UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS num. 1572

End-to-End Deep Learning Model for Base Calling of MinION Nanopore Reads

Neven Miculinić

Zagreb, June 2018.

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING
MASTER THESIS COMMITTEE

MASTER THESIS ASSIGNMENT No. 1572

Student: **Neven Miculinić (0036477241)**
Study: Computing
Profile: Computer Science

Title: **End-to-End Deep Learning Model for Base Calling of MinION Nanopore Reads**

Description:

In the MinION device, single-stranded DNA fragments move through nanopores, which causes drops in the electric current. The electric current is measured at each pore several thousand times per second. Each event is described by the mean and variance of the current and by event duration. This sequence of events is then translated into a DNA sequence by a base caller. Develop a base-caller for MinION nanopore sequencing platform using a deep learning architecture such as convolutional neural networks and recurrent neural networks. Instead of events, use current waveform at the input. Compare the accuracy with the state-of-the-art basecallers. For testing purposes use publicly, available datasets and Graphmap or Minimap 2 tools for aligning called reads on reference genomes. Implement method using TensorFlow or similar library. The code should be documented and hosted on a publicly available Github repository.

Issue date: 16 March 2018
Submission date: 29 June 2018

Mentor: Committee Chair:

_____   _____
Associate Professor Mile Šikić, PhD   Full Professor Siniša Srbljić, PhD

Committee Secretary:

_____
Assistant Professor Tomislav Hrkać, PhD

Zagreb, 2. ožujka 2018.

DIPLOMSKI ZADATAK br. 1572

Pristupnik: **Neven Miculinić (0036477241)**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **Model dubokog učenja s kraja na kraj za određivanje očitanih baza dobivenih uređajem za sekvenciranje MinION**

Opis zadatka:

Unutar uređaja MinION, fragmenti jednostruke DNA prolaze kroz nanopore, što uzrokuje promjene u električnoj struji. Struja proizvedena na svakoj nanopori mjeri se nekoliko tisuća puta u sekundi. Svaki događaj opisan je srednjom vrijednosti i varijancom struje te svojim trajanjem. Postupak kojim se takav slijed događaja prevodi u niz nukleotida naziva se određivanje očitanih baza. Razviti alat za prozivanje baza za uređaj za sekvenciranje MinION koristeći modele dubokog učenje kao što su konvolucijske i povratne neuronske mreže. Umjesto događaja na ulazu koristi valni oblik struje. Usporediti dobivenu točnost s postojećim rješenjima. U svrhu testiranja koristiti javno dostupne skupove podataka i alate GraphMap ili Minimap 2 za poravnanje očitanja na referentni genom. Alat implementirati koristeći programsku biblioteku TensorFlow (ili neku sličnu). Programski kod treba biti dokumentiran i javno dostupan preko repozitorija GitHub.

Zadatak uručen pristupniku: 16. ožujka 2018.
Rok za predaju rada: 29. lipnja 2018.

Mentor:

Izv. prof. dr. sc. Mile Šikić

Djelovođa:

Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za
diplomski rad profila:

Prof. dr. sc. Siniša Srbljić

CONTENTS

LIST OF FIGURES

1. Introduction

In recent years, deep learning methods significantly improved the state-of-the-art in multiple domains such as computer vision, speech recognition, and natural language processing (LeCun and Bengio, 1998; Krizhevsky et al., 2012) In this paper, we present application of deep learning for DNA basecalling problem.

Oxford Nanopore Technology's MinION nanopore sequencing platform Mikheyev and Tin (2014) is the first portable DNA sequencing device. It produces longer reads than competing technologies. In addition, it enables real-time data analysis which makes it suitable for various applications. Although MinION is able to produce long reads, even up to 882 kb Loman (a,b), they have an error rate of 10% or higher. This master thesis uses R9.4 pore model and compares previous techniques with novel auto-encoder multi-task training.

1.1.   Organization

Chapter 2 covers the sequencing technologies basics, Oxford nanopore technologies(ONT) sequencing model, and available solutions to MinION sequencing. Chapter 3 covers common concepts, most from deep learning, used throughout this thesis. Chapter 4 covers the whole system architecture and important programming system parts. Chapter 5 goes into result analysis from the experiments. Chapter 6 reflects on the overall problem and voices author's opinion on related topics. Finally, in chapter 7 this thesis final words are uttered in the closing statement.

2. Background

Due to technical constraints, it's infeasible to sequence whole DNA in a single strand. Every sequencing technology to date has an upper limit how big strand can it precisely sequence. This limit is considerably smaller than the size of the genome. For example, E.Coli has 4.5 million base pairs in its DNA, while Sanger's sequencing maximum output is around 1000 base pairs max. To make DNA basecalling feasible technique called shotgun sequencing was invented. The strand is cloned number of times, then via chemical agent broken down into smaller fragments of appropriate length. Sequenced fragments are called reads.

Genome assembly is the process of reconstructing the original genome from reads and usually starts with finding overlaps between reads. The quality of reconstruction heavily depends on the length and the quality (accuracy) of the reads produced by the sequencer.

If we have reference sequence we usually align the reads on the reference to aid us in the genome assembly. Otherwise, we have to use many de novo assembly techniques.

The right analogy would be building a puzzle. Since we cannot scan the whole puzzle because our camera is too small or imprecise, we are scanning pieces of the whole picture. Puzzle pieces would represent fragments in this analogy. If we have a map, even a rough one, it shall aid us in assembling those puzzle pieces into complete pictures. Otherwise, we're fiddling in the dark and using de novo assembly techniques.

Figure 2.1 depicts process of sequencing visually.

In 1977, Frederick Sanger (Mile Šikić, 2013)(Pettersson et al., 2009) started development of sequencing technologies. It allowed reading lengths up to 1000 bases with very high accuracy(99.9%) at the cost of 1$ per 1000 bases. Later, seconds generation sequencing, like IAN Torrent and Illumina devices, reduced the price while keeping the accuracy high. However, they had a cost of shorter read lengths, about a few hundred base pairs, which makes resolving repetitive regions practically impossible.

Third generation sequencing technologies have longer read lengths at the accuracy's expense. PacBio, for example, developed technology with a few thousand bases with error rates of ~10-15%.

MinION sequences, which this master thesis use, made sequencing less expensive and even portable.
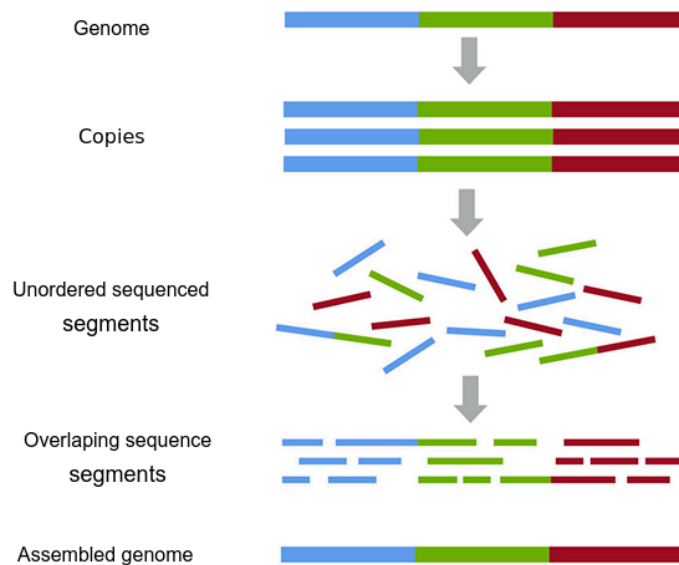
Figure 2.1: Depiction of the sequencing process

2.1. Oxford Nanopore MinION

The MinION device by Oxford Nanopore Technologies is the first portable DNA sequencing device. Its small weight, low cost, and long read length combined with decent accuracy yield promising results in various applications including full human genome assembly Jain et al. (2017) what could potentially lead to personalized genomic medicine. It weighs only 87 grams, and its portability leads to uses on the international space station and the antarctic among other places.

Under the hood, it has numerous nano-meter sized pores, thus a names Nanopore. Each pore has a width of around 6 nucleotides. Under electric current DNA strand passing through the pore and changes its electric resistance. The sensor measures the current through the pore multiple times a second [1]. This signal varies depending which k-mer is occupying the pore, and on its basis, we're performing the basecalling. On figure 2.2 this process is visually depicted.

MinION devices can produce long reads, usually tens of thousand base pairs (with reported reads lengths of 100 thousand Loman (a) and even recently above 800 thousand base pairs Loman (b)), but with high sequencing error than older generations of sequencing technologies.

