas `hinge`, `mean absolute error` and `mean absolute percentage error` delivered the lowest precision.

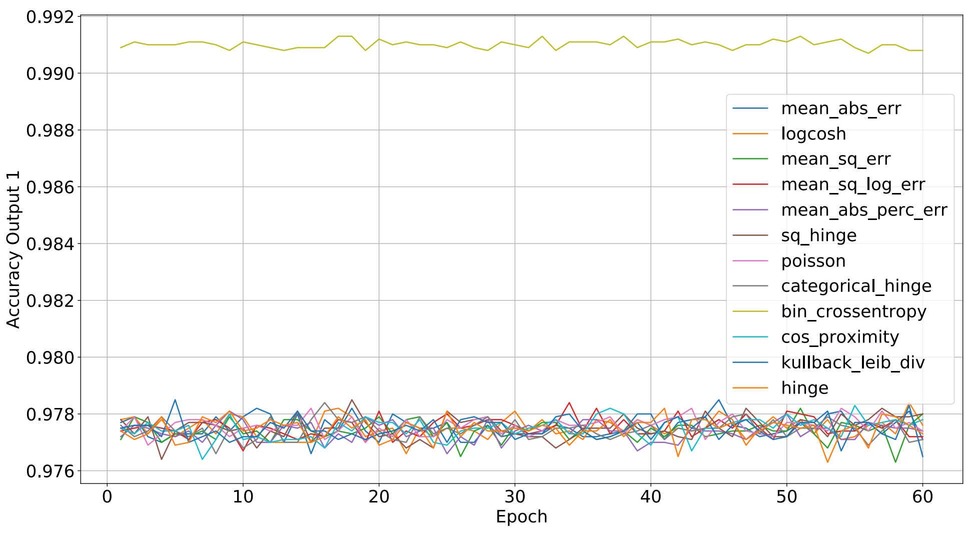

Figure 3.1: Accuracy of output 1 for different loss functions.

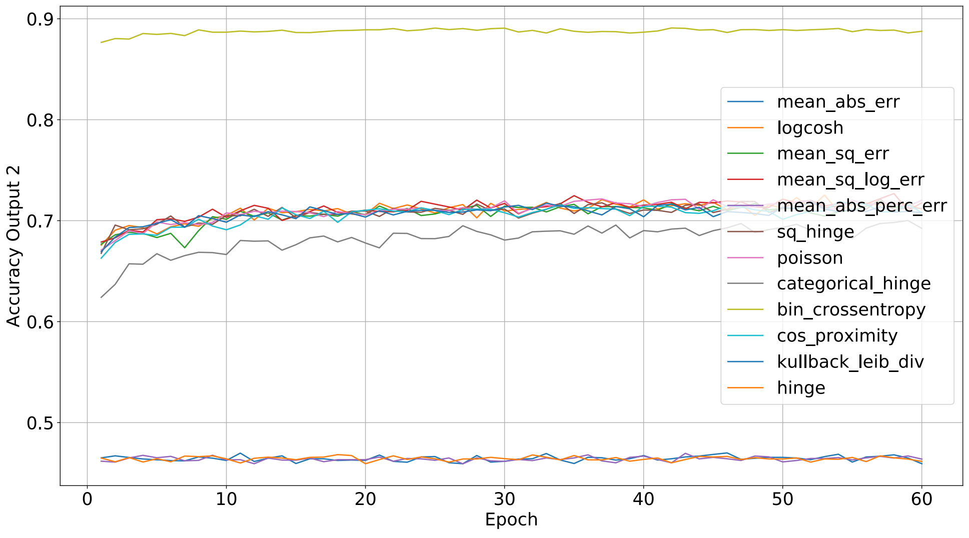

Figure 3.2: Accuracy of output 2 for different loss functions.

3.1.2 Optimizer

The choice of the optimizers was crucial for precise training results. Figure 3.3 compares the loss function and Figure 3.4 and Figure 3.5 compare the accuracies of output 1 and 2 for several optimizers. Evidently the *Adam*-

optimizer had the highest precision and leads just ahead of *Nadam*, followed by *RMSprop*, *Adamax* and *Adagrad*. *SGD* and *Adadelta* delivered the most imprecise results.

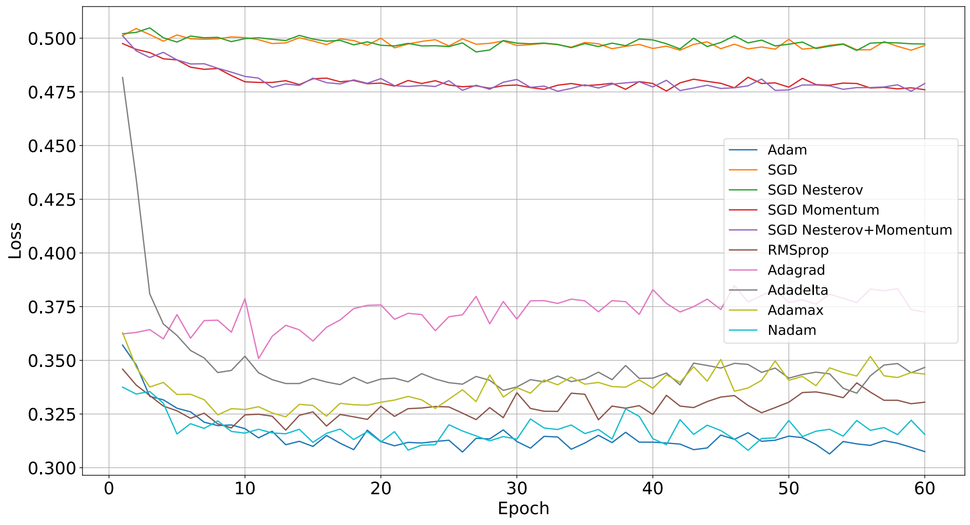

Figure 3.3: Loss for different optimizers.

