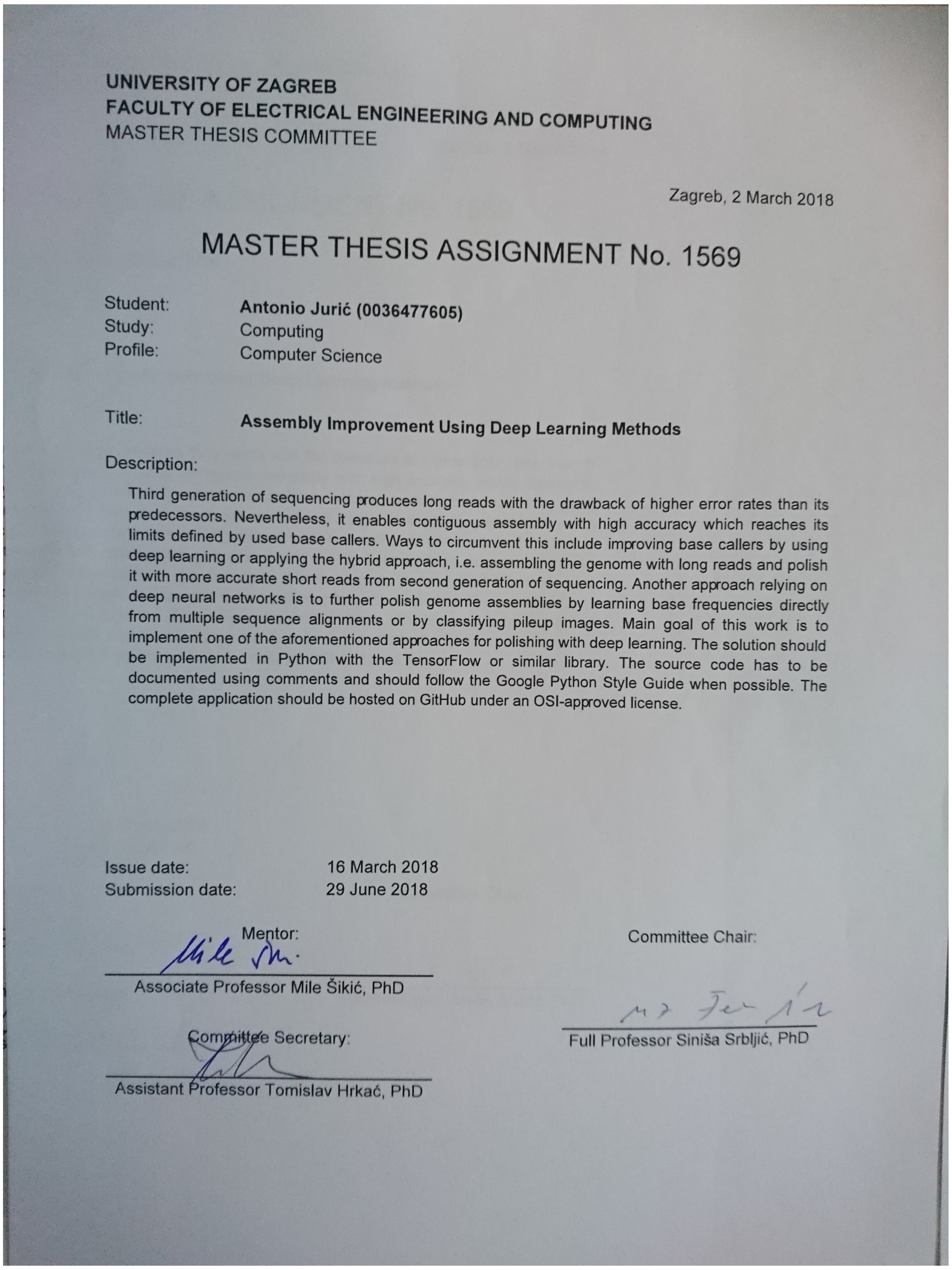
UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS ASSIGNMENT Nr. 1569

Assembly Improvement Using Deep Learning Methods

Antonio Jurić

Zagreb, September 2018.


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

Zagreb, 2 March 2018

MASTER THESIS ASSIGNMENT No. 1569

Student:	**Antonio Jurić (0036477605)**
Study:	Computing
Profile:	Computer Science

Title: **Assembly Improvement Using Deep Learning Methods**

Description:

Third generation of sequencing produces long reads with the drawback of higher error rates than its predecessors. Nevertheless, it enables contiguous assembly with high accuracy which reaches its limits defined by used base callers. Ways to circumvent this include improving base callers by using deep learning or applying the hybrid approach, i.e. assembling the genome with long reads and polish it with more accurate short reads from second generation of sequencing. Another approach relying on deep neural networks is to further polish genome assemblies by learning base frequencies directly from multiple sequence alignments or by classifying pileup images. Main goal of this work is to implement one of the aforementioned approaches for polishing with deep learning. The solution should be implemented in Python with the TensorFlow or similar library. The source code has to be documented using comments and should follow the Google Python Style Guide when possible. The complete application should be hosted on GitHub under an OSI-approved license.

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

Mentor:

Mile Šikić

Associate Professor Mile Šikić, PhD

Committee Chair:

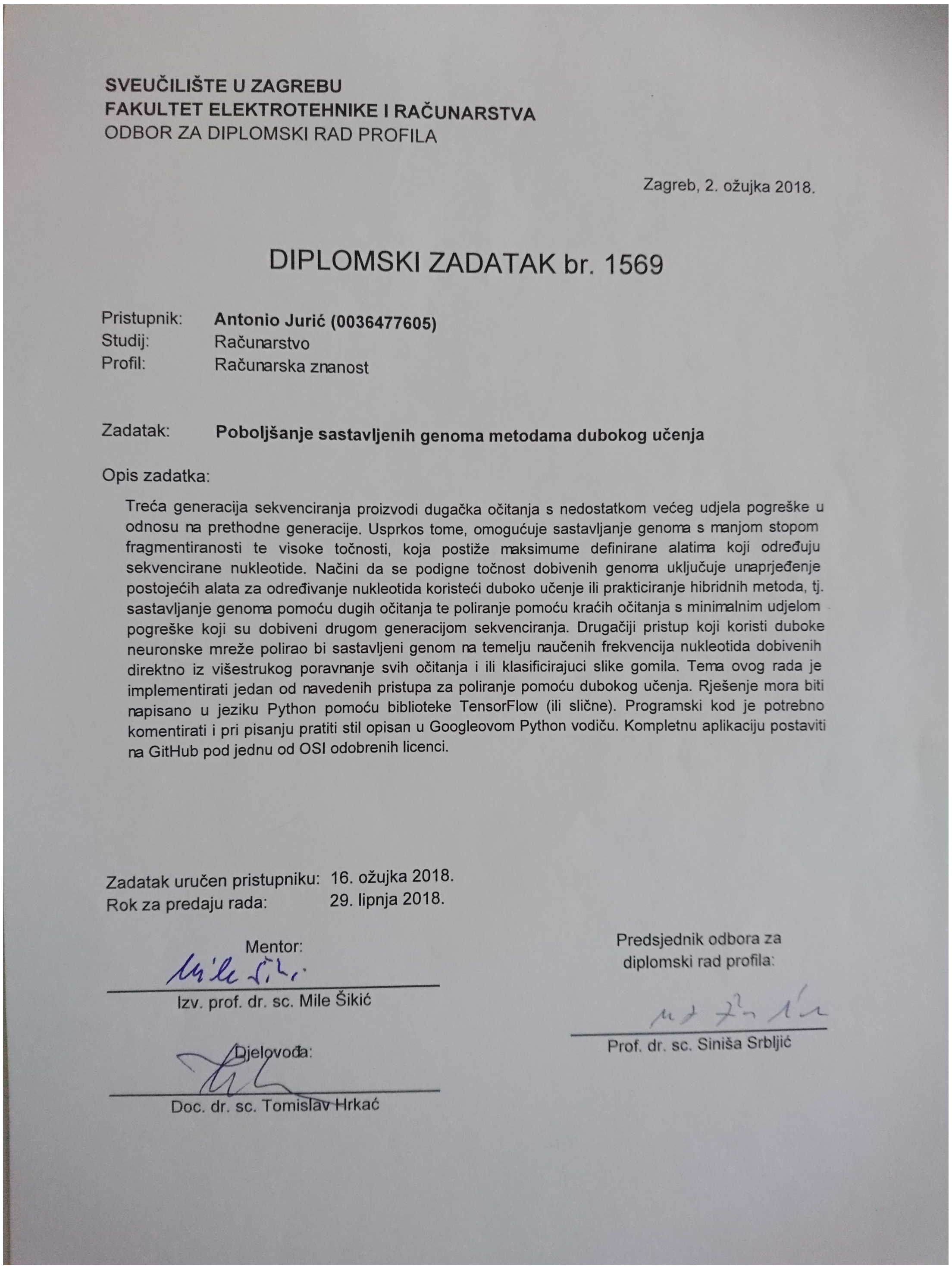
[illegible]

Full Professor Siniša Srbljić, PhD

Committee Secretary:

[illegible]

Assistant Professor Tomislav Hrkać, PhD



SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 2. ožujka 2018.

DIPLOMSKI ZADATAK br. 1569

Pristupnik:	**Antonio Jurić (0036477605)**
Studij:	Računarstvo
Profil:	Računarska znanost

Zadatak: **Poboljšanje sastavljenih genoma metodama dubokog učenja**

Opis zadatka:

Treća generacija sekvenciranja proizvodi dugačka očitanja s nedostatkom većeg udjela pogreške u odnosu na prethodne generacije. Usprkos tome, omogućuje sastavljanje genoma s manjom stopom fragmentiranosti te visoke točnosti, koja postiže maksimume definirane alatima koji određuju sekvencirane nukleotide. Načini da se podigne točnost dobivenih genoma uključuje unaprjeđenje postojećih alata za određivanje nukleotida koristeći duboko učenje ili prakticiranje hibridnih metoda, tj. sastavljanje genoma pomoću dugih očitanja te poliranje pomoću kraćih očitanja s minimalnim udjelom pogreške koji su dobiveni drugom generacijom sekvenciranja. Drugačiji pristup koji koristi duboke neuronske mreže polirao bi sastavljeni genom na temelju naučenih frekvencija nukleotida dobivenih direktno iz višestrukog poravnanje svih očitanja i ili klasificirajuci slike gomila. Tema ovog rada je implementirati jedan od navedenih pristupa za poliranje pomoću dubokog učenja. Rješenje mora biti napisano u jeziku Python pomoću biblioteke TensorFlow (ili slične). Programski kod je potrebno komentirati i pri pisanju pratiti stil opisan u Googleovom Python vodiču. Kompletnu aplikaciju postaviti na GitHub pod jednu od OSI odobrenih licenci.

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

Mentor:

[illegible]

Izv. prof. dr. sc. Mile Šikić

Djelovođa:

[illegible]

Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za diplomski rad profila:

[illegible]

Prof. dr. sc. Siniša Srbljić

Thanks to my mentor Mile Šikić for full support during the studies under his mentorship and specially during the work on master thesis. Thanks also to Krešimir Križanović and Robert Vaser.