The sequencing resulting file is in FAST5 format, which is adapted HDF5 file format, popular in the bioinformatics community. It stores raw signal, alongside various metadata. Unfortunately, many basecallers, including the official ones, upon executing store their re-

---

[1]The model we worked with had 4000 samples per seconds

[2]Figure adapted from https://nanoporetech.com/how-it-works
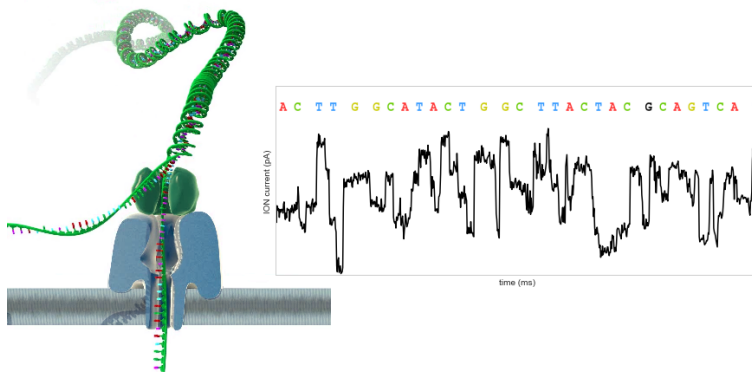
Figure 2.2: DNA strain being pulled through a nanopore [2]

sults in the FAST5 files. It leads to data and processing coupling in single file and bloated file sizes.
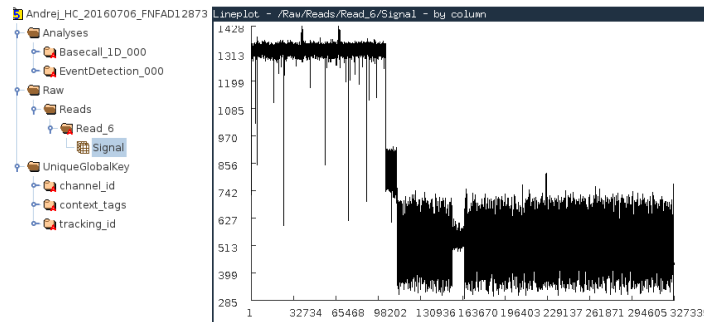
Figure 2.3: Structure of FAST5 file and raw signal line plot show in *HDFView* [3]

2.2. Related work

2.2.1. Oxford Nanopore Technologies

This subsection covers various basecallers published by ONT, the MinION device maker.

Metrichor

Metrichor is now defunct cloud-based basecaller. The older versions used *hidden Markov models* (HMM) as the underlying algorithm. Preprocessing started with segmenting signal into smaller chunks called events with their start, end, length, mean signal strength and standard deviation. These events are observations in the HMM model, and the underlying

---

[3]https://support.hdfgroup.org/products/java/hdfview/

generating hidden sequence is the sequence of 6-mers. They build their HMM transition matrix with stay, skip 1 and skip 2 probabilities, that is underlying 6-mer cannot move for more than 2 nucleotides per event. Basecalling is performed using the Viterbi algorithm. This approach showed poor results when calling long homopolymer stretches as the context in the pore remains the same Goodwin et al. (2016)Ip et al. (2015).

Nanonet

Nanonet is ONT first-generation neural network basecaller. It used to be available on GitHub, but it's now defunct and unavailable.

Albacore

Albacore is a production basecaller provided by Oxford Nanopore and uses a command-line interface. It utilizes the latest in Recurrent Neural Network algorithms in order to interpret the signal data from the nanopore and basecall the DNA or RNA passing through the pore. It implements stable features into Oxford Nanopore Technologies' software products, and is fully supported. It receives .fast5 files as an input, and is capable of producing:

- .fast5 files appended with basecalled information

- .fast5 files that have been processed, but basecall information present in a separate .fastq file

Guppy

Guppy is ONT's new basecaller that can use GPUs to basecall much faster than Albacore. Both the GridION X5 and PromethION contain GPUs and use Guppy to basecall while sequencing. Guppy can also use CPUs and scales well to many-CPU systems, so it may run faster than Albacore even without GPUs. In the future, it's intended to replace Albacore as production basecaller.

Scrappie

Scrappie[4] is ONT's research basecaller. Scrappie is reported to be the first basecaller that specifically address homopolymer base calling. It became publicly available just recently in June 2017 and supports R9.4 and future R9.5 data.

Unlike Albacore, Scrappie does not have fastq output, either directly or by writing it into the fast5 files – it only produces fasta reads.

---

[4]https://github.com/nanoporetech/scrappie

2.2.2. Third-party basecallers

Nanocall

Nanocall (David et al., 2016) was the first third-party open source basecaller for nanopore data. It uses HMM approach like the original R7 Metrichor. Nanocall does not support newer chemistries after R7.3.

DeepNano

DeepNano (Boža et al., 2017) was the first open-source basecaller based on neural networks. It uses bidirectional recurrent neural networks implemented in Python, using the Theano library. When released, originally only supported R7 chemistry, but support for R9 and R9.4 was added recently.

basecRAWller

basecRAWller [5] is developed by Marcus Stoiber and James Brown at the Lawrence Berkeley National Laboratory.

Chiron

Chiron (Teng et al., 2017) is developed by Haotian Teng and others in Lachlan Coin's group at the University of Queensland. They are basecalling from the raw signal, first using a residual convolutional neural network, than LSTM and finally beam search or greedy decoder depending on chosen configuration.

---

[5] `https://basecrawller.lbl.gov/`

3. Methods

This chapter is dedicated to explaining key deep learning concepts used throughout the master thesis. It's here primarily for completeness, and it's author recommendation to go into detail via other sources, for example, Deep learning book (Goodfellow et al., 2016), google, or research papers cited for most of the techniques. TensorFlow (Abadi et al., 2015) and Keras (Chollet et al., 2015) deep learning frameworks were used for implementation. For some deep learning concept, I'll provide equivalent keras/tensorflow code whichever one is simpler and used throughout the codebase.

3.1. Neural Network

The feed-forward Neural network is the basic building block of any deep learning system. It's composition of multiple differentiable functions. Commonly we have input vector x, apply some linear transformation to it and add bias, and finally on the result some activation function. Details on common choices for activation function are in section 3.2. In mathematical language $y = f(Ax + b)$ would be one layer of neural network transforming input x into output y. Stacking those operations we get multiple layers, hence the word deep in deep learning. Simple 3-layer neural network is depicted in figure 3.1.

$$y_1 = f(A_1 x + b_1)$$
$$y_2 = f(A_2 y_1 + b_2)$$
$$y = f(A_3 y_2 + b_3)$$

Figure 3.1: Simple three layer feed forward neural network

3.2. Activation functions

Most neural network operations are a linear transformation. Composing multiple linear transformations we get the new linear transformation. Thus have non-linear behavior we use the non-linear activation functions. Originally, the most popular choice was $\tanh$ and $\sigma(x) = \frac{1}{1+e^{-x}}$ activation functions. They are nice because of limited output domain.

However, other choices proved more effective, especially with deep neural network due to greater learning speeds, and to overcome gradient vanishing problem. Gradient vanishing refers to neural network gradient approaching zero as we backpropagate through more and more layers. Exploding gradient is related phenomena in which gradient approaches infinity. Both present serious hampering to neural network training.

ReLU, The rectified linear unit, $f(x) = \max(0, x)$, is one hugely popular choice and decent baseline compared to other ReLU variants. ReLU greatly accelerates the convergence of stochastic gradient descent compared to σ and $\tanh$ activation functions (Krizhevsky et al., 2012).

Furthermore, its calculation is drastically simpler than computing transcendental functions, like σ or $\tanh$.