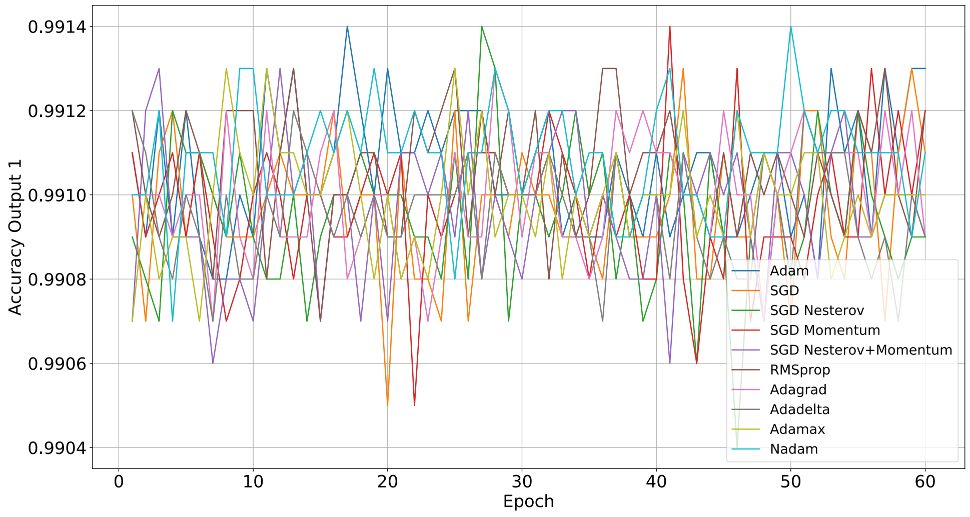

Figure 3.4: Accuracy of output 1 for different optimizers.

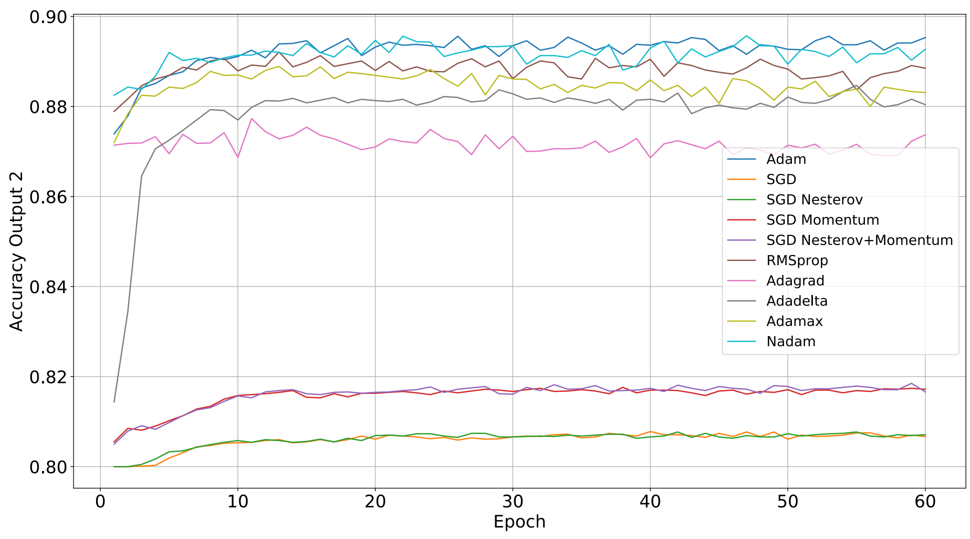

Figure 3.5: Accuracy of output 2 for different optimizers.

3.1.3 Size of the Layers

Another crucial factor was the choice of the size of the hidden layers. Figures 3.6–3.8 indicate that, up to a certain value, a higher layer size led to a higher accuracy. This held up to a size of 80 layers. More than 80 layers did not lead to any improvements, but only to higher computation times.

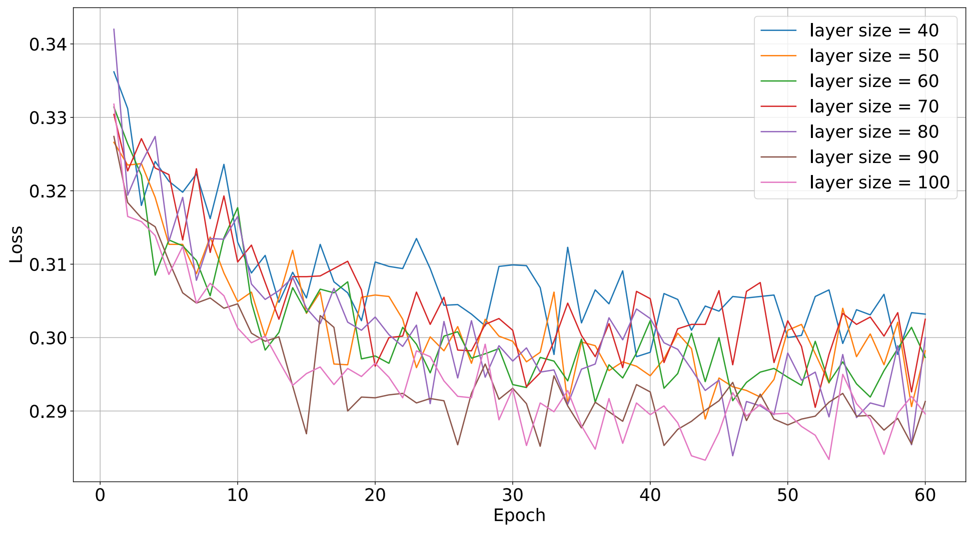

Figure 3.6: Loss of different layer sizes.

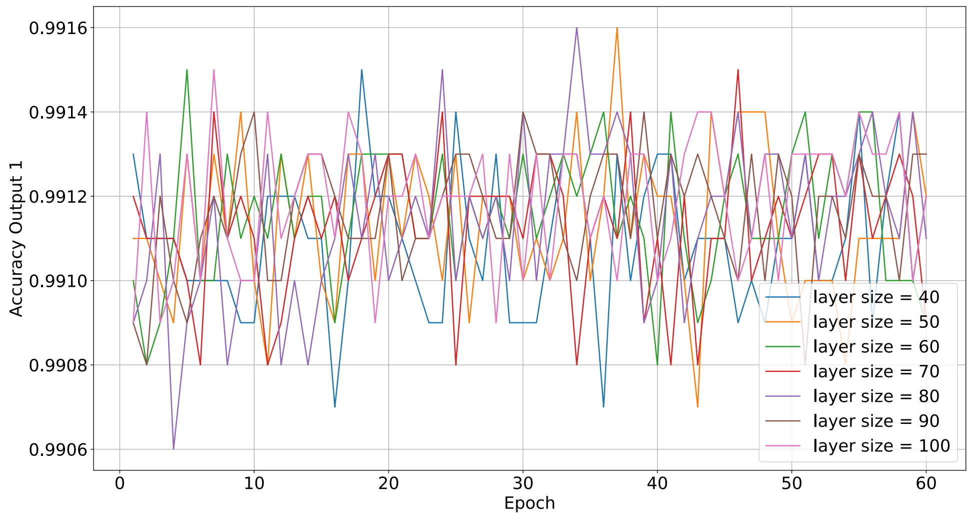

Figure 3.7: Accuracy of output 1 for different layer sizes.