Many thanks to the parents who were guiding me well during my life.

Also, thanks to my wife Ivana and the soon coming baby.

Finally, thanks and glory to the God Almighty, Creator of the universe, whose beauties we have the privilege of exploring.

Zahvaljujem mojem mentoru Mili Šikiću za punu podršku tijekom studiranja pod njegovim mentorstvom i posebno tijekom izrade diplomskog rada. Zahvaljujem i Krešimiru Križanoviću te Robertu Vaseru.

Posebno hvala i roditeljima koji su me dobro usmjeravali tijekom života.

Hvala i mojoj ženi Ivani te bebi koja uskoro dolazi.

Konačno, hvala i slava Bogu Svemogućemu, svemira Stvoritelju, čije krasote imamo privilegiju istraživati.

Contents

1. Introduction

Third generation sequencing technologies create reads which are much longer than those reads created with older technologies. One advantage of longer reads is that the abundant information helps to remove the uncertainty of repeating regions in the assembly. But, this long length comes at cost: longer reads have much higher error rates. When using those reads for genome assembling, one of the main problems is capturing those errors and correcting them. Even though there are some assemblers which are quite robust to such errors, there is still room to improve the correctness of the assemblies. This work tries to apply deep learning techniques in order to improve the assemblies. Code can be found at *https://github.com/ajuric/consensus-net*.

Classical pipeline for creating an assembly consists of three phases: overlap, layout and consensus phase. In the overlap phase, reads are aligned with each other. Those alignments are then used in layout phase to construct a graph structure which is then exploited to create an assembly genome. In last phase, consensus phase, assembled genome is "polished" - we traverse all genome positions and see if the base at given position matches the bases in reads aligned to that position. The most simple method used for consensus is the majority vote - the base in the genome should be the one which is in the most reads aligned to that position. Described pipeline is often referenced as Overlap-Alignment-Consensus (OLC) paradigm (Myers E.W. et al. (2000)).

Existing classical consensus tools use handcrafted rules for calculating the consensus. The risk with this method is that it is quite possible that not all cases will be covered with the handcrafted rules. Instead, this work will let the deep convolutional neural network to learn those rules by itself.

Second chapter represents the related work in the field - both using the classical and deep learning methods. Environment used for running the experiments is described in the third chapter. Fourth chapter describes various workflows, pileup creation versions, models and data shapes. Results along with datasets analysis and ablation study are in the fifth chapter. Future ideas and possible work are represented in the sixth chapter. Finally, conclusion in the seventh chapter concludes the results of this work.

2. Related work

Existing tools for consensus phase are based on statistics, classical algorithms and/or machine learning. Some of the popular tools for consensus are Nanopolish, Quiver (and Arrow), Racon, and on of the recent tools, Medaka.

Nanopolish (Loman et al. (2015)) tool has multiple purposes, one of which is a calculating an improved consensus. It can also detect base modifications, call SNPs and indels with respect to a reference genome and more. It is developed for Oxford Nanopore MinION data. The method used in Nanopolish is the Profile Hidden Markov Model. At the time, method improved assembled genome of E. Coli from 98.4% to 99.4% nucleotide identity.

Quiver and **Arrow** (PacificBiosciences (2018)) are methods developed by PacificBiosciences and for PacBio data. Quiver was first version of method. It is a consensus model based on a conditional random field approach. It's accuracies were approaching or even exceeding Q60 (one error per million bases). But, over the years Quiver has proven difficult to train and develop so it was replaced by Arrow - improved consensus model based on a more straightforward hidden Markov model approach.

Racon (Vaser et al. (2016)) is an ultrafast consensus module for raw de novo genome assembly of long uncorrected reads. It can be used as a polishing tool after the assembly with either Illumina data or data produced by third generation of sequencing (PacBio and Oxford Nanopore MinION data). It uses SIMD accelerated, partial order alignment based stand-alone consensus module which when coupled with Miniasm enables consensus genomes with similar or better quality than state-of-the-art methods while being an order of magnitude faster.

Medaka (Nanoporetech (2018)) is a new tool for error correcting sequencing data, particularly aimed at nanopore sequencing and the first known tool to use deep learning methods for consensus phase (even though it is not finished yet). It is developed for both training and inference. The model consists of two GRUs. Alignments in dataset are converted to run-length encoding. On some assemblies it gave better results than Racon and Nanopolish and, also, run time was faster. Those benchmarks can be found on documents page of Medaka.

Similar problem to consensus is variant calling - problem in which we have a reference genome and then try to find differences between that reference genome and assembled genome. Output is positions which differ in bases - variants. Since this problem is even more similar to consensus problem in terms of deep learning, it is definitely worth mentioning and studying variant calling methods which use deep learning: DeepVariant, VariantNet and Clairvoyante.

DeepVariant (Poplin et al. (2018)) is a pioneer work for using deep learning in variant calling. It represents a significant step from expert-driven statistical modeling towards more automatic deep learning approaches for developing software to interpret biological instrumentation data. Instead of hand-crafting parametrized statistical models which still produced a lot of errors despite invested effort, DeepVariant uses deep convolutional neural network to call genetic variation in aligned next-generation sequencing read data by learning statistical relationships (likelihoods) between images of read pileups around putative variant sites and ground-truth genotype calls. Convolutional neural network was selected due to the great results in image processing in computer vision, so dataset for variant calling problem was formed by creating images of read pileups. DeepVariant won "highest performance" award for SNPs in a FDA-administered variant calling challenge. Method was developed to perform well not only on a specific technology, but on a variety of sequencing technologies.

VariantNet (Chin (2017)) is a simple convolutional network inspired by DeepVariant. It's motivation was a loss of information in dataset during conversion of read pileups to images in DeepVariant. Instead of creating images of read pileups, VariantNet uses one-hot-like matrices to encode counting number of bases (A, C, G and T) and differences between reads and reference for consecutive positions - the similar dataset design is used in this work. Advantage of such dataset desing is much smaller neural network: VariantNet uses only two convolutional and 3 fully-connected layers in contrast to DeepVariant which uses large neural network developed for image classification.

Clairvoyante (Luo et al. (2018)) is recent work in variant calling using deep learning - it uses multi-task 5-layered convolutional neural network model for predicting variant type (SNP or indel), zygosity, alternative allele and indel length from aligned reads. It gave slightly better results than DeepVariant (e.g. on HG001 dataset Clairvoyante had 97.65% precision, 96.53% recall and 97.09% F1-score, and DeepVariant had 97.25% precision, 90.73% recall and 92.67% F1-score). Method is developed for Ilumina, PacBio and Oxford Nanopore MinION data.

3. Environment

Nowadays more and more tools have multiple different dependencies on other tools, frameworks, programming languages and drivers. Common case is installing different versions of same tool on the same machine. This increased complexity of developing makes projects very hard to maintain and almost impossible to reproduce on other machines.

Container technology is popular and practical solution in this cases, especially in deep learning where listed problems are faced daily. Containers allow isolating programs and packaging them with their dependencies. One of the other advantages of containers is their fast start.

Container technology used here is Docker (Babak et al. (2017)). Created Docker image used in these experiments contains appropriate CUDA, cuDNN, TensorFlow versions, appropriate Python module dependencies (Pysam, Pysamstats, Keras, CometML, ...) and installation of various bioinformatics data handling tools (Minimap, Minimap2, Racon, ...). Dockerfile can be found in provided Github repository so all dependencies can be found in it. Further sections will describe each of those dependencies.

Exact version of Docker used is Nvidia-Docker2 (NVIDIA (2018)) runtime. Nvidia-Docker2 runtime allows the usage of NVIDIA GPUs in the containers. If there are multiple GPUs on a single machine, this runtime allows configuring containers to use only specified GPUs by exposing only those GPUs in a container. This can be achived with '***NV_GPU=X***' parameter in command line arguments where X is the GPU number which can be found with '***nvidia-smi***' command (if more GPUs need to be used, numbers are split with ','). This is helpful since default behaviour of TensorFlow is to occupy all GPUs on a machine, even in cases when only one GPU is used.

4. Methods

This chapter describes different models used in experiments and various workflows which are the same for all models. Those workflows include dataset creation, model training and inference (new consensus). There are some steps which are shared between different workflows. Most notable one is pileup creation which has three modes of work.

4.1. Workflows

All workflows described here have corresponding scripts in code repository.

4.1.1. Dataset workflow

Figure 4.1 shows workflow for creating a dataset which will be used for training. Input to this workflow is a list of pairs of references and corresponding reads.

For each pair, reference and its corresponding reads, following process is executed. Reads are aligned to reference using Minimap2 (Li (2018)) because Minimap2 can output an alignment in SAM format (Wikipedia (2016b)). (‘***-ax map-pb/ont-pb -t***‘ are command line arguments for Minimap2 for alignment specifying PacBio or Nanopore data and the number of threads for running). SAM format is then converted to BAM format (Wikipedia (2016a)) using SAMtools (Research (2009)) (‘***view***‘ command in SAMtools). Alignment in BAM format is then sorted by leftmost coordinates (‘***sort***‘ command in samtools), and, finally, index is created out of sorted BAM alignment (‘***index***‘). That index is used by some tools which create pileups (Pysam (developers (2009)) and Pysamstats (Miles (2012)) - both tools are described in Pileups creation section).