Over time, ReLU showed its downsides, called *dying ReLU*. It still saturates the gradients when it's 0, that is when $x <= 0$ giving no useful gradient to back propagate. Thus several ReLU variant have been proposed: PrRelu (He et al., 2015a) in equation 3.1, ELU (Clevert et al., 2015) in equation 3.2, and finally SeLU (Klambauer et al., 2017) in equation 3.3. In Selu constants α and λ are chosen in such a way that output gravitates towards normal distribution with zero mean and unit variance. Those constants are: $\lambda = 1.0507$ and $\alpha = 1.6732$. Code generating the function plot is displayed in figure **??**, and the plot is in figure 3.3

$$PrELU(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases} \tag{3.1}$$

$$ELU(x) = \begin{cases} x & \text{if } x > 0 \\ \exp(x) - 1 & \text{otherwise} \end{cases} \tag{3.2}$$

$$\text{selu}(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{otherwise} \end{cases} \tag{3.3}$$

3.3. CNN

Convolutional Neural Networks(CNN) are the bread and butter of almost all computer vision system today, and they launched the deep learning hype with unprecedented results on Image

```
1  import tensorflow as tf
2  import keras
3  import seaborn as sns
4  import numpy as np
5  import matplotlib.pyplot as plt
6
7  for act in ["relu","selu", "elu"]:
8      with tf.Graph().as_default():
9          x = tf.placeholder(shape=(None, ), dtype=tf.float32)
10         y = keras.layers.Activation(act)(x)
11
12         with tf.Session() as sess:
13             xx = np.linspace(-1, 1)
14             yy = sess.run(y, feed_dict={
15                 x: xx
16             })
17             plt.plot(xx, yy, label=act)
18             plt.legend()
```

Figure 3.2: Code generating the activation function plot. Also shows how tensorflow and keras could be used
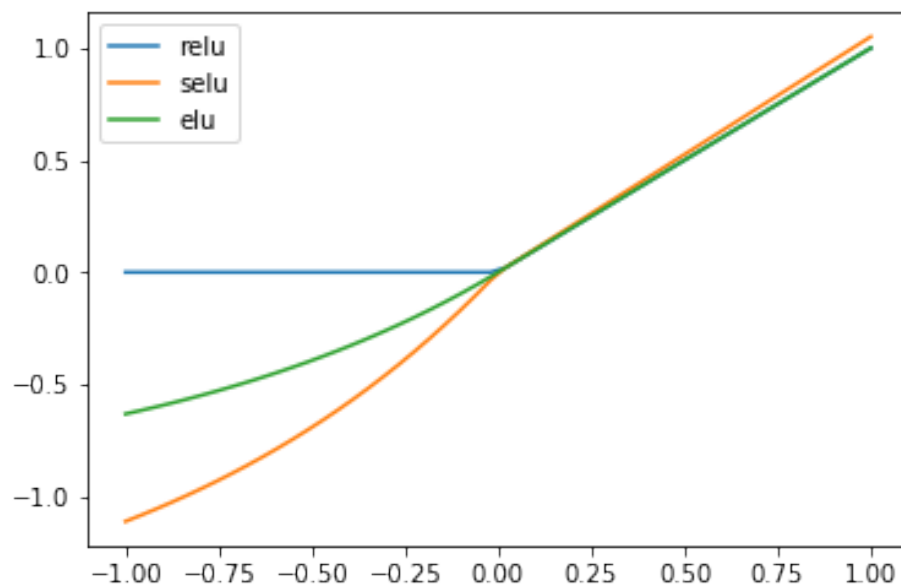
Figure 3.3: Plot of common activation functions between -1 and 1. PrELU is excluded since the parameter α is learnt during training
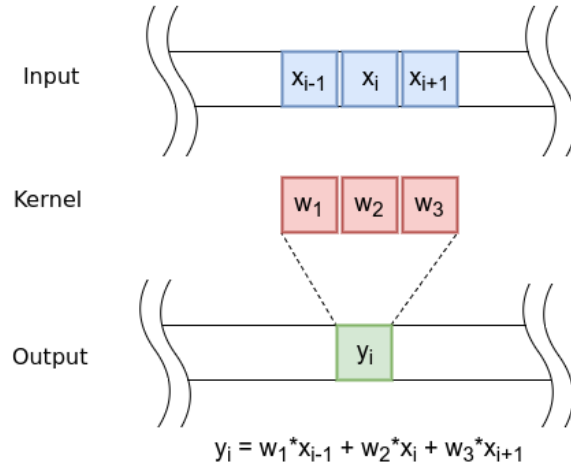
Figure 3.4: Convolution layer, kernel size 3 with stride 1. Adapted from (Ratković, 2017) with authors permission.

Classification problems. Lately, their scope is expanded to Natural language processing (NLP) tasks with promising results (Kalchbrenner et al., 2016; Gehring et al., 2017).

Convolution is a type of feed-forward neural network layer where the weights are tied together. We have a sliding window over which we apply a linear transformation of the input, and get the output element. This operation is repeated to get the full output tensor. For 1D case [1] figure 3.4 depicts it visually.

There's also atrous convolution Chen et al. (2017), also known as dilated convolution where spaced input tensor elemets are taken depending on the dilation factor. Ordinary convolution is a special case with dilation $= 1$.

3.4. RNN

RNN, residual neural networks are one of the basic building blocks in sequence models. They are feed-forward neural network spanned in a time domain. The same neural network is applied to two inputs, hidden state and input at time t and gives two outputs, new hidden state, and output at time t. The historic information remains saved in the hidden state, and it's propagated to the future. Its visual description is depicted in figure 3.5.

They can be unrolled in one massive feed-forward neural network. They are trained with backpropagation through time, on this unrolled graph. Usually the unrolling is capped and fixed number or time steps. A common issue to vanilla RNNs is vanishing and exploding gradients problem. It's solved by redesigning the basic RNN cell. There are two common approaches called LSTM (Hochreiter and Schmidhuber, 1997) and GRU (Chung et al., 2014).

---

[1]`https://keras.io/layers/convolutional`
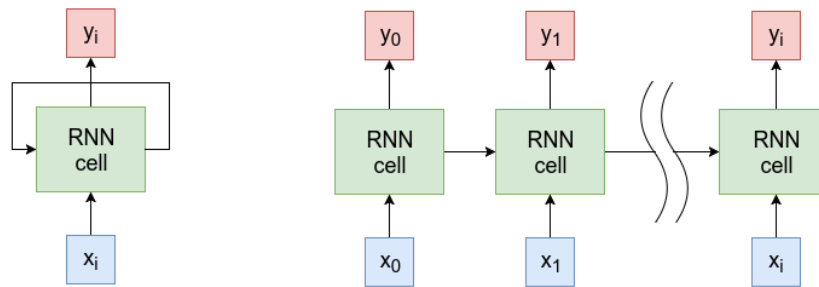
Figure 3.5: An unrolled recurrent neural network. Adapted from (Ratković, 2017) with authors permission.

Bidirectional Recurrent Neural (BiRNN) networks are used when the current output not only depends on the previous elements in the sequence but also future elements. Basically, we're stacking to RNNs, one in a forward direction and another in the backward and concatenating the hidden states/outputs. This approach was used in DeepNano (Boža et al., 2017).

Despite their modeling power, the main drawback is their speed. Since they are processing data sequentially, it's hard to parallelize those operations.

3.5. Pooling layer

Pooling layers refer to tensor dimensionality reduction. Tensor is squeezed by some factor, and each block of factor width is aggregated using some aggregation function. In other words, we divide our tensor into equally shaped hyperrectangles, and each aggregate using aggregation function yielding new hyperrectangle. Fox example, maxPool2d for image sized 50x50 shall with pool size (2,2) divides it into rectangles of size (2, 2) max those 4 elements and result in image sized 25x25. This is visually depicted in figure 3.7. Common choices are AvgPool, and MaxPool (Scherer et al., 2010) [2].

3.6. CTC loss

How do you measure how much are two sequences similar in the differentiable manner? If we were using classing encoder/decoder architecture with RNN's there's only one way to output the target sequence, and we'd optimize our RNN's outputs logits to match the target 1-hot encoded vector.

However, in this architecture, we're outputting logits whose size depend only on the input

---

[2] https://keras.io/layers/pooling/
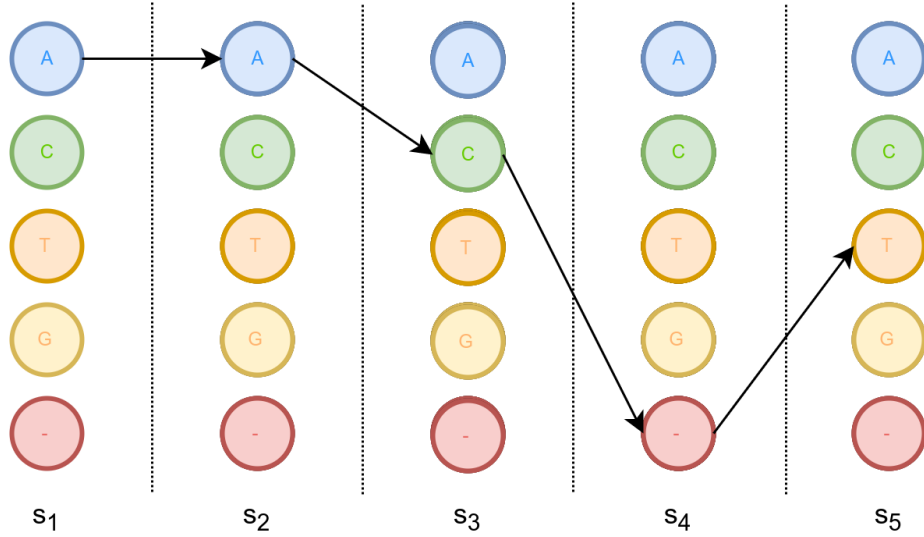
Figure 3.6: Depiction of CTC decoding. At each time step we're picking one class, and the probability of this path is the product of probabilities at each time step for that class. Equivalently, in log domain it's the sum of log probabilities, that is logits. Adapted from (Ratković, 2017)