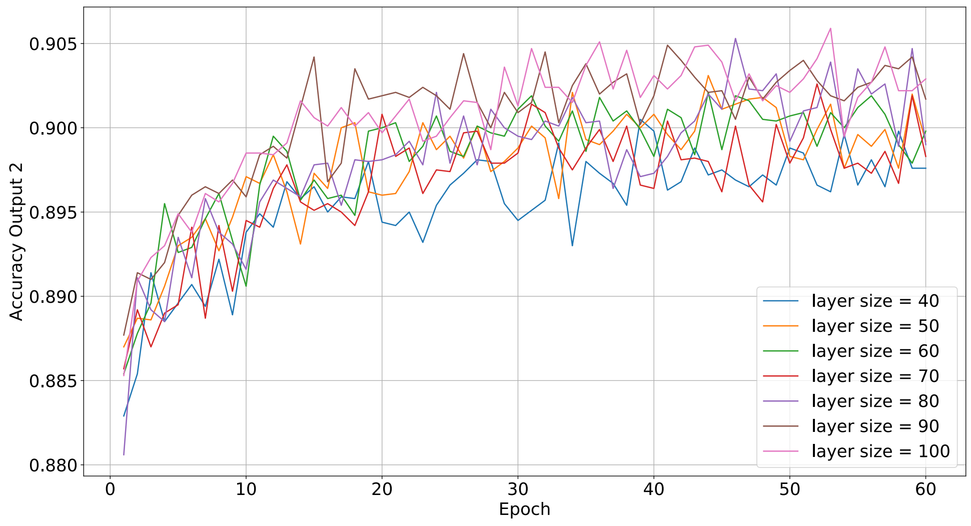

Figure 3.8: Accuracy of output 2 for different layer sizes.

3.1.4 Number of Layers

It was also important to choose the optimal number of hidden layers. If this number was too small the complexity of the problem cannot be covered, if it was too high the computation time and the risk of overfitting would increase unnecessarily. As we can see from the Figures 3.9–3.11 the error rate of RNNs with two hidden layer was higher than the error rate of networks with three or four layers. However a fourth layer had no benefits.

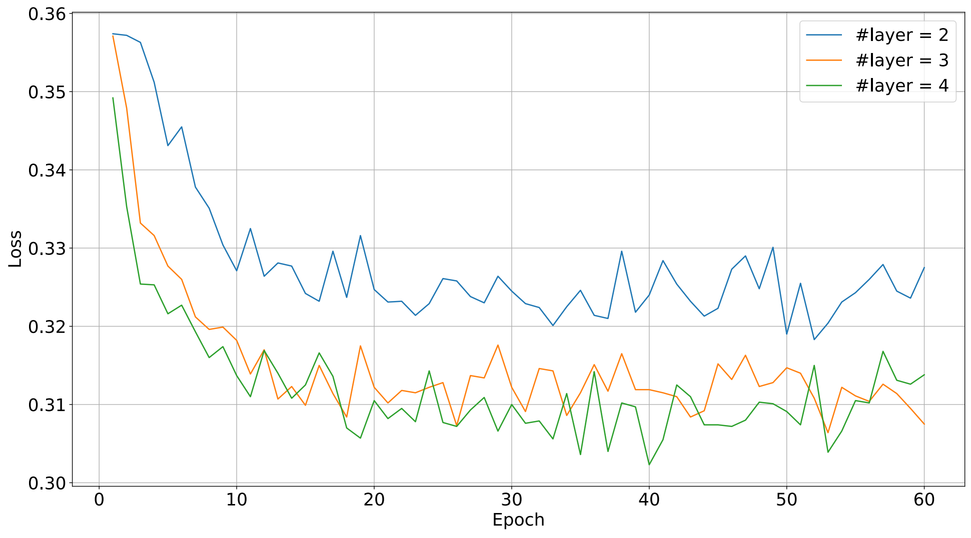

Figure 3.9: Loss for different numbers of layers.

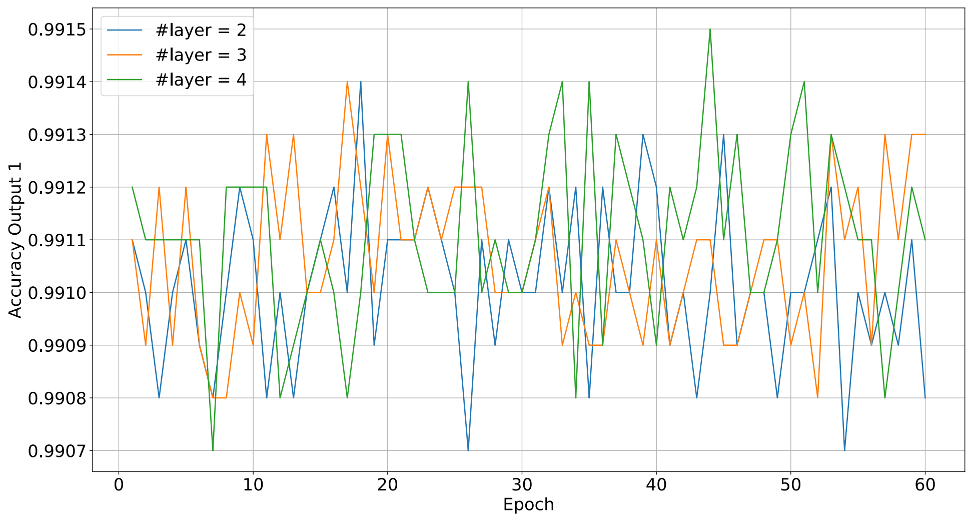

Figure 3.10: Accuracy of output 1 for different numbers of layers.

3.1.5 Learning Rate Decay

At the beginning of training an RNN a higher learning rate accelerates the learning progress. On the other hand, if the network is near the optimum it is advisable to use smaller learning rates in order to make only small adjustments. This process was implemented by using learning rate decay. Therefore, an initial learning rate and a decay rate were set. Later the learning rate was divided by the decay rate at every 10 epochs. The effects of this procedure are illustrated in the Figures 3.12–3.14. It is evident that a decay rate of 2 led to the highest accuracy and the fastest learning progress. A decay rate of 1.5 also accelerated the training compared to using no learning rate decay, but stagnated at a certain point.