Sorted BAM alignment and index are used in the next step by *PileupGenerator* to create pileups. Pileups are shown in figure 4.2. For each position on reference, we store the number of bases of reads aligned to that position. E.g. if there are 10 reads

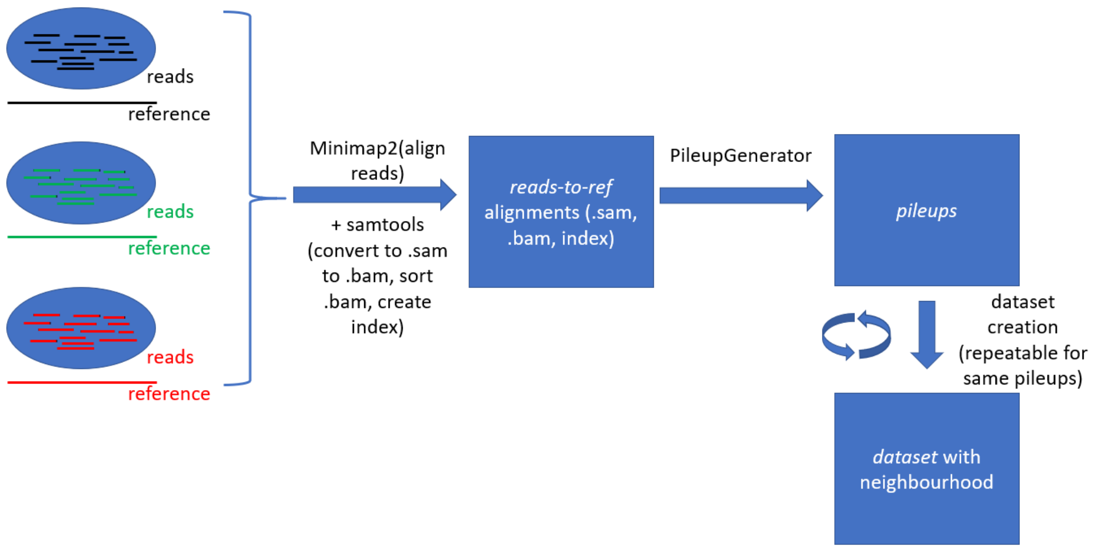
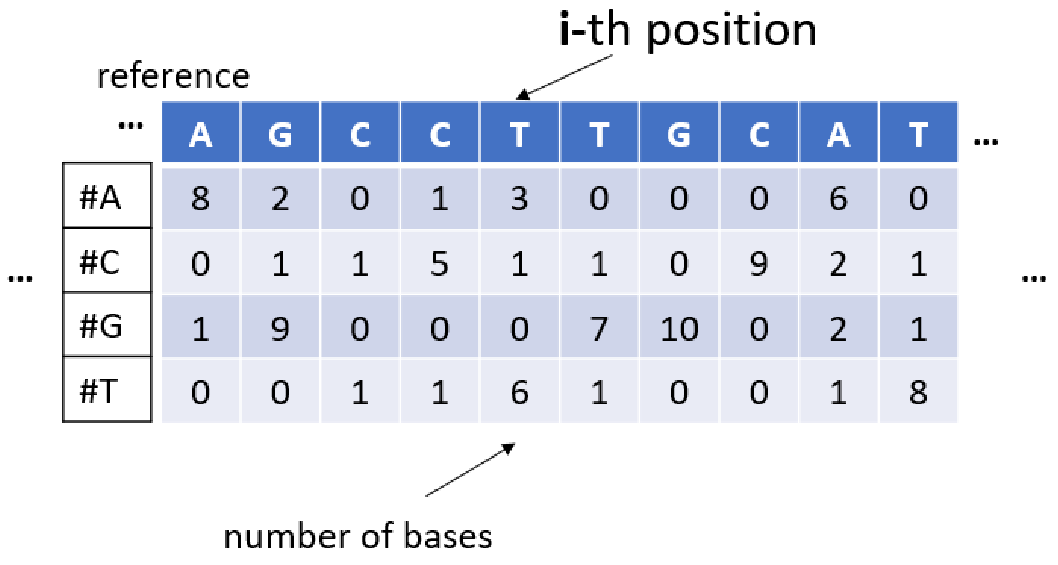

reads
reference
reads
reference
reads
reference
Minimap2(align reads)
+ samtools (convert to .sam to .bam, sort .bam, create index)
reads-to-ref alignments (.sam, .bam, index)
PileupGenerator
pileups
dataset creation (repeatable for same pileups)
dataset with neighbourhood

Figure 4.1: Dataset workflow - shows how dataset is being created from reference and reads pairs

aligned to some position ***i*** and 3 of those reads have base 'A' at that position, 1 read has base 'C' at that position and 6 reads have base 'T' at that position, then numbers for that pileup column would be 3, 1, 0 and 6 with respect to bases 'A', 'C', 'G' and 'T'. Those numbers of bases will be used to create samples (***X***), and bases in references will be used to create labels (***y***) for those samples.

Note that the reference could consist of multiple contigs due to various biological reasons or due to the reference reconstruction errors caused by the assembler tools. Above description for reference processing could be seen as situation in which there is only one contig. In situation with multiple contigs, each of those contigs is processed in the same manner as described above for the reference processing. After all contigs are processed, all pileups from different contigs are concatenated in one list as if there are no contigs (or, from another perspective: as if there is only one contig). This is possible because the source of one pileup column (the exact contig) is irrelevant for the model training. Multiple contigs are shown on figure 4.3.

The final step of this workflow is creating a dataset. At the start we select a *neighbourhood_size*. *neighbourhood_size* specifies, when we are at some position ***i***, how much positions do we consider to the left and to the right of our current position ***i***. This results in creating a sample which covers a *2*neighbourhood_size+1* positions in total. This is showed in figure 4.4. Final shape of samples is *(2*neighbourhood_size+1, R)* where *R* is the number of rows shown on figure 4.4 and it depends on *PileupGenerator*. (Actually, more intuitive shape which resembles the situation on figure 4.4 would be


i-th position
reference
number of bases

...	A	G	C	C	T	T	G	C	A	T	...
#A	8	2	0	1	3	0	0	0	6	0	
#C	0	1	1	5	1	1	0	9	2	1	...
#G	1	9	0	0	0	7	10	0	2	1	
#T	0	0	1	1	6	1	0	0	1	8	

Figure 4.2: Pileups

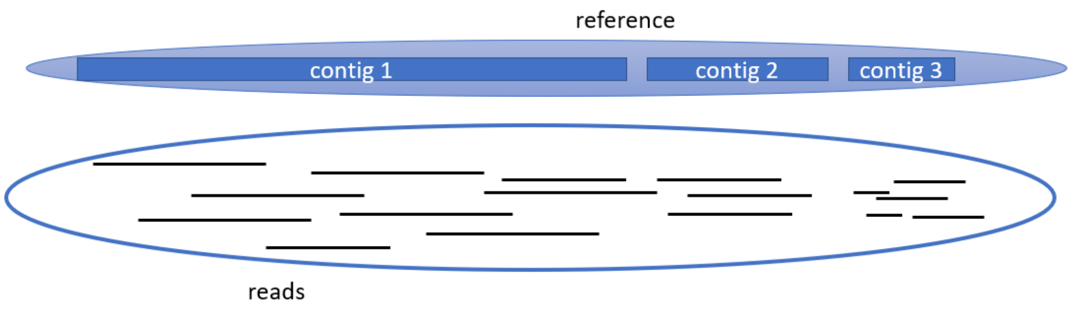
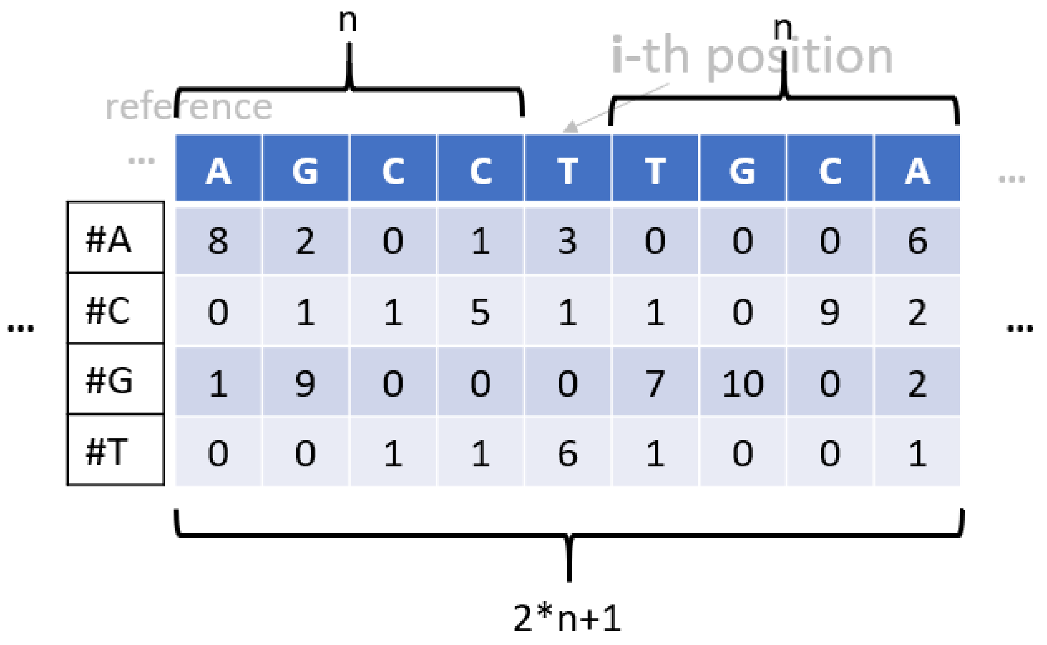


Figure 4.3: Reference consisting of multiple contigs

*(R, 2*neighbourhood_size+1)*, but Keras tool needs it this way. More about Keras input shapes can be found in section about models.) Constructing a sample in such way, we hope to provide more information to model when trying to learn the mapping between samples and labels, i.e. for each position ***i***, we are trying to predict from sample the base at position ***i*** in reference using the pileup at that position and the neighbouring pileups as help.

After samples are created for each reference and reads pair, those samples are mixed together in order to make dataset more general (in contrast to having samples from only one reference in dataset which could easily lead to overfitting to that particular reference). Notice that pileups for some reference-read pair and *PileupGenerator* is always the same. Hence, pileups could be reused to create datasets with different *neighbourhood_size*. They are computed only once per reference-read pair and *Pile-*



	A	G	C	C	T	T	G	C	A
#A	8	2	0	1	3	0	0	0	6
#C	0	1	1	5	1	1	0	9	2
#G	1	9	0	0	0	7	10	0	2
#T	0	0	1	1	6	1	0	0	1

Figure 4.4: Sample

upGenerator and stored on a disk.

4.1.2. Training workflow

Figure 4.5 shows workflow for training the models. Input to this workflow is a dataset and list of models to be trained.

Training queue manages the whole training process: it takes a model one by one from the list of models, starts the training of that model by providing the model with dataset and saves the trained model after training. Saved model is used for inference in inference workflow.

During the training, various information is uploaded on CometML (developers (2018)) - framework which captures the training progress, saves the training metrics and provides visualization of training process. Such tool is inevitable since a lot of experiments are run and without it lot of results would be just lost. Key feature of CometML is that in order to start tracking the training progress almost no code needs to be modified: user just needs to import CometML at the beginning of the Python code and provide it with his user key which is always the same for the one specific user.

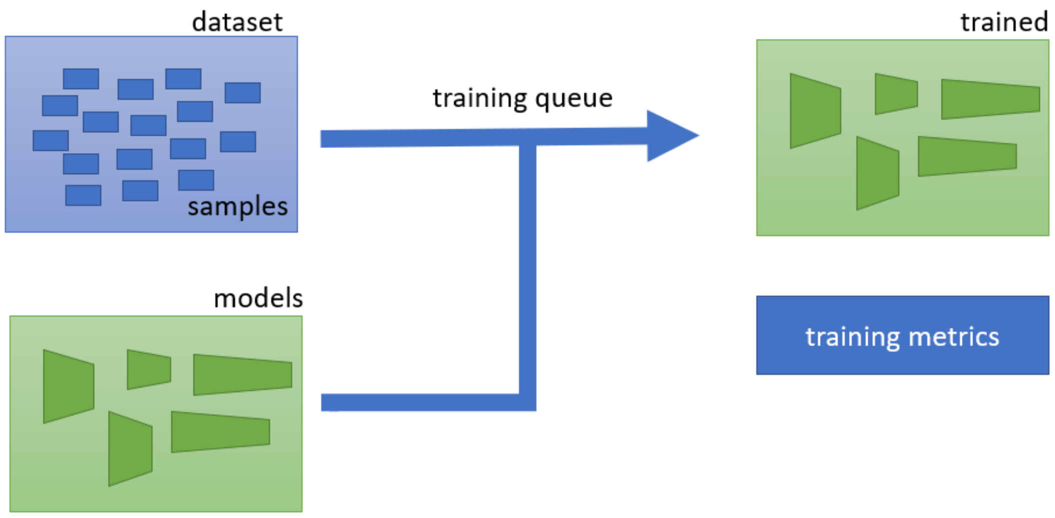


Figure 4.5: Training workflow

4.1.3. Inference workflow

Inference workflow consists of two parts: assembly creation part, shown on figure 4.6, and inference part, shown on figure 4.7. Input to inference workflow is reference and corresponding reads. In real usage, reference could be omitted, but in these experiments reference is used for calculation of assembly identity with reference before and after polishing it with deep neural network.