signal size. First, let's start with some alphabet, $\Sigma = \{A, G, T, C\}$ of the output symbols we're using. Let's say the target sequence is `AAG` and we're having 5 logits due to input signal constraints. Logits have shape $[\text{sequence length}, \text{num classes}]$. What is the ideal logit values to get this target sequence to the output?. First, we must define how the sequence is decoded. The logit sequence is decoded by finding the most likely path through the logits at each time step t. Our logits can be represented as on figure 3.6. But we're presented with a problem, if we pick path `AAAAG` how do we finally decode this sequence as target `AAG`? The solution proposed in original paper for Connectionist Temporal Classification(CTC) (Graves et al., 2006; Gibiansky) is indeed brilliant. We introduce a special gap symbol, $-$. That way our logits can output `A-A-G` which shall decode in the target sequence. But what about `AA-G`? How is this decoded when there are consecutive repeated classes in the most probable path. Decoding depends on which CTC variant we used during training. There's option to `merge_repeated`, that is decode `AAA-G` as `AG`, or if it's false decode it as `AAAG` by simply dropping the blanks.

Now, let's get back to the original problem. What's the probability our sequence is `AAG` under those logits. The solution is summing probabilities for all paths which decode into target sequence. Or in mathematical language, we define function decoded as out decoding procedure. Path probability is the product of probabilities per time step in the path, as in equation 3.4. An example can be seen in equation 3.5. Then the probability of our text being equal `AAG` is given by equation 3.6. This is a procedure we can optimize with usual MLE [3],

---

[3] maximum likelihood estimator

the common loss in machine learning area.

$$P(\pi|X) = \prod_{t=1}^{m} o_t(\pi_t), \tag{3.4}$$

where $o_t(\pi_t)$ is probability of element π_t being t^{th} element on path π

$$AAG = \begin{cases} decode(A, -, A, -, G) \\ decode(A, A, --, G) \\ \vdots \\ decode(A, -, -, A, G) \end{cases} \tag{3.5}$$

$$P(y = \text{AAG}|\text{logits}) = \sum_{\pi \in decode^{-1}(\text{AAG})} P(\pi|\text{logits}) \tag{3.6}$$

3.6.1. Beam search decoding

All is fine and dandy during training, but during inference, we're left with a computationally intensive problem. Enumerating all paths lead to exponential complexity, things we'd like to avoid. Even smart dynamic programming solutions shall use more resources then feasible for any non-trivial length sequence.

To battle this problem we have 2 solutions. First is using greedy approach, and find the path such that we're picking the most probably class at each time step, and composing path out of it.

Better approximation, however, is between those two extremes. Beam search decoding Graves and Jaitly (2014). At each time step, we're only looking at top N paths up until this point. Parameter N is called beam width. From time t to $t + 1$ we have dynamic programming transitions, where we look at all possible path continuations and keep only the best N ones until reached the end. Then we output the best one as our decoded path, apply post-processing (merge repeated if configure in that mode, drop blanks) and we're done.

3.7. Autoencoder

Autoencoders are a type of neural network where the output target is the input target [4]. They funnel the data through smaller and smaller layers until we're reaching the neural network bottleneck – a thought vector as Geoffrey Hinton calls it. From there multiple layers are stacked on top of each other until we get the input back. In it's basic for they're quite simple, input x, output y and loss some loss function which measures the similarity between x and

---

[4]Unless we're talking about denoising autoencoder for which the input has additional noise

y. Variations include denoising autoencoder (Vincent et al., 2008), variational autoencoder (VAE) (Kingma and Welling, 2013), U-net (Ronneberger et al., 2015) and many others.

3.8. Multi task training

Multi-task learning [5] (Ruder, 2017; Caruana, 1997; Baxter, 2011) is approach where we're optimizing multiple tasks on the same dataset. It has been shown it may lead to better performance than learning a specific model for each task. So far it has been successfully applied in many domains, including natural language processing (Collobert and Weston, 2008), speech recognition (Deng et al., 2013), computer vision (Girshick, 2015), drug discovery (Ramsundar et al., 2015) and many other areas.

In the context of MinION basecalling, I'm combining two tasks. The first one is basecalling the sequence, that is deriving appropriate logits which shall minimize CTC loss defined in section 3.6, and autoencoder reconstruction original signal from said logits using L2 loss for distance measurement [6] between signals.

The idea is this additional loss helps models learn better logits by training on not only one but two tasks. In chapter 5 you can see the detailed analysis of this approach.
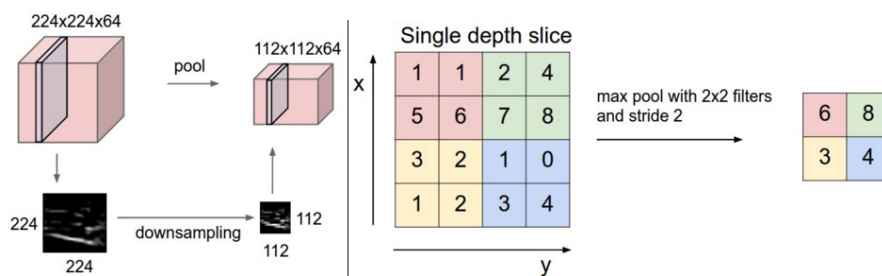
Figure 3.7: Example of max pool layer in convolutional neural network. Adapted from `https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/pooling_layer.html`

---

[5] `http://ruder.io/multi-task/`

[6] Maybe it would be better using dynamic time warping distance measurement, however, due to technical requirements of implementing in tensorflow the author chose the simpler option. This would be future work section

4. System architecture

4.1. Data Preparation

In previous project incarnation, we used fast5 with additional custom .ref files as our dataset. This setup had some issues. First, it was IO heavy since we were reading the whole fast5 file which contained non-raw data section [1], data management with separated raw files and labels, and lastly it was a magic to parse it every time. Which exactly group or subgroup within fast5, or is the signal start normalized to 0, or is it some arbitrary begging written at some other place in fast5 files changes from version to version.

Therefore, for this project, I've decided to make my own custom format for training described via protobuf (Google). The issue from previous versions wasn't the only reason. Another was ease of importing other peoples data in their formats into this one. If I can read it, I can easily write plugin converting between formats. As of writing this thesis, 4 such plugins were written since I wasn't sure which data I shall use and in which format am I going to import them in.

Ease of machine encoding/decoding as well as other previously mentioned factors guided this decision. Also, I've written datapoint format inspector operating on this common format, making my life easier. The source code for conversion procedures can be found on minion-data [2] GitHub repository as well as pypi. The specific interface description language(IDL) can be seen in figure 4.1.

4.1.1. Data correction

Each data point has been previously basecalled, and some resulting sequence was obtained. Then each sequence was corrected by aligning to the reference genome. By aligning to the reference we get the reference genome region and the CIGAR strings describing the alignment. The CIGAR string is list ordered sequence of insertions(I), deletions(D), matches(=) and mismatches(X). Insertions are bases present in the query string but absent from reference, and deletions are the opposite.

---

[1] like basecalling from other basecallers

[2] https://github.com/nmiculinic/minion-data

```protobuf
syntax = "proto3";

package dataset;

enum BasePair {
    A = 0;
    C = 1;
    G = 2;
    T = 3;
    BLANK = 4;
}

enum Cigar {
    MATCH = 0;
    MISMATCH = 1;
    INSERTION = 2; // Insertion, soft clip, hard clip
    DELETION = 3;  // Deletion, N, P
}

message DataPoint {
    message BPConfidenceInterval {
        uint64 lower = 1;
        uint64 upper = 2;
        BasePair pair = 3;
    }
    repeated float signal = 1;
    repeated BasePair basecalled = 2; // What we basecalled
    repeated BPConfidenceInterval labels = 3; // labels describe ←
    corrected basecalled signal for training
}
```

Figure 4.1: dataset protobuf description

Figure 4.2: Example of simple alignment and CIGAR string. Adapted from (Ratković, 2017)

Example alignment is depicted in figure 4.2. We take the error corrected labels and place them in dataset protobuf file for further training processing.