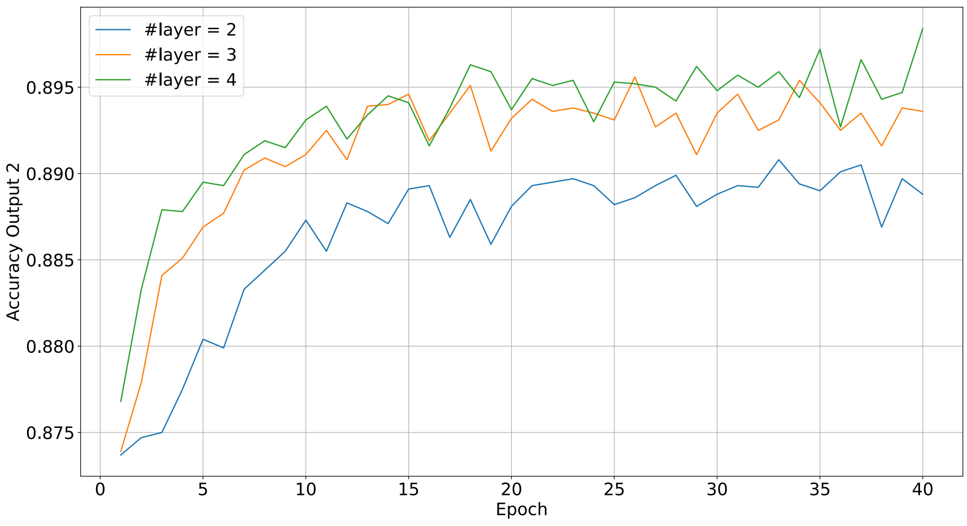

Figure 3.11: Accuracy of output 2 for different numbers of layers.

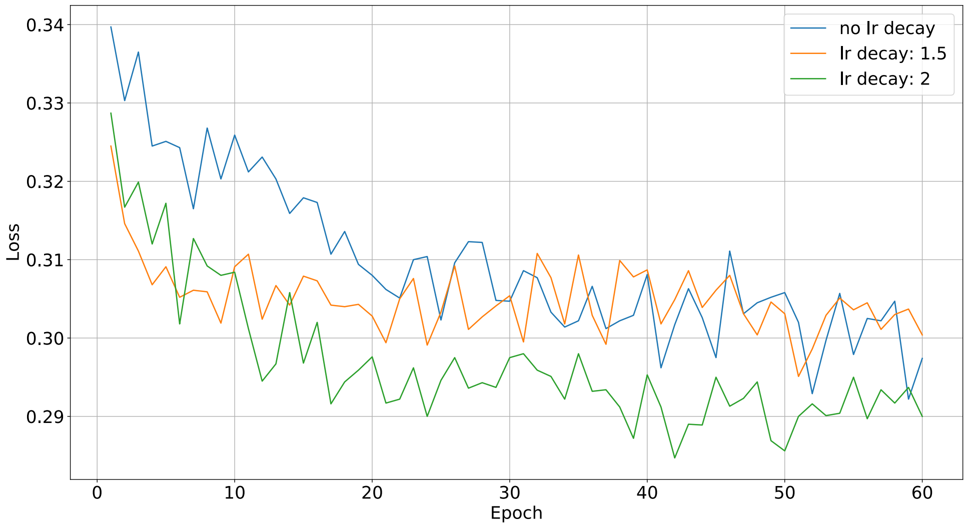

Figure 3.12: Loss for different decay rates.

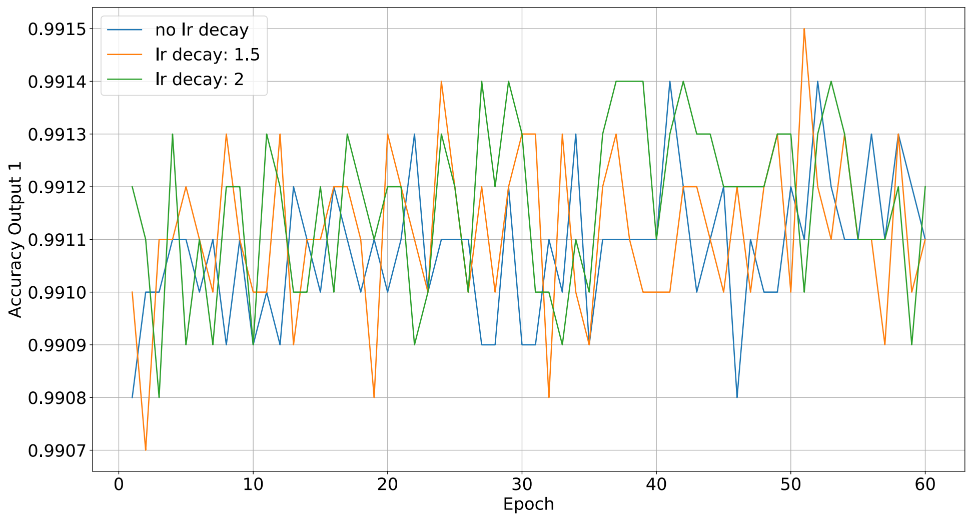

Figure 3.13: Accuracy of output 1 for different decay rates.

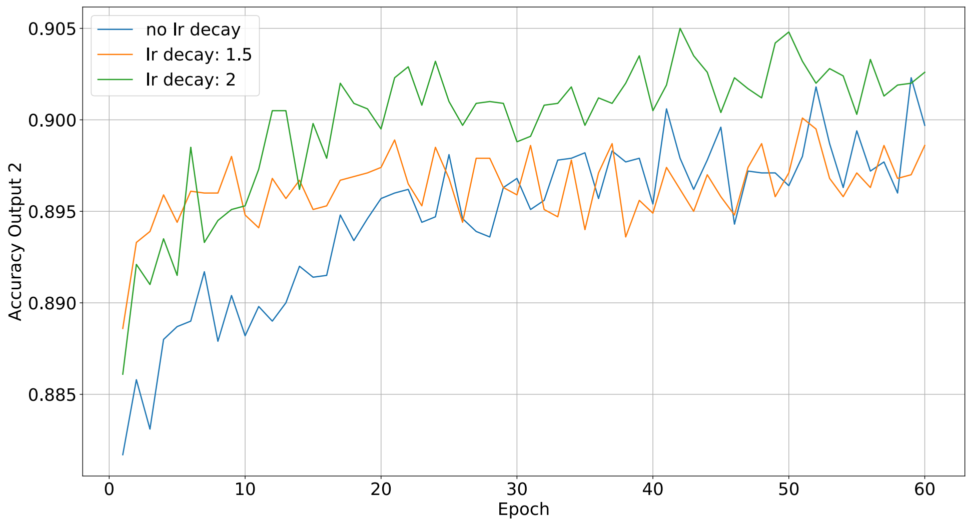

Figure 3.14: Accuracy of output 2 for different decay rates.