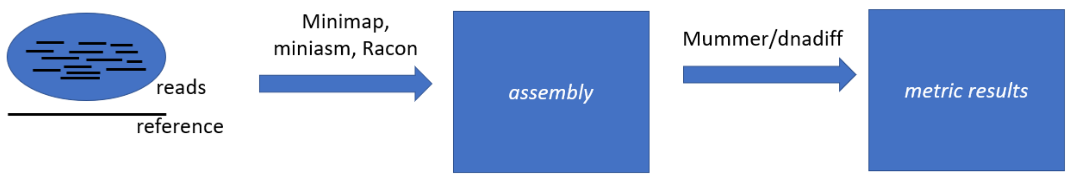


Figure 4.6: Assembly creation part of the inference workflow

In the **assembly creation part** of inference workflow "blade script" is used to create an assembly. Blade script follows the OLC paradigm in the assembly creation. First, Minimap (min) is used to calculate the overlaps between reads. In layout phase, Miniasm (min) is used to calculate the assembly graph. In consensus phase, Racon is used to polish the assembly with two iterations of polishing. Note that deep learning model consensus polishing comes after another consensus tool. The idea is to try to polish the assembly after another consensus tool so that the identity measure of assembly to reference rises even more. Also, other consensus tools usually follow hand-crafted polishing rules, while with deep learning model polishing rules are being

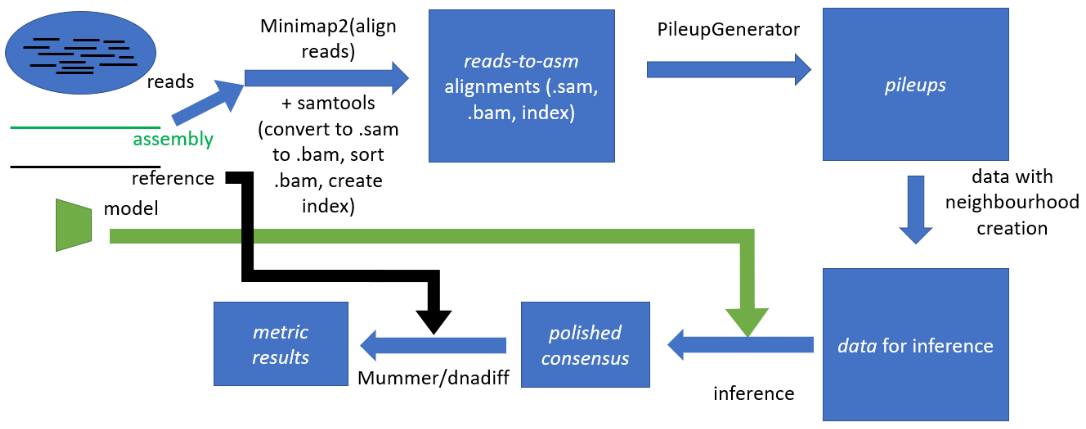


Figure 4.7: Inference part of the inference workflow

learned automatically. The last step in this part is the calculating the identity measure with Mummer tool (Kurtz et al. (2004)) (using '***dnadiff***' command): it compares reference and assembly and calculates the identity measure.

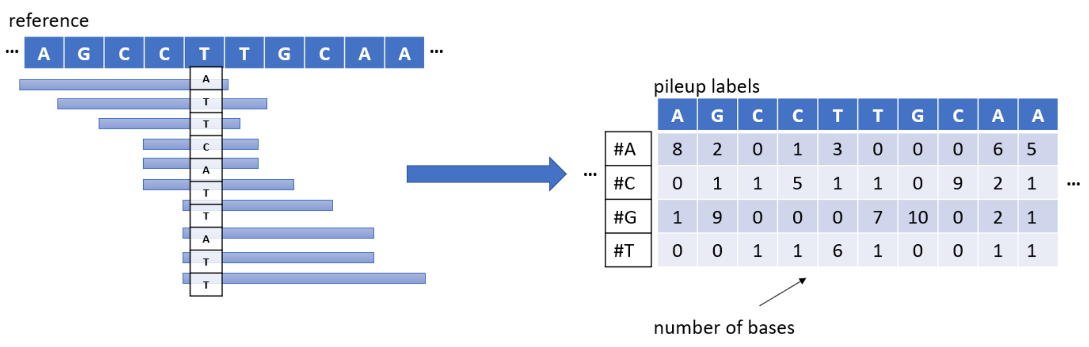
The **inference part** of inference workflow shares a lof of steps with dataset workflow as can be seen in figure 4.7. The input to this part are reads and assembly from previous part of this workflow. First, reads are aligned to the assembly with Minimap2 and the aligment is converted to BAM file in the same manner as in dataset workflow. Sorted BAM alignment and index are used in the next step by *PileupGenerator* to create pileups. As opposite to the dataset workflow, pileups from different contigs are kept separate so that the final consensus after the neural network polishing has the same number of contigs as the assembly. After pileups creation, same procedure for dataset creation is used as in dataset workflow to prepare data for inference. Prepared data needs to have the same *neighbourhood_size* as the model which will be used for inference (that model was trained on dataset with the same *neighbourhood_size*). Output of this step is the polished consensus with the neural network. The polished consensus is then processed with Mummer together with the reference to calculate the identity measure. Two identity measures are compared (one from assembly creation part and the other from this part) to check the consensus quality after the polishing with neural network.

4.2. Pileups creation

Pileups creation is used in dataset and inference workflow and is implemented in *PileupGenerator* class. Configuration of pileup directly affects the dataset and model performance - if pileup is constructed well (contains important information and that information is represented well), model will perform better. Three version of pileup creation are supported: overlap with no indels, overlap with indels and MSA supported overlap with indels. All versions support multiple contigs: contigs are processed separately and, depending on the dataset or inference workflow, they are concatenated (dataset workflow) or they are passed to following steps separated (inference workflow).

4.2.1. Overlap with no indels

This version is implemented in *PysamstatsNoIndelGenerator* class. It provides the pileups as described in dataset workflow. For each position on reference, we only store the number of bases of reads aligned to that position. Indels are ignored. E.g. if there are 10 reads aligned to some position $\boldsymbol{i}$ and 3 of those reads have base 'A' at that position, 1 read has base 'C' at that position and 6 reads have base 'T' at that position, then numbers for 6 that pileup column would be 3, 1, 0 and 6 with respect to bases 'A', 'C', 'G' and 'T'. The label for that column would be the base in reference at position $\boldsymbol{i}$. Such pileup is show on figure 4.8.



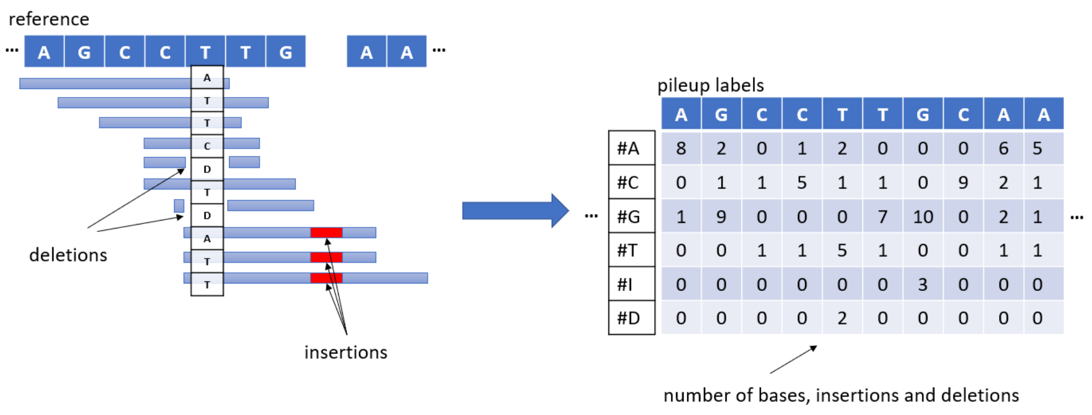
	A	G	C	C	T	T	G	C	A	A
#A	8	2	0	1	3	0	0	0	6	5
#C	0	1	1	5	1	1	0	9	2	1
#G	1	9	0	0	0	7	10	0	2	1
#T	0	0	1	1	6	1	0	0	1	1

Figure 4.8: *Overlap with no indels* version of pileups

Pysamstats is a tool based on Pysam tool. Tools provide information needed for pileup creation - for each position on reference, tools can provide the exact number of bases in the alignment. Pysamstats provides aggregated results for each position, while Pysam provides a list of reads for each position which then needs to be processed to extract the number of each base. Therefore, Pysamstats is faster in execution and is used in this work.