4.1.2. Data sources

Data has been downloaded from `https://data.genomicsresearch.org/Projects/online_dataset/train_set_all/`. The following species were provided there by the Chiron team:

– Human

– E. Coli

– Lambda Phage

For the concrete Chiron dataset, the re-squiggled preparation method was used. (The data was re-squiggled, that is after aligning the read on the reference, the read data is improved and each base pairs place on the raw signal is calculated.)

The re-squiggled basecalled data is located at `/Analyses/RawGenomeCorrected_000/BaseCa` The interesting code fragments are in function processDataPoint of file `minion_data/preperation/_` from minion-data python package.

After the gzipped dataset is prepared, it goes into the training pipeline. The whole training & testing pipeline is available open source on `https://github.com/nmiculinic/minion-basecaller`

4.2. Training pipeline

After getting data in the uniform format, the training part comes next. In aiming this as observable and simple as possible I've modeled config in using python3 `typing.NameTuple`[3] and voluptuous for data validation. With those two tools, I've created statically typed configuration file format and data validation with a human-readable error. As serialization format,

---

[3]`https://docs.python.org/3/library/typing.html`

I've chosen YAML for its readability and terseness. The config file definiton is in figure 4.3 and example config file for training is in figure 4.4. The source code can be found in module `mincall.train._train.py` and we're recommending reading it for extended details.

Input data is read via background thread which processes it, packages it into signal and label slices, and pushed into tensorflow space queues. Inside those queues, data is shuffled, and composed into batched for further training. This part of the code has an extensive test suite, including end2end tests with the golden file [4]

From configuration rest of the pipeline is initialized. This included model, with its hyperparameters, loading config defined train and test sets, learning rate schedule, sequence lengths, batch size and other settings. Keras is used for model definition due to its simplicity in design.

Additionally, I've embedded edlib (Šošić and Šikić, 2017) in the tensorflow's `tf.py_func` operation which enabled me having match, mismatch, insertion, deletion and identity rates observer during training on both train and test set.

Standard python logging module was used with multiple backends. One backend saved to log file for further inspection. The other one shipped logs to graylog system which I've set up on faculty servers. This system enabled me to log searching and greater efficiency in finding bugs.

Example command for starting the training procedure is shown in figure 4.5.

4.3. Hyperparameter optimization

Our models have various hyperparameters. For example, learning rate, receptive width size, number of blocks, etc. To gain a better model we're not only optimizing models parameters, but also hyperparameters on the validation set. For this, I've implemented hyperparam search capabilities in the programming solution.

The design rationally does as little things as possible. It takes the training configuration with injected hyperparams, and it creates a new directory and `config.yml` for each new hyperparam assignment. Then it releases control to the training code part as if we had run training on the command as previously described in figure 4.5 making this maximally observable. After the training completes it reports back to the hyperparam controller the results and according to backing hyperparameter optimization procedure new assignment is calculated.

There are two concrete hyper-optimization strategies implemented. One is random search,

---

[4]golden file is test technique where you save function output to the golden file, verify manually it's correct, and after each successive test, you're checking that nothing changed. This way we can be sure refactoring didn't influence correctness.

```python
class TrainConfig(NamedTuple):
    model_name: str
    train_data: List[DataDir]
    test_data: List[DataDir]
    logdir: str
    seq_length: int
    batch_size: int
    surrogate_base_pair: bool

    train_steps: int
    init_learning_rate: float
    lr_decay_steps: int
    lr_decay_rate: float

    model_hparams: dict = {}
    grad_clipping: float = 10.0
    validate_every: int = 50
    run_trace_every: int = 5000
    save_every: int = 2000

    tensorboard_debug: str = ""  # Empty string is use CLI debug
    debug: bool = False
    trace: bool = False

    @classmethod
    def schema(cls, data):
        return named_tuple_helper(
            cls, {
                'train_data': [DataDir.schema],
                'test_data': [DataDir.schema],
            }, data
        )
```

Figure 4.3: dataset protobuf description

```
version: "v0.1"
train:
  train_data:
    - name: "r9.4"
      dir: "./mincall/example"
  test_data:
    - name: "r9.4"
      dir: "./mincall/example_test"
  model_name: "dummy"
  model_hparams:
    num_layers: 5
  surrogate_base_pair: false
  train_steps: 60
  init_learning_rate: !!float 1e-4
  lr_decay_steps: 10000
  lr_decay_rate: 0.5
  seq_length: 40
  batch_size: 10
  logdir: "./logs"
```

Figure 4.4: training config yaml

```
1 docker run --rm -it -u="$(id -u):$(id -g)" \
2 -v $DATA_DIR:/data:ro  \
3 -v $MODEL_DIR:/model \
4 nmiculinic/mincall:latest-py-gpu train --config /model/config.yml
5
```

Figure 4.5: Example of starting training command. In the config model is properly configures to use /data and /model paths

```python
1  class HyperParamCfg(NamedTuple):
2      model_name: str
3      train_data: List[DataDir]
4      test_data: List[DataDir]
5      seq_length: Param
6      batch_size: Param
7      surrogate_base_pair: Param
8
9      train_steps: Param
10     init_learning_rate: Param
11     lr_decay_steps: Param
12     lr_decay_rate: Param
13
14     model_hparams: Dict[str, Param]
15
16     work_dir: str
17     grad_clipping: float = 10.0
18     validate_every: int = 50
19     run_trace_every: int = 5000
20     save_every: int = 2000
21
```

Figure 4.6: HyperParameter options description

and other is sigopt [5]. I've primarily used sigopt as the main hyper-optimization strategy.

Its config is similar to the training and its NamedTuple definition can be seen in figure 4.6. The param is flexible entry; it can be a single scalar value, and then it's not hyper-optimized, or it can be a dictionary with `type` field and other fields depending on the parameter types, e.g. int, float, categorical, etc.

4.3.1. Basecalling

Basecalling is performed using either beam search decoder explained in subsection 3.6.1 or greedy search decoding [6].

However, it's often infeasible passing whole signal through the GPU and the model. Then we divide the signal into overlapping stripes, calculate logits from each stipes and assembly the final read.

There are two possible strategies. We can combine the stripes in the logits space, and

---

[5]https://sigopt.com

[6]It's similar to beam search with beam width 1, that is we greedily take the most likely symbol at each time step

```
1111111122223333
11111111........
....22222222....
........33333333
```

Figure 4.7: Process of overlapping stripes in logit space. Number signal from which stripe the logit information comes and at the top composite logit is assembled.

perform beam search/greedy search on them, or we first perform beam search/greedy search on them and combine the stripes in the reading space.

I've opted out for the former option. Combining them in logits space is a pretty simple pattern – by overlaying stripes in reversed order. There's one caveat here, combining in logit space works best when the model is history independent, e.g. using only CNN layers without RNN layers which have hidden state. I have a hunch it should work alright with RNN's too but I haven't tested it.

A Picture is worth a thousand words, and this process is depicted in figure **??**

An alternative approach is used in Chiron (Teng et al., 2017) where they first decode each stripe, then assemble them. The assembly process works by overlapping consecutive stripes, thus getting relative offset, then using consensus on each output position. This approach requires a greater deal of overlap between stripes, unlike in logit space, which leads to slower basecalling. They also use RNN's, which naturally have a performance length limit. Their sequence length is around 300, and relative signal offset around 30, while in mincall sequence lengths are around 100k with offsets around 90k.

4.3.2. Final model arhitecture

The final model has residual layer architecture, with Gated residual layer residual blocks (He et al., 2015b; Savarese, 2016). It's composed of stacked blocker intertwined with max-pooling layers. Each block is composed of multiple residual layers. Whole composition is visually depicted in figure 4.8. For the best mode, `num_blocks` is 2 and `num_layers` is [TODO]. In the backward pass, in an autoencoder stage, we take the logits and use the same architecture with maxPools replaced with Upsampling1D layers. Combined loss is calculated as total loss = autoenc_coeff ∗ autoencoder L2 loss + ctc loss. During the hyperparemeter serach the best possible `autoenc_coeff` is 0. [TODO]
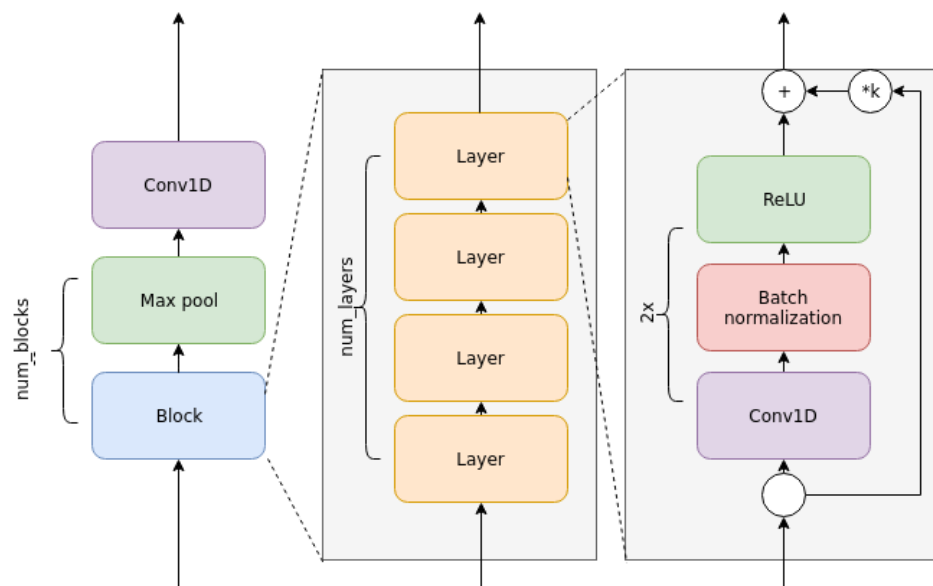
Figure 4.8: Final arhitecture overview