3.1.6 Class Weight

In order to focus on output classes that are underrepresented during training Keras provides a function called `class weight`. In our case, this concerned the bases of output 1, since over 95% of the output elements were blanks, see Table 3.2. Therefore, setting the class weight of N to 0.5 and the weights for the bases to 1 had the potential to increase the accuracy of output 1. As it is evident by Figure 3.16 using class weights had no positive impact, but it slightly decreased the precision of output 2, see Figure 3.17.

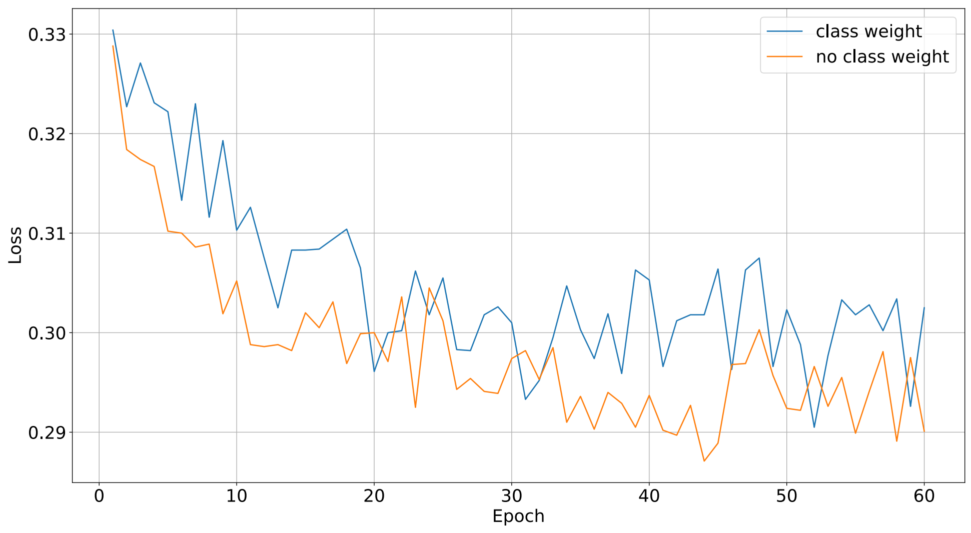

Figure 3.15: Influence of using class weights on the loss.

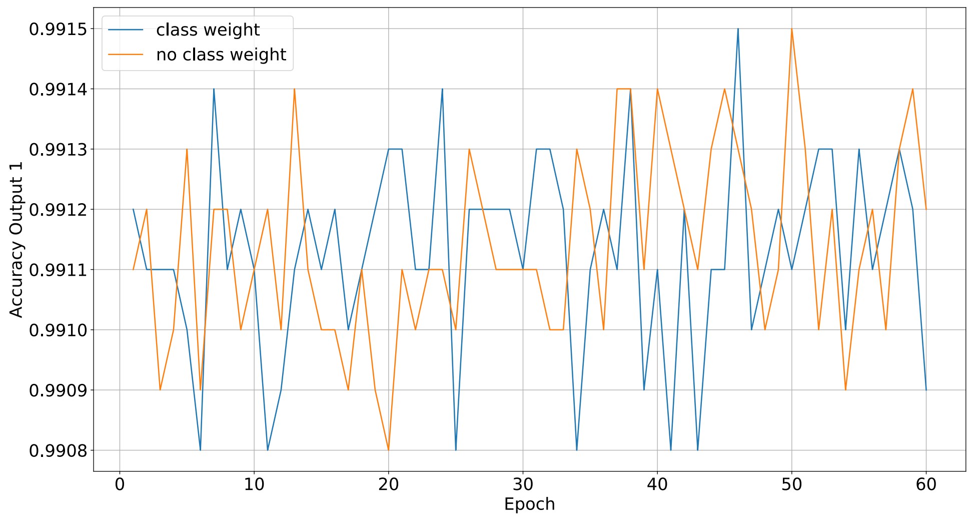

Figure 3.16: Influence of using class weights on the accuracy of output 1.

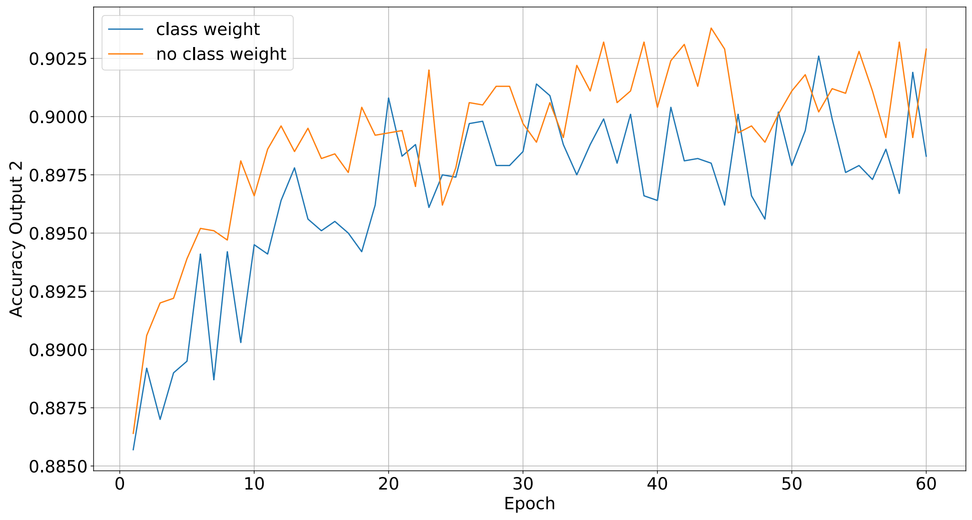

Figure 3.17: Influence of using class weights on the accuracy of output 2.

3.1.7 Regularizer

Regularizers [22, 23, 14] are able to prevent the network from overfitting. Due to the huge amount of training data overfitting did not take place. For this reason, using regularizers had no positive impact on the accuracy for small values, as Figures 3.18–3.20 show. The higher the value got, the slower the RNN learned.