4.2.2. Overlap with indels

This version is implemented in *PysamstatsIndelGenerator* class. It differs from the above version in a way that it stores indels in the following manner. For each position on reference, Pysamstats provides the number of bases, but it also provides the number of deletions and insertions. Pileup column now has two more rows: 4 rows as before for bases and one row for insertions and one row for deletions as shown in figure 4.9. Label is now not just the base in the reference at current pileup position. If the number of insertions or deletions for that position is greater then the number of all other bases ($num_I > max(num_A, num_C, num_G, num_T)$ or $num_D > max(num_A, num_C, num_G, num_T)$), then label is insertion or deletion, depending which number is greater. Otherwise, label is the base in the reference at that position.


reference
... A G C C T T G A A ...
A T T C D T D A T T
deletions
insertions
pileup labels
number of bases, insertions and deletions

	A	G	C	C	T	T	G	C	A	A
#A	8	2	0	1	2	0	0	0	6	5
#C	0	1	1	5	1	1	0	9	2	1
#G	1	9	0	0	0	7	10	0	2	1
#T	0	0	1	1	5	1	0	0	1	1
#I	0	0	0	0	0	0	3	0	0	0
#D	0	0	0	0	2	0	0	0	0	0

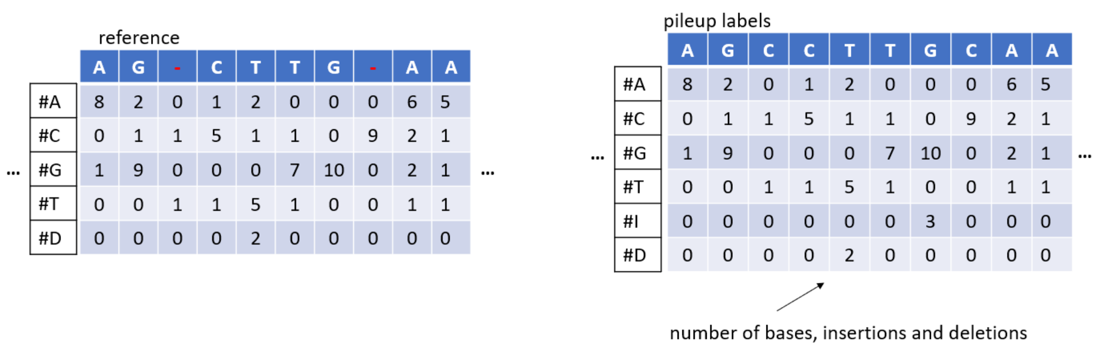
Figure 4.9: *Overlap with indels* version of pileups

Note that insertions and deletions in this work are considered in regards to reads. Deletion means that there is one base missing in the read so there is a gap in the read. Insertion means that there is one base inserted in the read so there is a gap in the reference. When Pysamstats reports number of deletions for some position, it just counts the deletions in reads on that position. Since Pysamstats only iterates over positions in reference, reporting number of insertions is not that trivial. Hence, when Pysamstats reports the number of insertions for some position, it counts how many reads have an insertion sequence on the right of that position. All contigs are processed separately.

4.2.3. MSA supported overlap with indels

This version is implemented in *RaconMSAGenerator* class. Previous version didn't consider the alignment between the reads, but only the alignment of reads to the reference. This version considers the alignment between the reads also. By using the sliding window technique when looking at the alignment to the reference, it tries to align reads between themselves to make pileups more accurate. Described technique is know as Multiple Sequence Alignment (MSA).

Modified version of Racon tool is used to provide MSA. It outputs the textual file which is organized in the following way. Every six lines represent the following: contig, number of 'A' bases, number of 'C' bases, number of 'G' bases, number of 'T' bases and the number of deletions. Insertions are encoded into contig. As insertion is the gap in the reference, it is represented by empty position in the contig. MSA output file is shown in the figure 4.10. When calculating the label for some position, if there is a gap in the contig (insertion), it is labeled as insertion. Otherwise, if number of deletions for that position is greater than number of all bases, pileup is labeled as deletion. At last, if non of the previous two cases are filled, pileup is labeled as the base in the contig for that position.


reference
...
...
pileup labels
...
...
number of bases, insertions and deletions

	A	G	-	C	T	T	G	-	A	A
#A	8	2	0	1	2	0	0	0	6	5
#C	0	1	1	5	1	1	0	9	2	1
#G	1	9	0	0	0	7	10	0	2	1
#T	0	0	1	1	5	1	0	0	1	1
#D	0	0	0	0	2	0	0	0	0	0

	A	G	C	C	T	T	G	C	A	A
#A	8	2	0	1	2	0	0	0	6	5
#C	0	1	1	5	1	1	0	9	2	1
#G	1	9	0	0	0	7	10	0	2	1
#T	0	0	1	1	5	1	0	0	1	1
#I	0	0	0	0	0	0	3	0	0	0
#D	0	0	0	0	2	0	0	0	0	0

Figure 4.10: *MSA supported overlap with indels* version of pileups

4.3. Models

During experimenting, lot of model architectures were tested. Some of them showed poor performance, some of them were good. Hence, here are described only four out of 30 models tested. When hyperparameter tuning is taken into account, actual number of tested architectures is much greater. Model naming was their serial number of creation and does not have any special meaning.

One specificity of designed models is their size - such models are called slim models: 3 out of 4 described models have only 10^4 parameters, while 1 model has $0.5M$ parameters. Small size enables fast training and inference. But, despite fast inference, the inference workflow takes a couple of minutes for bacteria with genome size of few million bases because a lot of time is spent in other steps (preprocessing - creating pileups and data with appropriate *neighbourhood_size* - and postprocessing - converting predictions to nucleus bases).

All described models have the same following settings. Adam (Kingma i Ba (2014)) is the algorithm used for optimizing the cross-entropy loss. Batch size was 10000. Models were trained for 150 epochs with early stopping (Goodfellow et al. (2016)) criterion set to stop the process if there were no improvement in validation loss for 3 consecutive epochs. Early stopping is actually one way of the implicit regularization. In the experiments, early stopping usually ended the training long before $150th$ epoch - experiments were stopped around $20th$ epoch. The learning rate policy was set to reduce the learning rate after 50 epochs - every 5th epoch after 50th epoch, learning rate was set to 0.95 value of the current learning rate. As can be seen, this learning rate policy rarely took effect due to the early stopping.

Abbreviations for layers parameters used in the model figures are the following. For convolutional layer, *f* is the number of filters (output channels), *p* is padding, *ks* is kernel size, *s* is the stride and *reg* is the kernel regularization - weight decay - in form of L2-norm. For polling layer, *ps* is the pool size and *s* is the stride. For dense layer, *o* is the number of neurons. For dropout layer, *p* is the fraction of input units to drop.

4.3.1. Model 7

Model 7 is shown in figure 4.11. This slim model consists only of two 1D-convolutional layers, one maximum polling layer (after the first convolutional layer) and one dense block. Activations used in convolutional layers are ReLU and in the output (dense) layer is the softmax. All layer parameters are described in the figure. Neither layer has explicit regularization (like weight regularization in convolutional and dense layer). Explicit regularization showed performance decrease on validation set. Reasons for good performance without explicit regularization are probably the effect of implicit regularization caused by large enough dataset used for training, slim model itself (model with not too large capacity) and early stopping.

Model has 10286 parameters for dataset with *neighbourhood_size* $= 20$.

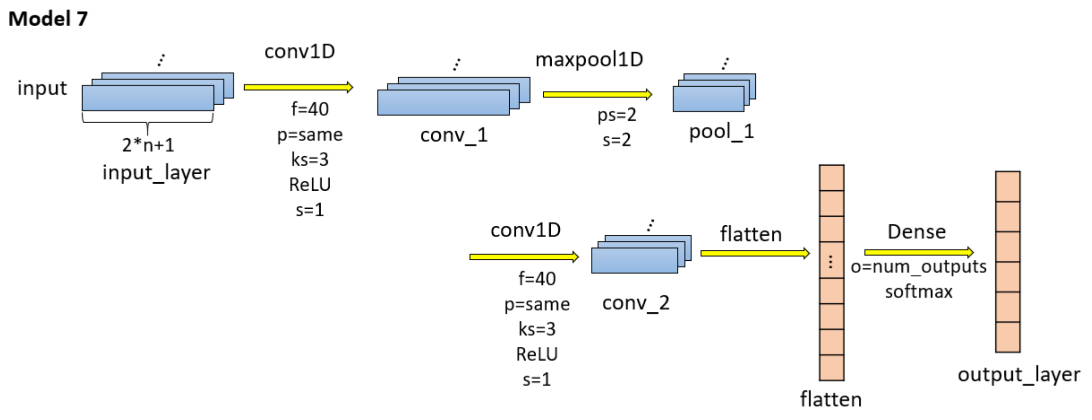


Figure 4.11: Model 7

4.3.2. Model 11

Model 11 is shown in figure 4.12. This slim model differs from model 7 only in one layer: batch normalization (Ioffe i Szegedy (2015)) layer after the second convolutional layer. Activations used in convolutional layers are ReLU and in the output (dense) layer is the softmax. Again, neither layer has explicit regularization (like weight regularization in convolutional and dense layer) due to the same reasons as for model 7.

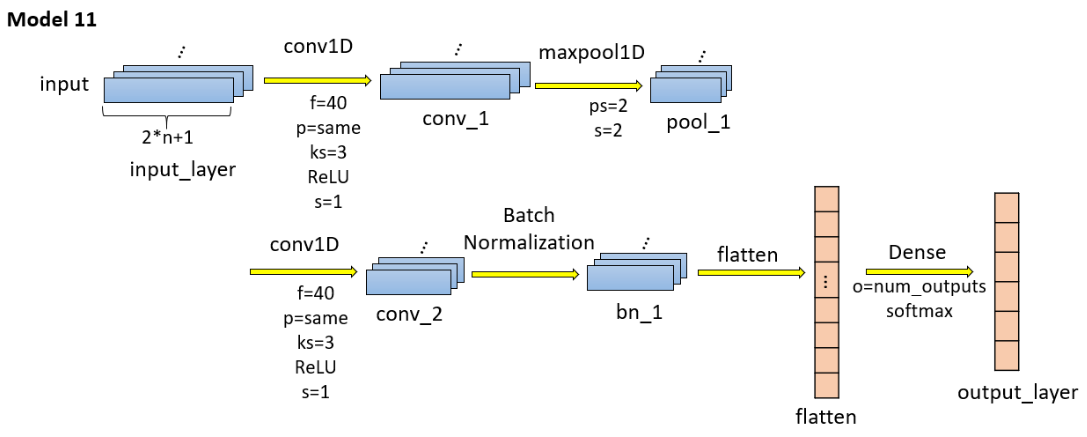


Figure 4.12: Model 11

Model has 10466 parameters for dataset with $neighbourhood\_size = 20$.

4.3.3. Model 23

Model 23 is shown in figure 4.13. This slim model has three 1D-convolutional layer, two maximum polling layer (after the first two convolutional layers), one batch normalization layer (after the third convolutional layer) and one dense layer. Activations used in convolutional layers are ReLU and in the output (dense) layer is the softmax. Neither layer has explicit regularization due to the same reasons as for previous two models.

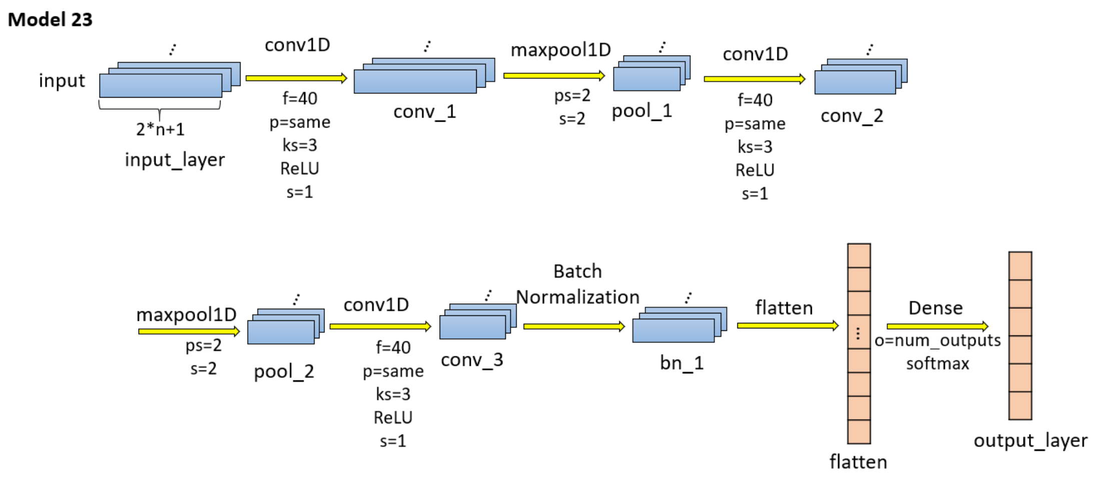


Figure 4.13: Model 23

Model has 12866 parameters for dataset with *neighbourhood_size* $= 20$.