5. Results

5.1. Definitions

We're measuring two things. Each read is aligned to the reference genome and multiple metrics are measured, defined at the end of this section in figure **??**. However, the reads don't tell the whole picture since we're interested in the final product, consensus assembly of reads. The reads could have a systemic bias in the same regions and suffer from same errors which would yield subpar consensus versus reads correcting each other and yielding better consensus accuracy than any read on their own.

The consensus is generated from pileup using a simple algorithm. We simply align all the reads to the genome, stack them on top of each other forming pileup of read bases. Bases are called using majority vote. Deletions are handled if there's majority call for deletion, and for insertion, there has to be majority length and the bases of insertion.

In figure 5.1 shows how consensus is called from pileup created from aligned reads.

After both consensus and read, alignment is calculated, we can represent the sequence alignment with CIGAR string, that is a sequence of matches(=), mismatches(X), insertions(I) and deletions(D). Additionally, at the beginning and end of the read alignment, soft(S) and hard(H) clips may be present which are removed. Additionally, padding information(P) and spliced alignment skip cigar(N) is likewise stipped from the analysis. These conventions are taken from SAM file format definitions. In the end, we're left with the following metrics as
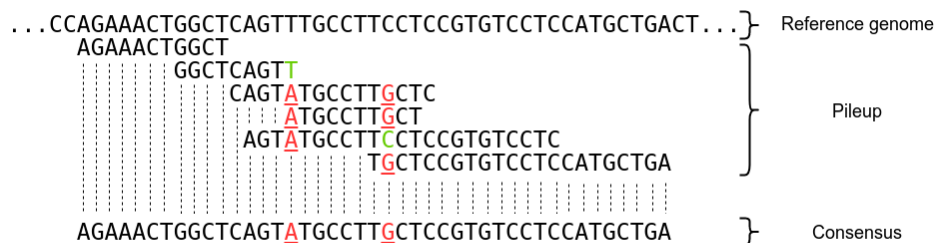
Figure 5.1: Consensus from pileup. Adapted from (Ratković, 2017)

$$read\_length = \#(=) + \#(X) + \#(I)$$

$$match\_rate = \frac{\#(=)}{read\_length}$$

$$mismatch\_rate = \frac{\#(X)}{read\_length}$$

$$insertion\_rate = \frac{\#(I)}{read\_length}$$

$$deletion\_rate = \frac{\#(D)}{read\_length}$$

$$identity\_rate = \frac{\#(=)}{\#(=) + \#(X) + \#(I) + \#(D)}$$

Figure 5.2: Various perfomance measurement metrics as defined throughout this thesis.

defined in figure **??**.

5.2. Other solutions

On his GitHub repo, Ryan Wick composed multiple basecaller comparasion (Wick et al., 2018) [1]. This section briefly summarizes his work and it's recommended to check original authors work, it's quite good! Due to resource constraints both time and compute, I'm unable to run my own testing on my data, however, I can use a subset of Ryan Wick's dataset to perform my model evaluation, thus it's somewhat relevant. This summarization is composed of two figures, read identity in figure 5.3 and consensus identity in figure 5.4. All figures are pre-polished states.

All definitions defined in the previous section are valid.

5.3. Mincall

All previously mentioned metrics are displayed here. In figure 5.5 is this solution compared to other basecallers on dataset proposed by (Wick et al., 2018). Figure **??** provides mean per read statistics for all previously defined metrics. For mincall, I've also calculated a distribution of each CIGAR operation (I, D, X, and =) over the relative reads length in figure **??**. Figure **??** reports various consensus related metrics once all reads are assembled together. Finally, I compare basecalling speeds between greedy search and beam search with the beam width of 50 in figure **??**. Here it's clearly visible how much of bottleneck beam search is.

---

[1] https://github.com/rrwick/Basecalling-comparison
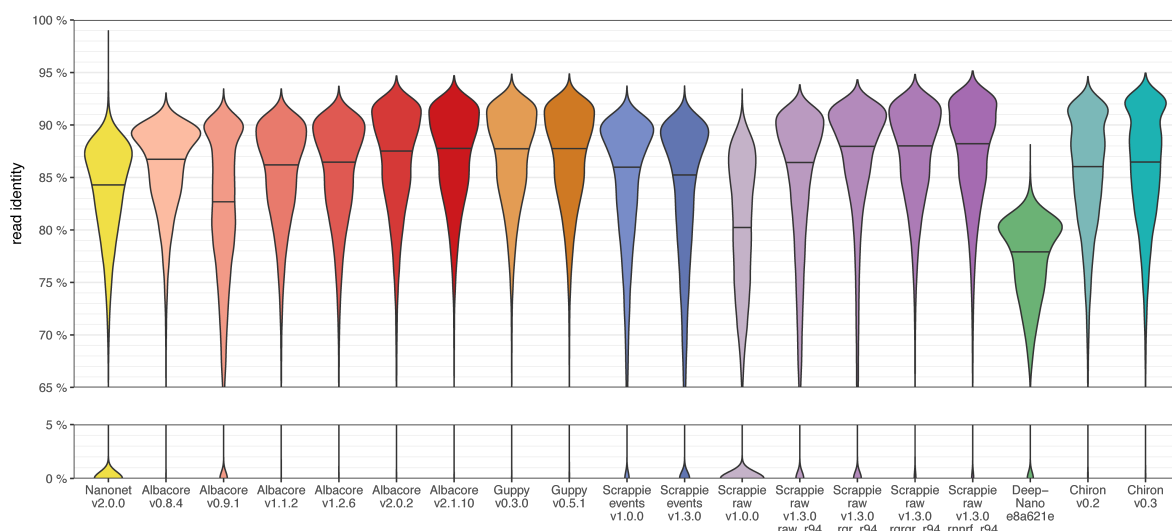
Figure 5.3: Basecallers read identity as tested by (Wick et al., 2018)
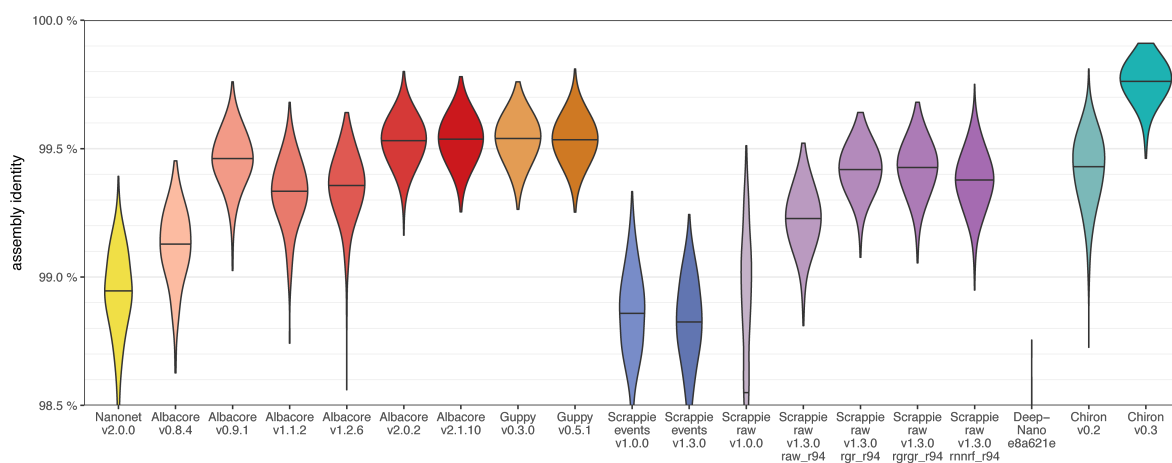
Figure 5.4: Basecallers assembly identity as tested by (Wick et al., 2018)
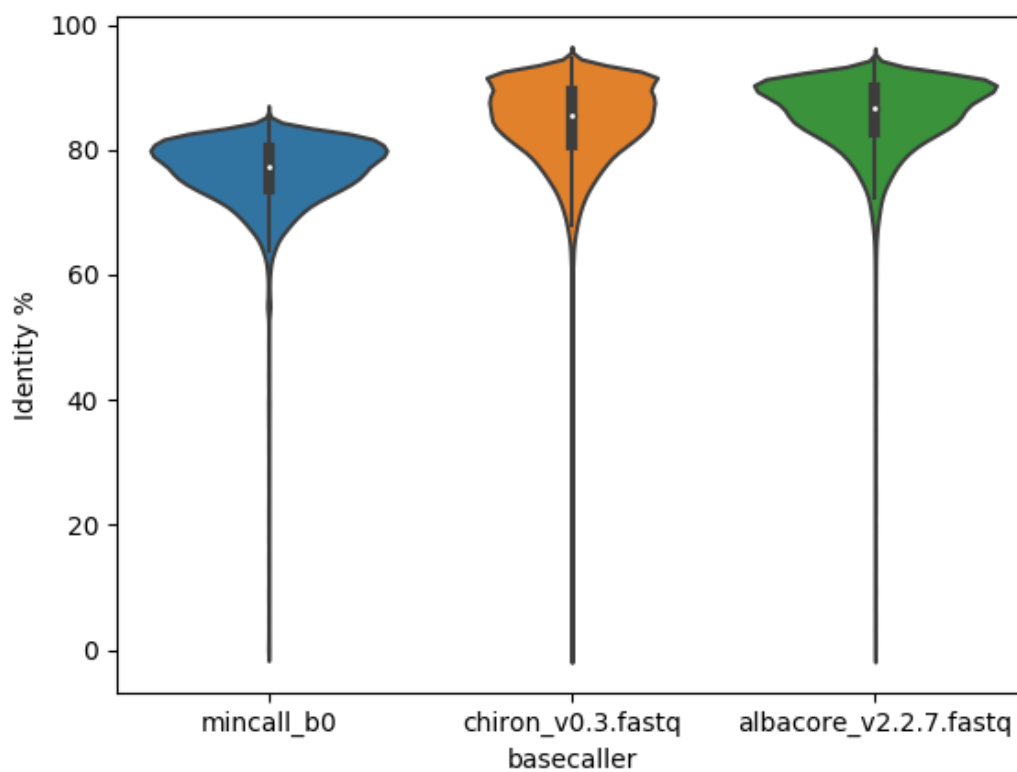
Figure 5.5: Basecallers read identity. All non-mincall ones are taken from Wick et al. (2018)