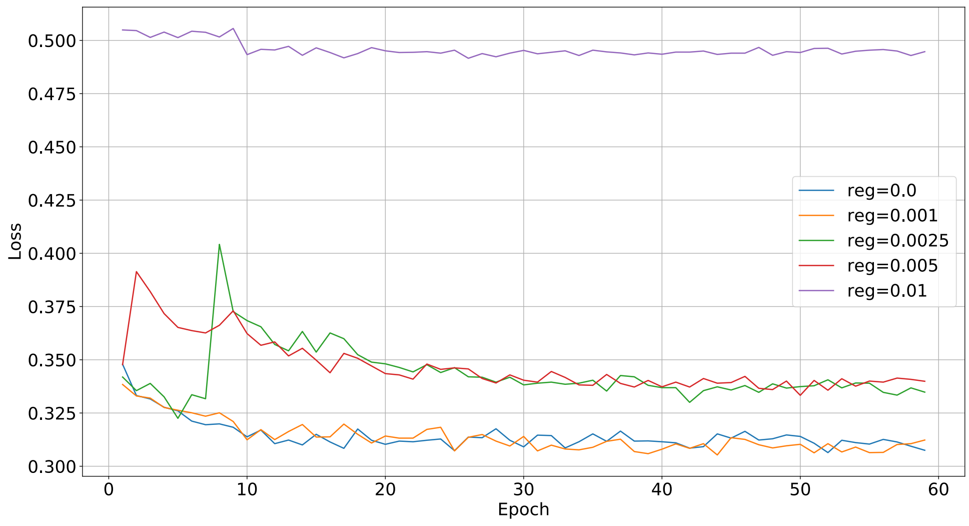

Figure 3.18: Influence of using a regularizer on the loss.

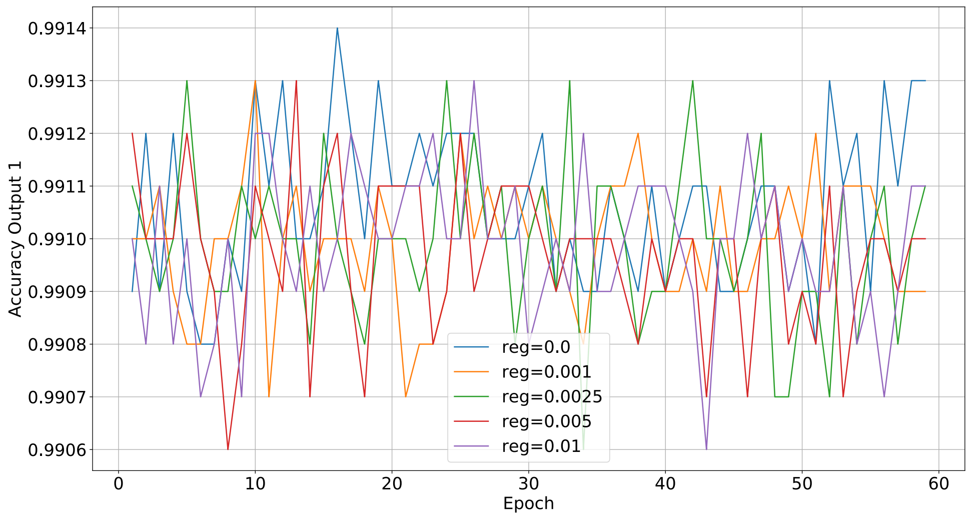

Figure 3.19: Influence of using a regularizer on the accuracy of output 1.

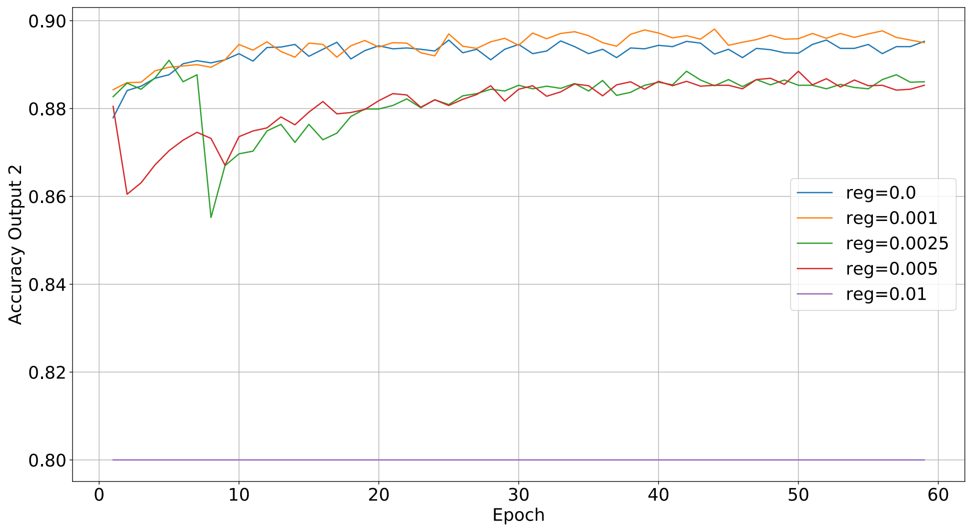

Figure 3.20: Influence of using a regularizer on the accuracy of output 2.

3.1.8 Training vs. Validation

Finally, after testing each hyperparameter the optimal training conditions can be set up. In order to recap a summary of the final configurations is listed in Table 3.1. An easy way to determine if the network is overfitted is to compare the loss and accuracies of the training with those of the validation. This comparison is visualized in the Figures 3.21–3.23. Against all expectations the precision of the validation was higher than the precision of the training. One explanation for this circumstance might be that the accuracies and the loss of the validation were evaluated at the end of each epoch and the values of the training were determined during the whole epoch. Nevertheless, this led to the conclusion that no overfitting has taken place.

Loss function	Binary cross entropy
Optimizer	Adam
Layer size	80
Number of layers	3
Learning rate decay	Factor 2 for every 10 epochs
Class weights	No class weights used
Regularizers	No regularizers used

Table 3.1: Summary of the final configurations for the training.