4.3.4. Model 24

Model 24 is shown in figure 4.14. This model is not so slim as the previous three models. It consists of three 1D-convolutional layers, three maximum polling layers (one after each convolutional layer), two dense layers and one dropout. Activation used in all layers except the last dense layer is SeLU (Klambauer et al. (2017)). The last dense layer has softmax activation function. Convolutional layers in this model are regularized with L2 weight regularization with weight decay set to 10^{-3}.

Model has 527062 parameters for dataset with *neighbourhood_size* $= 20$.

4.3.5. Input and output shape of models

Datasets for training and data which is prepared for inference are shaped in a particular way to fit the Keras input shapes.

The first layer in all models in 1D-convolutional layer. It's input shape is *(batch, steps, channels)*. $batch$ is the number of samples in batch. $steps$ is the width of sample (in this case it is equal to *2*neighbourhood_size+1*). This number could be viewed as the number of time steps in time series sequence, hence the name $steps$ is used. $channels$ is the number of input channels in that 1D-convolutional layer.

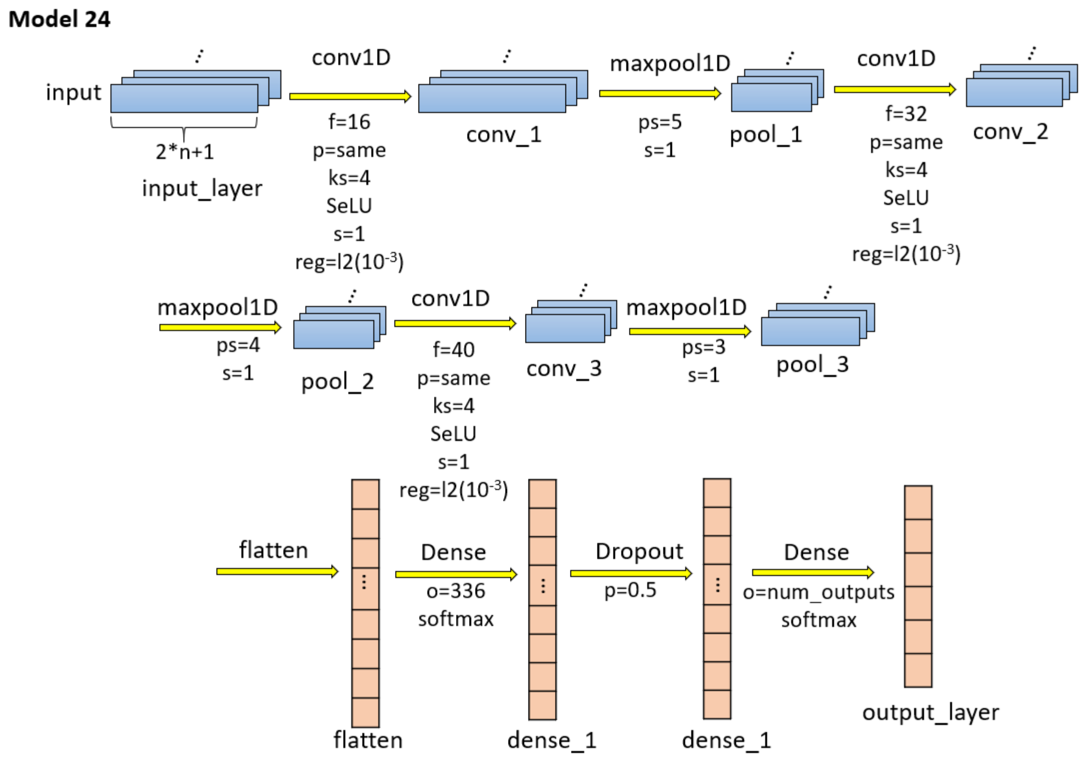


Figure 4.14: Model 24

Because of such input shape, samples are not in a more intuitive form as shown on figure 4.4 (**kod datasetworkflowa**), but are in a shape shown on figure 4.15. Every row which stores the number of specific base for consecutive positions is placed in it's own channel making the data one dimensional (with more channels) instead of two dimensional. Actually, such shape better fits the convolutional layer properties.

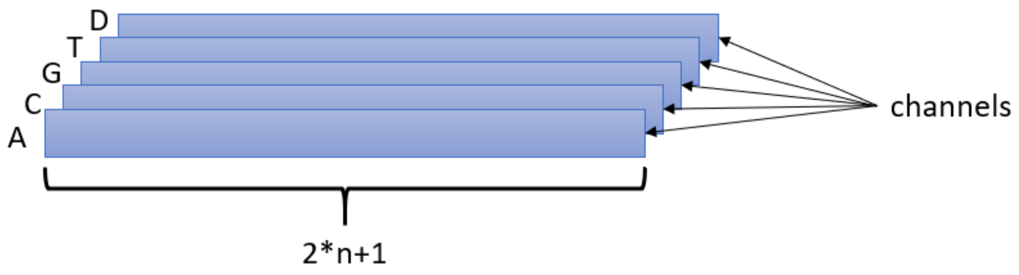


Figure 4.15: Input shape consists of multiple channels

If the data were two dimensional (all rows one below each other in two dimensional matrix), that would suggest some two dimensional topological order. But, rows in a sample are placed in arbitrary order - even though in the first row is the number of bases 'A', that could be any other base. Shaping a data in two dimensional shape would provide the information that e.g. ’A’s are above ’C’s, which are above ’G’s, and so on, which is obviously misleading. As it is shown here, two dimensional topological order does not exist here as in images, but only a one dimensional topological order (genomes and reads can be viewed as a sequence with one dimensional topological order). Hence, placing all the rows in their own channels is reasonable.

One dimensional topological order which is processed with 1D-convolution is the difference from other similar models (like Clairvoyante, DeepVariant, VariantNet, ...) which use 2D-convolution.

5. Results

Results represented here show that constructed models performed slightly worse and in the few cases slightly better then the consensus without neural network polishing. Various reasons for such results are described in the following sections. Nevertheless, since a lot of steps in the workflows are questionable (like dataset configuration, post-processing of predictions in the inference) a lot of room remains for progress and the model improvement.

5.1. Polishing results

All tables are divided in 3 parts. In the first row there is an average identity of the consensus before the polishing with neural network calculated with Mummer's tool ***dnadiff*** command. Other rows show the average identity of the consensus after the polishing with neural network calculated with the same command. These rows are grouped by models trained on the same *neighbourhood_size*. First group has *neighbourhood_size* $= 15$ and the other *neighbourhood_size* $= 20$. Also, last column shows the validation accuracy of the trained models. Validation accuracy is missing for some models due to the failure of CometML to store the training progress.

Different tables shows the results for different pileup configurations. Not all 4 described models are used in all datasets and pileup configurations. The reason is that some pileup configurations were shown to be inefficient: e.g. overlap with no indels definitely does not contain all important information because it does not include indels neither in samples neither in the labels. Also, some models were omitted during the experiments in later phases of research as newer models showed better performance.

Results are shown for 3 different datasets. One dataset contains data from PacificBiosciences sequencers, other two from Oxford Nanopore MioION sequencers. In some datasets the part of the data which was used during the training was also used in the inference: reads for some bacteria were used in dataset workflow to create the dataset for training, and later those same reads were used for polishing the consensus.

	Method	Avg. Identity					Vall acc
		E	K	M	S92	S129	
	Baseline	**99.57**	**99.15**	98.83	**98.44**	**98.35**	-
n=15	model 7	99.56	98.95	98.89	98.43	98.33	98.71
	model 11	99.56	98.95	**98.90**	98.43	98.33	98.65
n=20	model 7	99.56	98.93	98.89	98.42	98.32	98.68
	model 11	99.56	99.01	**98.90**	98.43	98.34	98.70

Table 5.1: Average identity and validation loss of models on dataset created from *Overlap with no indels* version of pileups for *All bacteria dataset*. Abbreviations meanings are given in the text.

This is obviously not an unbiased measure, but the reason why this was done is to provide some type of sanity check. If the consensus polishing with neural network cannot improve the consensus whose reads were part of the training process, then it is hard to expect that it will generalize well on samples from reads which were not in the dataset.

5.1.1. All bacteria dataset - PacificBiosciences data

This dataset consists of reads created with PacificBiosciences sequencers from 3 bacteria: E. coli(NCTC86), M. morgani(NCTC235), S. enterica(NCTC92 and NCTC129). In total, there are 4 groups of reads. In the inference, all 4 groups are used which were in the training set and also an additional group of reads - K. penumoniae (NTCT204) - which was not in the dataset. The name of the dataset, *All bacteria dataset*, only suggest that it was a mix of reads from multiple bacteria.

Table 5.1 shows the results with pileups created by *PysamstatsNoIndelGenerator*. Table 5.2 shows the results with pileups create by *PysamstatsIndelGenerator*. Table 5.3 shows the result with pileups created by *RaconMSAGenerator*. The abbreviations in the tables are the following: E. coli NTCT86 (**E**), K. pneumoniae NCTC 204(**K**), M. morgani NCTC235 (**M**), S. enterica NCTC92 (**S92**), S. enterica NCTC129 (**S192**).

5.1.2. S. cerevisiae dataset - Oxford Nanopore MinION data

This dataset consists of reads created with Oxford Nanopore sequencers. Dataset has reads from two reads groups, both groups for S. cerevisiae: one is from R7 and the other from R9 (Nanopore (2016)) sequencing chemistry. In the inference, along with S. cerevisiae read groups, E. coli reads were also used.

	Method	Avg. Identity					Vall acc
		E	K	M	S92	S129	
	Baseline	**99.57**	**99.15**	98.83	**98.44**	**98.35**	-
n=15	model 7	99.56	98.96	**98.89**	98.43	98.32	99.75
	model 11	99.56	98.97	**98.89**	98.42	98.31	99.76
n=20	model 7	99.56	99.05	**98.89**	98.42	98.32	99.74
	model 11	99.56	98.91	**98.89**	98.42	98.31	99.78

Table 5.2: Average identity and validation loss of models on dataset created from *Overlap with indels* version of pileups for *All bacteria dataset*. Abbreviations meanings are given in the text.