basecaller	Error %	Match %	Mismatch %	Insertion %	Deletion %	Identity %
albacore_v2.2.7.fastq	14.623744	90.164753	4.668960	5.166287	4.788497	85.107055
chiron_v0.3.fastq	15.853857	88.699263	5.306113	5.994623	4.553120	83.880875
mincall_b0	26.071517	85.939433	7.996294	6.064274	12.010950	75.979653

Figure 5.6: Mean read metrics across basecallers. All non-mincall ones are taken from Wick et al. (2018)
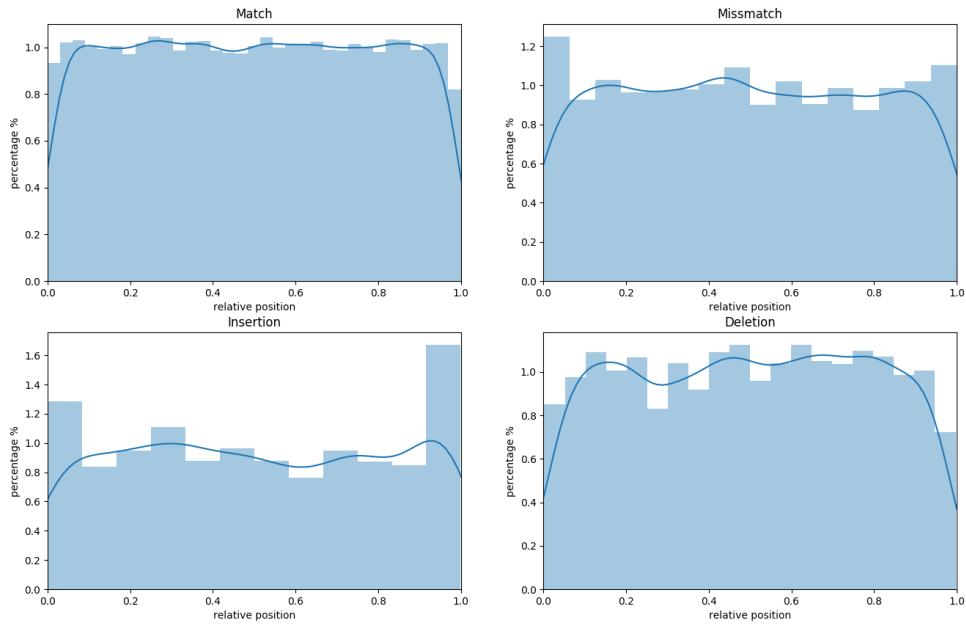
Figure 5.7: CIGAR string distribution over the relative read position

	alignment_b0	alignment_b50
coverage_threshold	0.000000	0.000000
snp_count	3558.000000	3607.000000
insertion_count	1532.000000	1536.000000
deletion_count	4841.000000	4431.000000
num_undercovered_bases	0.000000	0.000000
num_called_bases	38131.000000	38888.000000
num_correct_bases	33041.000000	33745.000000
average_coverage	0.893800	0.912106
snp_rate	9.330991	9.275355
insertion_rate	4.017728	3.949805
deletion_rate	12.695707	11.394260
correct_rate	86.651281	86.774841
identity_percentage	0.711837	0.727004

Figure 5.8: Consensus report among the tested basecallers. All non-mincall ones are taken from Wick et al. (2018)

	Speed(bp/s)
decoding strategy	
greedy search	178 412
beam search (beam width 50)	5 000

Figure 5.9: Basecaller speed depending on logit decoding strategy. All Tests were using Titan Xp GPU for logits and CPU for decoding on 32 core Intel Xeon E5-2640 @2Ghz

6. Discourse

During the development of the mincall, I've noticed a couple of things. First, as all the models are closely matched in their metrics, I suspect we've reached Bayesian error rate. Or we need the radically different approach to get further few % improvements. Since the problem is quite simple without long-term dependencies, I'd guess Bayesian error rate is the more likely reason.

For further speed improvements, Beam Search implementation is the bottleneck. Usually, it's not used on long reads, and therefore it wasn't as optimized in tensorflow as it could have been since there was no need. While monitoring the basecalling process with `htop` utility I've noticed poor CPU utilization, that is for 10 parallel beam searches I'd expect 10 CPU's having 100% utilization, but that wasn't the case indicating subpar performance. This fate befalls all using beam search as their primary decoding mechanism, including Chiron (Teng et al., 2017) and other projects. What Albacore uses underneath the hood isn't certain and I cannot comment on that.

Most of the training is done on E. Coli, and lately, this set is expanded to the human genome and few other species. It would be beneficial gathering dataset in some standardized format with quality control. I've proposed the protobuf format and described it in section **??**. Many other fields have their own Common Task Framework (CTF) benchmark, e.g. ImageNet, Cifar, etc.

7. Conclusion

[TODO] Write something smart here depending on the results.

BIBLIOGRAPHY

Marko Ratković. Deep learning model for base calling of minion nanopore reads, 2017.

Ryan Wick, Louise M Judd, and Kathryn E Holt. Comparison of oxford nanopore basecalling tools, March 2018. URL `https://doi.org/10.5281/zenodo.1188469`.

Yann LeCun and Yoshua Bengio. The handbook of brain theory and neural networks. chapter Convolutional Networks for Images, Speech, and Time Series, pages 255–258. MIT Press, Cambridge, MA, USA, 1998. ISBN 0-262-51102-9. URL `http://dl.acm.org/citation.cfm?id=303568.303704`.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL `https://goo.gl/UpFBv8`.

Alexander S Mikheyev and Mandy MY Tin. A first look at the oxford nanopore minion sequencer. *Molecular ecology resources*, 14(6):1097–1102, 2014.

Nick Loman. *Nanopore R9 rapid run data release*, a. URL `http://lab.loman.net/2016/07/30/nanopore-r9-data-release/`. [Online; posted 30-July-2016].