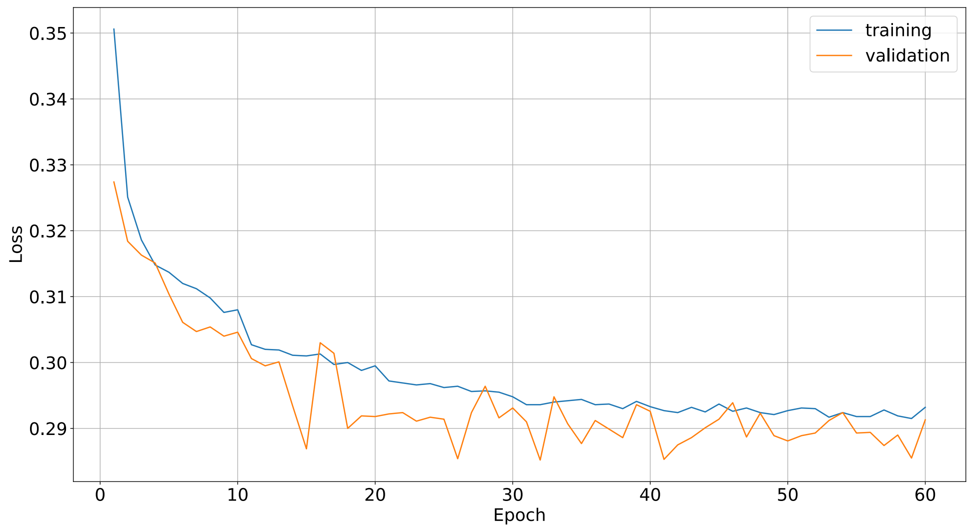

Figure 3.21: Comparison of the training with the validation based on the loss.

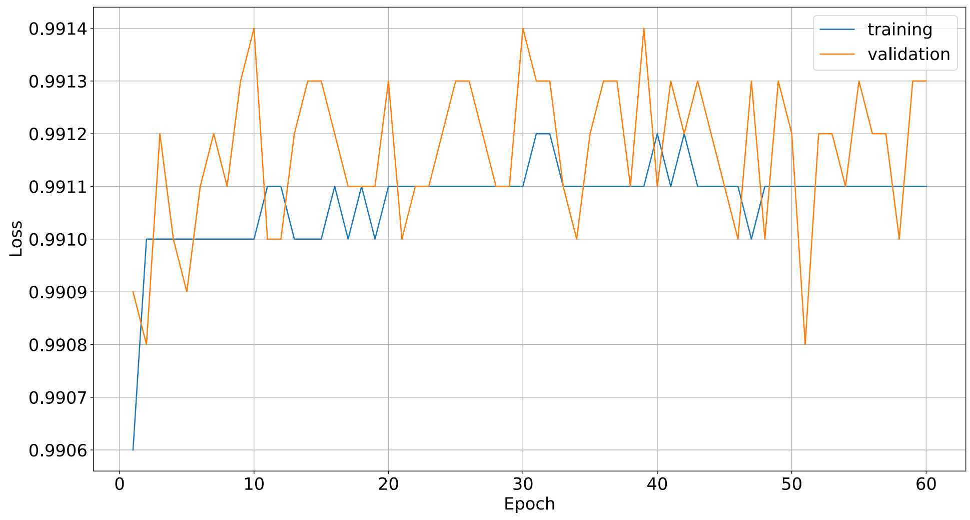

Figure 3.22: Comparison of the training with the validation based on the accuracy of output 1.

3.2 Break Point Detection

As mentioned in Section 2.1.5 the goal is to set the break points so that exactly one base per event goes through the pore as often as possible. Therefore we evaluate the accuracies and how many bases went through the pore while using the window-based and Metrichor's break point detection. In this section we compare both.

3.2.1 Window Lengths and Thresholds

Applying Algorithm 4 of Section 2.1.5 to the raw reads yielded certain values for the window lengths and thresholds: short window length = 6, long window length = 10, threshold for the short window = 1.4 and threshold for the long window = 2.2. Table 3.2 shows the amount of events where zero, one and two or more bases went through the pore and how many bases were skipped due to too long events for the window-based break point detection and the break point detection of Metrichor. The ideal case was maximizing the amount of events with exactly one base, while simultaneously minimizing the deleted bases. On the one hand the window-based break point detection had more events with one base, on the other hand it had more deleted bases.

3.3 Comparison of the results

In order to compare the results of the different basecallers they were applied to test files. Afterwards, Minialign was applied on the outputs to determine the accuracies. Table 3.3 and Figure 3.24 compare the accuracies using the basecaller of Metrichor and the RNN with and without break point detection. Apparently, Metrichor had the highest accuracy, but also the

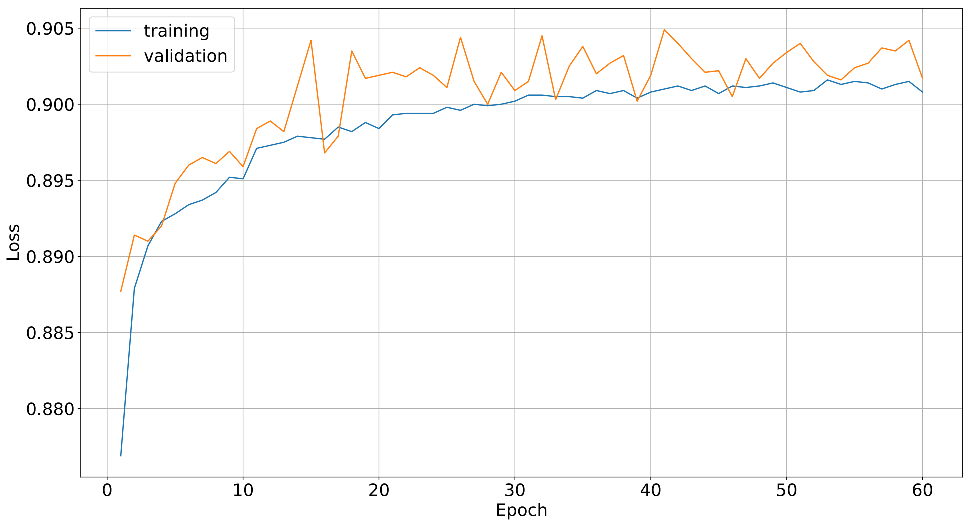

Figure 3.23: Comparison of the training with the validation based on the accuracy of output 2.

	Window-based break point detection	Metrichor break point detection
0 bases/event	41.1%	46.5%
1 base/event	53.5%	51.3%
≥2 bases/event	5.43%	2.22%
Deleted bases	3.07%	1.71%

Table 3.2: Amount of events, with 0, 1 and 2 or more bases passing the pore and the amount of deleted bases for the window-based and Metrichor's break point detection.

highest standard deviation. The average precision of the RNN was slightly lower, especially when using the window-based break point detection. This indicates that using another break point detection algorithm may have the potential for improvements.