	Method	Avg. Identity					Vall acc
		E	K	M	S92	S129	
	Baseline	99.57	**99.15**	98.83	**98.44**	98.35	-
n=15	model 7	99.57	98.77	**98.93**	98.41	98.36	99.04
	model 11	99.57	98.89	**98.93**	98.41	**98.37**	99.03
	model 23	99.57	98.91	98.92	98.41	98.36	99.01
	model 24	99.54	97.92	98.84	98.36	98.24	98.88
n=20	model 7	99.57	99.01	98.92	98.41	**98.37**	99.05
	model 11	**99.58**	98.98	98.92	98.40	**98.37**	99.03
	model 23	-	-	-	-	-	-
	model 24	99.54	97.86	98.84	98.36	98.25	98.87

Table 5.3: Average identity and validation loss of models on dataset created from *MSA supported overlap with indels* version of pileups for *All bacteria dataset*. Abbreviations meanings are given in the text.

	Method	Avg. Identity			Vall acc
		E. coli	S. cerevisiae R7	S. cerevisiae R9	
	Baseline	**99.32**	**98.74**	97.40	-
n=15	model 7	99.32	98.74	**97.42**	99.99
	model 11	99.32	98.74	**97.42**	99.99
n=20	model 7	99.32	98.74	97.41	-
	model 11	99.32	98.74	**97.42**	99.99

Table 5.4: Average identity and validation loss of models on dataset created from *Overlap with no indels* version of pileups for *S. cerevisiae dataset*

	Method	Avg. Identity			Vall acc
		E. coli	S. cerevisiae R7	S. cerevisiae R9	
	Baseline	**99.32**	**98.74**	**97.40**	-
n=15	model 7	99.30	98.62	97.32	99.93
	model 11	99.31	98.63	97.32	99.95
n=20	model 7	99.31	98.63	97.32	-
	model 11	99.31	98.64	97.33	-

Table 5.5: Average identity and validation loss of models on dataset created from *Overlap with indels* version of pileups for *S. cerevisiae dataset*

Table 5.4 shows the results with pileups created by *PysamstatsNoIndelGenerator*. Table 5.5 shows the results with pileups create by *PysamstatsIndelGenerator*. Table 5.6 shows the result with pileups created by *RaconMSAGenerator*.

5.1.3. Fusobacterium dataset - Oxford Nanopore MinION data

This dataset consists of reads created with Oxford Nanopore sequencer from different strains of fusobacterium (Todd et al. (2018)). Training dataset has reads from four strains: gonidiaformans (**G**), mortiferum (**M**), necrophorum (**Ne**) and nucleatum-25586 (**Nu25**). Inference was also done on additional strain which were not in the training set: nucleatum-23726 (**Nu23**), periodonticum (**P**), ulcerans (**U**) and varium (**V**). All reads are sequenced using R9 chemistry.

Table 5.7 shows the result with pileups created by *RaconMSAGenerator*. Abbreviations in the table are described above.

Results show that no significant improvement hasn't yet been made: sometimes

	Method	Avg. Identity			Vall acc
		E. coli	S. cerevisiae R7	S. cerevisiae R9	
	Baseline	**99.32**	**98.74**	97.40	-
n=15	model 7	99.21	98.53	**98.97**	99.48
	model 11	99.17	98.47	98.93	99.43
	model 23	99.24	98.53	98.94	99.48
	model 24	98.97	98.43	98.82	99.26
n=20	model 7	99.05	98.34	98.79	99.47
	model 11	99.17	98.46	98.91	99.47
	model 23	99.20	98.61	98.99	99.41
	model 24	99.00	98.42	98.82	99.29

Table 5.6: Average identity and validation loss of models on dataset created from *MSA supported overlap with indels* version of pileups for *S. cerevisiae dataset*

	Method	Avg. Identity								Vall acc
		G	M	Ne	Nu25	Nu23	P	U	V	
	Baseline	**99.38**	**99.27**	**99.35**	**99.34**	**94.78**	**99.33**	**98.28**	**99.20**	-
n=15	model 11	98.89	98.84	98.81	98.89	91.44	98.87	97.78	98.84	99.70
	model 23	98.98	98.93	98.90	98.97	91.71	98.94	98.00	98.91	99.69
	model 24	98.97	98.89	98.87	98.96	92.46	98.95	97.17	98.75	99.53
n=20	model 11	98.96	98.91	98.87	98.95	91.73	98.91	98.24	98.90	99.69
	model 23	98.82	98.77	98.73	98.81	90.91	98.78	98.20	98.81	99.70
	model 24	98.82	98.73	98.75	98.76	92.55	98.74	97.55	98.66	99.54

Table 5.7: Average identity and validation loss of models on dataset created from *MSA supported overlap with indels* version of pileups for *Fusobacterium dataset.* Abbreviations meanings are given in the text.

Dataset\pileup version	Overlap with no indels	Overlap with indels	MSA supported overlap with indels
All bacteria dataset	15M	15M	28M
S. cerevisia dataset	22M	22M	35M
Fusobacterium dataset	-	-	11M

Table 5.8: Number of samples in millions in the training split for each dataset and pileup version

models increase the average identity of polished consensus, sometimes they decrease it.

5.2. Datasets visualization

Investigating the dataset properties can reveal some of the reasons for model failing to improve the average identity of polished consensus.

Following figures 5.1, 5.2, 5.3, 5.4, 5.5, 5.6 and 5.7 show dataset properties for various datasets and pileups versions. First plot in the figure represents the total number of samples for each class. Second plot shows the ratio between the number with most samples and the number of samples of that current class. Class with the most samples will have the ratio of 1, and the class with the number of samples equal to the half of the largest number of samples will have the ration of 0.5.

All plots are made for dataset with *neighbourhood_size* $= 15$, but distributions are the same for *neighbourhood_size* $= 20$ and the absolute numbers do not differ much, if at all. Table 5.8 shows the number of samples in millions in the training split for each dataset and pileup version. The large dataset sizes (between 10 and 35 millions) causes implicit regularization effect during training and guarantee that models won't just remember all the samples. The validation splits of those dataset had size of 10% of the corresponding training dataset.

Read type	Error rate (Proportion of overall error) (%)			
	Overall	Insertion	Deletion	Mismatch
PacBio	1.72	0.087 (5.06)	0.34 (19.48)	1.30 (75.46)
ONT	13.40	3.12 (32.30)	4.79 (35.70)	5.50 (40.99)

Table 5.9: Comaprison of error rates between *PacificBiosciences* and *Oxford Nanopore* reads

It looks like the key problem is the imbalance in the number of classes. In a *MSA supported overlap with indels* pileup version, insertion class has the most examples, nucleus bases have approximately the same number of samples, while deletion class has the smallest number of samples. Such distribution of samples per classes affects the model ability to learn the underrepresented classes: models make the most mistakes with deletions (or insertions) as it will be shown in the next section. In a *Overlap with indels* pileup version, situation is the opposite for the insertions and deletions numbers.

There are significant differences in the error rates in *PacificBiosciences* and *Oxford Nanopore* reads. Comparison between error rates is given in (Weirather et al. (2017)) and is shown in table 5.9. It can be seen that *PacificBiosciences* reads have much smaller error rate and that the most errors come from mismatches, while in *Oxford Nanopore* reads error rate is much larger and different error types share similar proportion of overall error. Also, it is important to be aware of the average read length: for *PacificBiosciences* reads it is $10 - 15kb$, while for *Oxford Nanopore* length can reach up to $900kb$ as shown in (Birla (2017)).

5.3. Visualization of samples

Figure 5.8 shows the incorrect predictions for model 11 on *Fusobacterium dataset* with *MSA supported overlap with indels* pileup version on a validation split. Plot shows that the most incorrect predictions were for insertions. Those predictions should have been some other class, but the model put them to insertion class. The reason for this is that the insertion class has the most samples in the training set as shown in the previous section. The next largest erroneous predictions were for deletions. Those predictions should have been some other class, but the model put them to deletion class. The reason for this is that the deletion class is overly underrepresented with samples as shown in the previous section.

Figure 5.9 shows the confusion matrix on validation split of mentioned dataset for model 11. Here can be seen that the most deletions were predicted to be insertions. Reason is simple: model gave them the label of most represented class. Figure 5.10 shows a random sample together with model's posterior probabilities for that sample where sample was wrongly labeled as not a deletion. Label t on x axis represents the *middle* position for which the prediction is being made, while flanking positions represent the neighbourhood with *neighbourhood_size* $= 15$.

Same visualizations for other models and datasets are omitted here because they are the same for this pileup version on all dataset for all models. Other pileup versions are not showed because models generally perform better with this pileup version.

5.4. Ablation study

5.4.1. Class weights

To confront the problem of class imbalance, one approach is to use the class weights during the training. Underrepresented classes can be given larger class weights, and overrepresented classes can be given smaller weights. Those class weights are then incorporated in the training process when calculating the loss. Loss for the current sample is multiplied with appropriate class weight and then backpropagated in the network to update the network weights.

This approach introduces new hyperparameters - class weights. The simplest way to calculate the class weights could be to set them to the inverse of class frequency in the dataset. Such class weights could be calculated with sklearn's ***compute_class_weights()*** method (learn developers (2007)).

Table 5.10 shows the validation accuracy of models with and without class weights. It turns out that inverse frequency for class weights does not improve the model performance as all validations scores with such class weight are $1-2\%$ lower. The inverse frequency puts too much weight on the underrepresented class - deletions.

5.4.2. Finetuning on smaller dataset

One other approach to the problem of class imbalance is to create a synthetic dataset with different samples distribution. For that purposes, couple of synthetic datasets were created from *Fusobacterium dataset* with *MSA supported overlap with indels* pileup version and *neighbourhood_size* $= 20$. All those synthetic dataset had larger number of deletions ($50k$ deletions) and other classes had the same or slightly larger number

Val acc	Without class weights	With class weights
model 11	**99.69**	98.86
model 23	**99.69**	98.51
model 24	**99.49**	97.78

Table 5.10: Comparison of validation accuracy of models on *Fusobacterium dataset* with *MSA supported overlap with indels* pileup version and *neighbourhood_size* $= 15$ with and without class weights calculated using inverse frequency

of samples.

All those synthetic datasets showed degradation of performance of models even though models were first trained on original datasets and then finetuned on synthetic datasets. Neither achieved the training accuracy over 50%. Freezing the weights in first layers of the models did not help (it is generally accepted that first layers handle lower abstraction features and that in finetuning it is enough to train just the couple of last layers).

5.4.3. Different pileups

Different pileup versions used in this project are described in Methods section. *Overlap with no indels* and *Overlap with indels* pileup versions showed that models trained with them on average showed slightly lower performance on consensus polishing then when trained with *MSA supported overlap with indels*. That is reasonable since distribution of samples on those two models almost does not include the insertions as shown in dataset visualizations.

5.4.4. Manual model arhitecture search

As stated before, 30 different model architectures were tested manually. Those experiments included models with different number of filters in convolutional layers, kernel sizes, batch normalization after various layers, maximum pollings to increase the receptive fields of layers and so on. Models 7, 11, 23 and 24 showed the best performance on average. List of all experiments (over 300) can be found on CometML page of the project - *https://www.comet.ml/ajuric/consensusnet*.