Nick Loman. *Thar she blows! Ultra long read method for nanopore sequencing*, b. URL `http://lab.loman.net/2017/03/09/ultrareads-for-nanopore/`. [Online; posted 9-March-2017].

Mirjana Domazet-Lošo Mile Šikić. *Bioinformatika*. Bioinformatics - course matherials, Faculty of Electrical Engineering and Computing, University of Zagreb, 2013.

Erik Pettersson, Joakim Lundeberg, and Afshin Ahmadian. Generations of sequencing technologies. *Genomics*, 93(2):105–111, feb 2009. doi: 10.1016/j.ygeno.2008.10.003. URL `https://doi.org/10.1016/j.ygeno.2008.10.003`.

Miten Jain, Sergey Koren, Josh Quick, Arthur C Rand, Thomas A Sasani, John R Tyson, Andrew D Beggs, Alexander T Dilthey, Ian T Fiddes, Sunir Malla, Hannah Marriott, Karen H

Miga, Tom Nieto, Justin O'Grady, Hugh E Olsen, Brent S Pedersen, Arang Rhie, Hollian Richardson, Aaron Quinlan, Terrance P Snutch, Louise Tee, Benedict Paten, Adam M. Phillippy, Jared T Simpson, Nicholas James Loman, and Matthew Loose. Nanopore sequencing and assembly of a human genome with ultra-long reads. *bioRxiv*, 2017. doi: 10.1101/128835. URL http://biorxiv.org/content/early/2017/04/20/128835.

Sara Goodwin, John D. McPherson, and W. Richard McCombie. Coming of age: ten years of next-generation sequencing technologies. *Nat Rev Genet*, 17(6):333–351, Jun 2016. ISSN 1471-0056. URL http://dx.doi.org/10.1038/nrg.2016.49. Review.

Camilla L.C. Ip, Matthew Loose, John R. Tyson, Mariateresa de Cesare, Bonnie L. Brown, Miten Jain, Richard M. Leggett, David A. Eccles, Vadim Zalunin, John M. Urban, Paolo Piazza, Rory J. Bowden, Benedict Paten, Solomon Mwaigwisya, Elizabeth M. Batty, Jared T. Simpson, Terrance P. Snutch, Ewan Birney, David Buck, Sara Goodwin, Hans J. Jansen, Justin O'Grady, and Hugh E. Olsen and. Minion analysis and reference consortium: Phase 1 data release and analysis. *F1000Research*, oct 2015. doi: 10.12688/f1000research.7201.1. URL https://doi.org/10.12688/f1000research.7201.1.

Matei David, Lewis Jonathan Dursi, Delia Yao, Paul C Boutros, and Jared T Simpson. Nanocall: An open source basecaller for oxford nanopore sequencing data. *bioRxiv*, 2016. doi: 10.1101/046086. URL http://biorxiv.org/content/early/2016/03/28/046086.

Vladimír Boža, Broňa Brejová, and Tomáš Vinař. DeepNano: Deep recurrent neural networks for base calling in MinION nanopore reads. *PLOS ONE*, 12(6):e0178751, jun 2017. doi: 10.1371/journal.pone.0178751. URL https://doi.org/10.1371/journal.pone.0178751.

Haotian Teng, Minh Duc Cao, Michael B. Hall, Tania Duarte, Sheng Wang, and Lachlan Coin. Chiron: Translating nanopore raw signal directly into nucleotide sequence using deep learning. *bioRxiv*, 2017. doi: 10.1101/179531. URL https://www.biorxiv.org/content/early/2017/09/12/179531.

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

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian

Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefow-icz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

François Chollet et al. Keras. `https://keras.io`, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015a.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus), 2015.

Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks, 2017.

Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time, 2016.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning, 2017.

Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *CoRR*, abs/1706.05587, 2017. URL `http://arxiv.org/abs/1706.05587`.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014. URL `http://arxiv.org/abs/1412.3555`.

Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *Proceedings of the 20th International Conference on Artificial Neural Networks: Part III*, ICANN'10, pages 92–101, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-15824-2, 978-3-642-15824-7. URL `http://dl.acm.org/citation.cfm?id=1886436.1886447`.

Alex Graves, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 369–376, New York, NY, USA, 2006. ACM. ISBN 1-59593-383-2. doi: 10.1145/1143844.1143891. URL http://doi.acm.org/10.1145/1143844.1143891.

Andrew Gibiansky. *Speech Recognition with Neural Networks.* URL http://andrew.gibiansky.com/blog/machine-learning/speech-recognition-neural-networks/. [Online; posted 23-April-2014].

Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1764–1772, Bejing, China, 22–24 Jun 2014. PMLR. URL http://proceedings.mlr.press/v32/graves14.html.

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015. URL http://arxiv.org/abs/1505.04597.

Sebastian Ruder. An overview of multi-task learning in deep neural networks. *CoRR*, abs/1706.05098, 2017. URL http://arxiv.org/abs/1706.05098.

Rich Caruana. Multitask learning. *Machine Learning*, 28(1):41–75, Jul 1997. ISSN 1573-0565. doi: 10.1023/A:1007379606734. URL https://doi.org/10.1023/A:1007379606734.

Jonathan Baxter. A model of inductive bias learning. *CoRR*, abs/1106.0245, 2011. URL http://arxiv.org/abs/1106.0245.

Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York, NY, USA, 2008.

ACM. ISBN 978-1-60558-205-4. doi: 10.1145/1390156.1390177. URL `http://doi.acm.org/10.1145/1390156.1390177`.

L. Deng, G. Hinton, and B. Kingsbury. New types of deep neural network learning for speech recognition and related applications: an overview. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8599–8603, May 2013. doi: 10.1109/ICASSP.2013.6639344.

R. Girshick. Fast r-cnn. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448, Dec 2015. doi: 10.1109/ICCV.2015.169.

Bharath Ramsundar, Steven Kearnes, Patrick Riley, Dale Webster, David Konerding, and Vijay Pande. Massively multitask networks for drug discovery. *arXiv preprint arXiv:1502.02072*, 2015.

Google. Protocol buffers. `https://github.com/google/protobuf`.

Martin Šošić and Mile Šikić. Edlib: a c/c++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*, 33(9):1394–1395, 2017. doi: 10.1093/bioinformatics/btw753. URL `http://dx.doi.org/10.1093/bioinformatics/btw753`.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015b.

Pedro H. P. Savarese. Learning identity mappings with residual gates. *CoRR*, abs/1611.01260, 2016. URL `http://arxiv.org/abs/1611.01260`.

End-to-End Deep Learning Model for Base Calling of MinION Nanopore Reads

Abstract

In the MinION device, single-stranded DNA fragments move through nanopores, which causes drops in the electric current. The electric current is measured at each pore several thousand times per second. Each event is described by the mean and variance of the current and by event duration. This sequence of events is then translated into a DNA sequence by a base caller. Develop a base-caller for MinION nanopore sequencing platform using a deep learning architecture such as convolutional neural networks and recurrent neural networks. Instead of events, use current waveform at the input. Compare the accuracy with the state-of-the-art basecallers. For testing purposes use publicly, available datasets and Graphmap or Minimap 2 tools for aligning called reads on reference genomes. Implement method using TensorFlow or similar library. The code should be documented and hosted on a publicly available Github repository.

Keywords: base calling, Oxford Nanopore Technologies, MinION, deep learning, seq2seq, convolutional neural network, residual network, CTC loss

S kraja na kraj model dubokog učenja za određivanje očitanih baza dobivenih uređajem za sekvenciranje MinION

Sažetak

Unutar uređaja MinION, fragmenti jednostruke DNA prolaze kroz nanopore, što uzrokuje promjene u električnoj struji. Struja proizvedena na svakoj nanopori mjeri se nekoliko tisuća puta u sekundi. Svaki događaj opisan je srednjom vrijednosti i varijancom struje te svojim trajanjem. Postupak kojim se takav slijed događaja prevodi u niz nukleotida naziva se određivanje očitanih baza. Razviti alat za prozivanje baza za uređaj za sekvenciranje MinION koristeći modele dubokog učenje kao što su konvolucijske i povratne neuronske mreže. Umjesto događaja na ulazu koristi valni oblik struje. Usporediti dobivenu točnost s postojećim rješenjima. U svrhu testiranja koristiti javno dostupne skupove podataka i alate GraphMap ili Minimap 2 za poravnanje očitanja na referentni genom. Alat implementirati koristeći programsku biblioteku TensorFlow (ili neku sličnu). Programski kod treba biti dokumentiran i javno dostupan preko repozitorija GitHub. **Ključne riječi:** određivanje baza, Oxford Nanopore Technologies, MinION, duboko učenje, prevođenje, konvolucijske neuronske mreže, rezidualne mreže, CTC gubitak