	Average of the accuracy	Standard deviation of the accuracy
Metrichor	79.0%	7.42%
RNN with Metichor bp detection	76.4%	6.18%
RNN with window-based bp detecion	73.7%	4.51%

Table 3.3: Accuracies of the different basecallers.

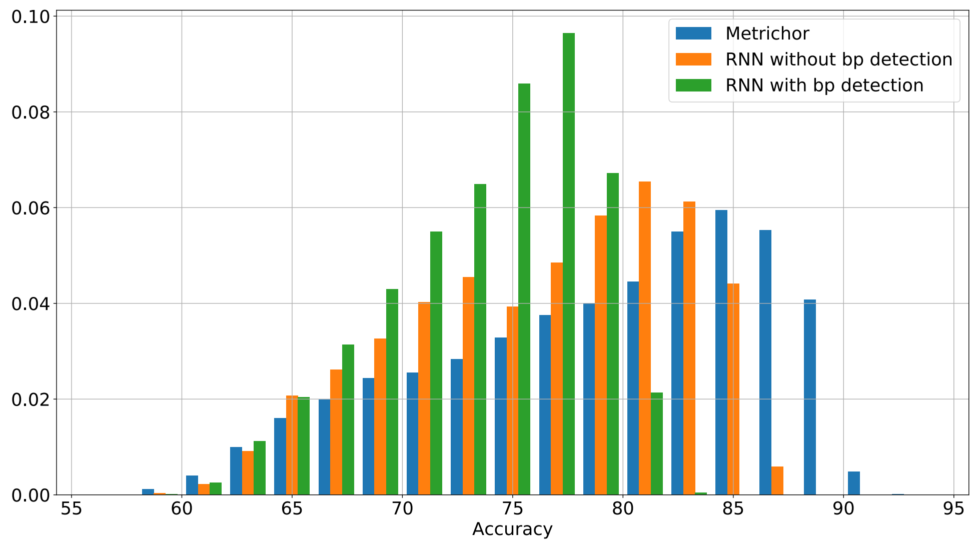

Figure 3.24: Histogram of the accuracies using the basecallers of Metrichor and the RNN with the Metrichor's and the window-based break point detection.

Chapter 4

Conclusions

The purpose of this master thesis was to develop and to evaluate a basecaller for nanopore sequencing with a high accuracy. The basecaller from Metrichor was used, which has an accuracy of 79%, served as a comparison. Inspired by other projects that used machine learning for basecalling, we decided to use RNNs. In order to maximize the precision many adjustments of the hyperparameters had to be made. The only method to figure out which options worked best was to test and compare each of them. During our research it became clear that using an appropriate loss function was the most influential factor. *Binary cross entropy* turned out to be the only useful loss function, as the outputs of other loss functions had such low accuracies that not even a single read could be aligned to the reference. Changing the other hyperparameters influenced the precision by just a few percent as Section 3.1 shows.

Although we optimized several hyperparameters, the accuracy of the RNN, when using the break point detection of Metrichor, was 76.4% and thus lower than the precision of Metrichor's basecaller. This suggests that RNNs work, but they are less suitable for basecalling than hidden Markov models.

It was necessary to implement a break point detection since some versions of Metrichor did not use events for basecalling. There were many algorithms to choose from. Because many of them were time consuming, the choice fell on the window-based break point detection algorithm. This algorithm achieved a higher amount of events with exactly one base passing through. However, there were also more deleted bases. Unfortunately, Metrichor's break point detection is unknown, thus no conclusions can be drawn as to why it works better. Nevertheless, the comparison of the results of the RNNs with Metrichor's break point detection on the one hand and with the window-based break point detection on the other hand, allow conclusions in regards to possible improvements. Apparently the number of deleted bases had a greater impact on the accuracy than the amount of events with exactly one base.

Bibliography

[1] `https://github.com/jeammimi/deepnano`.

[2] `https://metrichor.com/`.

[3] `https://github.com/ocxtal/minialign`.

[4] `http://s3.climb.ac.uk/nanopore/R9_Ecoli_K12_MG1655_lambda_MinKNOW_0.51.1.62.tar`.

[5] E. Ahmed, A. Clark, and G. Mohay. A novel sliding window based change detection algorithm for asymmetric traffic. In *IFIP International Conference on Network and Parallel Computing*, pages 168–175. IEEE, 2008.

[6] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT 2010*, pages 177–186. Springer, 2010.

[7] D. Branton and D. Deamer. *Nanopore Sequencing: An Introduction.* World Scientific, 2019.

[8] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[9] D. Deamer, M. Akeson, and D. Branton. Three decades of nanopore sequencing. *Nature Biotechnology*, 34(5):518, 2016.

[10] T. Dozat. Incorporating Nesterov momentum into ADAM. Caribe Hilton, San Juan, Puerto Rico, 2016.

[11] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[12] Y. Feng, Y. Zhang, C. Ying, D. Wang, and C. Du. Nanopore-based fourth-generation DNA sequencing technology. *Genomics, Proteomics & Bioinformatics*, 13(1):4–16, 2015.

[13] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2014.

[14] C.-S. Leung, A.-C. Tsoi, and L. W. Chan. Two regularizers for recursive least squared algorithms in feedforward multilayered neural networks. *IEEE Transactions on Neural Networks*, 12(6):1314–1332, 2001.

[15] S. Liu, M. Yamada, N. Collier, and M. Sugiyama. Change-point detection in time-series data by relative density-ratio estimation. *Neural Networks*, 43:72–83, 2013.

[16] Y. Nesterov. A method for solving the convex programming problem with convergence rate $O(1/k^2)$. In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

[17] Y. Nesterov. *Lectures on Convex Optimization*, volume 137. Springer, 2018.

[18] E. Pennisi. Search for pore-fection. *Science*, 336(6081):534–537, 2012.

[19] M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[20] R. Sutton. Two problems with back propagation and other steepest descent learning procedures for networks. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society, 1986*, pages 823–832, 1986.

[21] C. Truong, L. Oudre, and N. Vayatis. Ruptures: change point detection in Python. *arXiv preprint arXiv:1801.00826*, 2018.

[22] L. Wan, M. Zeiler, S. Zhang, Y. L. Cun, and R. Fergus. Regularization of neural networks using dropconnect. In S. Dasgupta and D. McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1058–1066, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[23] L. Wu and J. Moody. A smoothing regularizer for feedforward and recurrent neural networks. *Neural Computation*, 8(3):461–489, 1996.

[24] M. D. Zeiler. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.