5.4.5. Weight decay

In Models section, three described models had no weight decay applied to their weights during the training. Weight decay is one of the explicit regularization techniques. When weight decay was applied, model showed the decrease in the performance - validation accuracy dropped about $2 - 3\%$, so models were underfitted due to excessive usage of regularization (large enough dataset, slim model with lower capacity and early stopping are all forms of the implicit regularizations and are already used during the training).

5.5. Hyperparameter tuning

To find the better values of hyperparameters of the network (e.g. number of filters and kernel size in the convolutional layers), Bayesian Hyperparameter optimization was used.

Bayesian Hyperparameter optimization (Koehrsen (2018)) is the technique which estimates the score function in the hyperparameter space: $P(score|hyperaparameters)$. Optimization process picks the set of hyperparameters for which it believes are going in the direction of increasing the score function. After the model has finished training and is evaluated, optimization adjust it's estimation of the score function and repeats the process. The process finishes after the optimization process concludes that its estimation matches the real score function in the hyperparameter space or the maximum number of steps are reached. Because in each iteration the model goes through full training procedure, the process might take long to converge, especially if model is large.

CometML supports the Bayesian Hyperparameter optimization and was used to optimize the number of filters in convoultional layers, kernel size in convolutional layers, maximum pooling size, weight decay and batch size. Over 300 experiments were executed during under the Bayesian Hyperparameter optimization and all can be found at *https://www.comet.ml/ajuric/consensusnet* - the project page. Optimizations showed a small (0.01%) or no improvement in model performance in contrast to default model hyperparameters which are described in Model section. Such findings suggest that in the whole process the models are not the weak part as they are trained to their maximum extent, but that the other parts should be optimized: class imbalance in the datasets, other pileups versions, and so on.

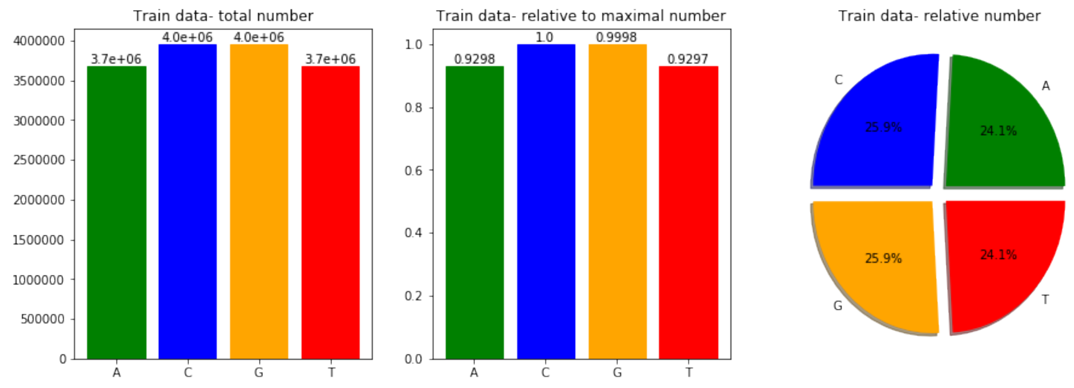


Figure 5.1: Visualization of *All bacteria* dataset with *neighbourhood_size* = 15 and *Overlap with no indels* pileup version

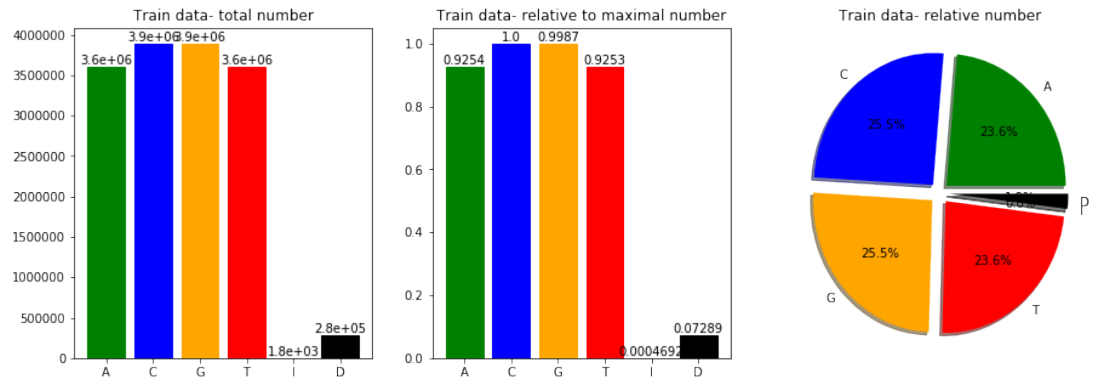


Figure 5.2: Visualization of *All bacteria* dataset with *neighbourhood_size* = 15 and *Overlap with indels* pileup version

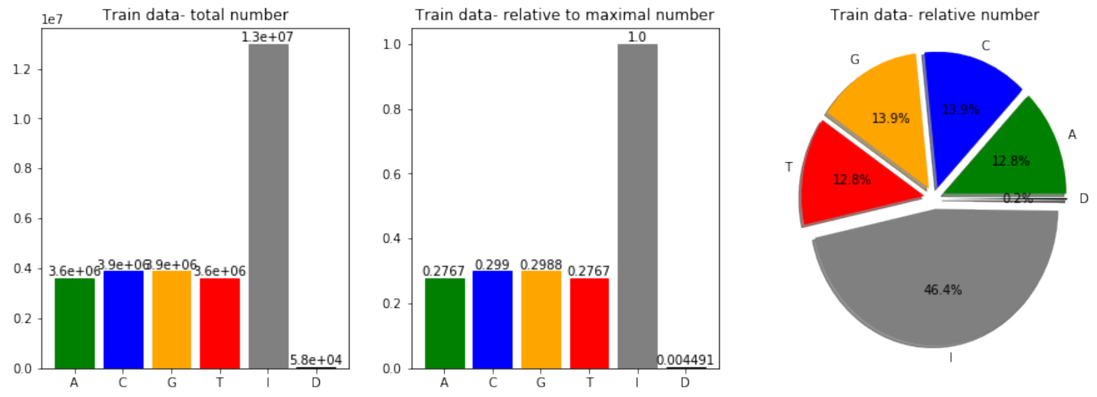


Figure 5.3: Visualization of *All bacteria* dataset with *neighbourhood_size* = 15 and *MSA supported overlap with indels* pileup version

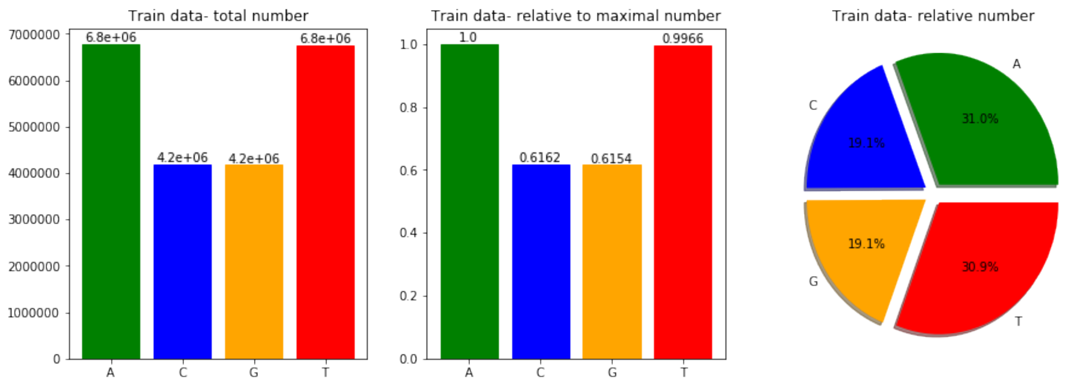


Figure 5.4: Visualization of *S. cerevisiae* dataset with *neighbourhood_size* = 15 and *Overlap with no indels* pileup version

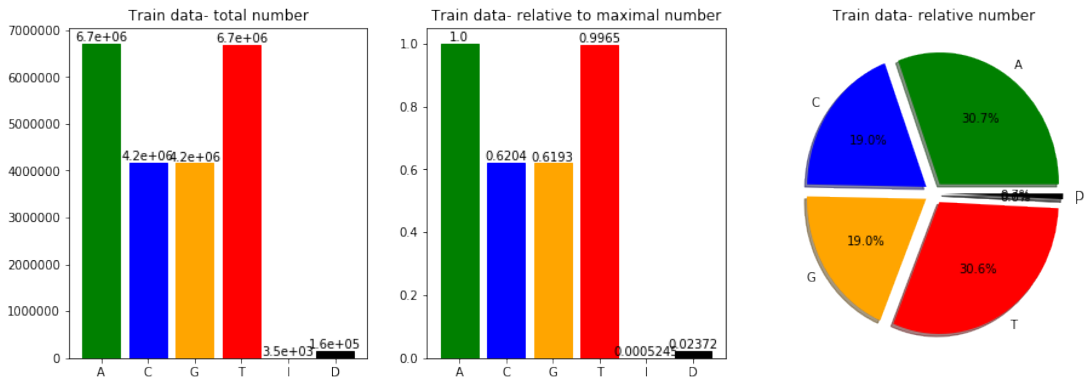


Figure 5.5: Visualization of *S. cerevisiae* dataset with *neighbourhood_size* = 15 and *Overlap with indels* pileup version

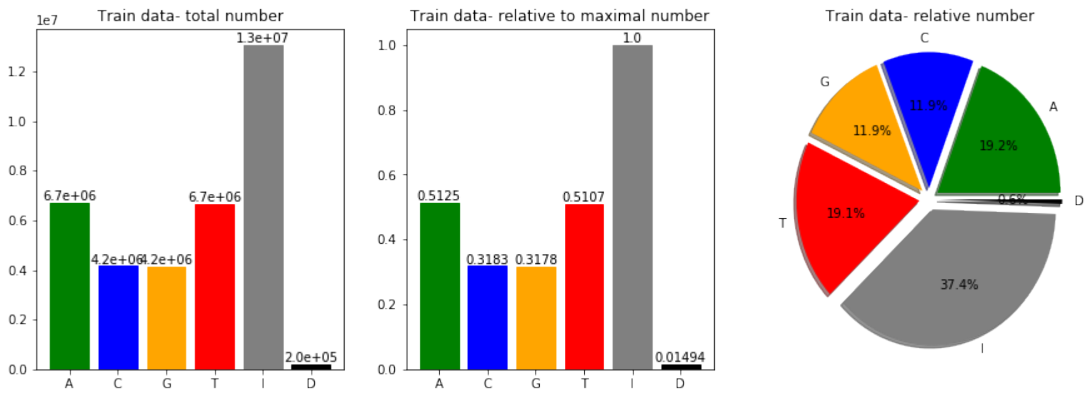


Figure 5.6: Visualization of *S. cerevisiae* dataset with *neighbourhood_size* = 15 and *MSA supported overlap with indels* pileup version

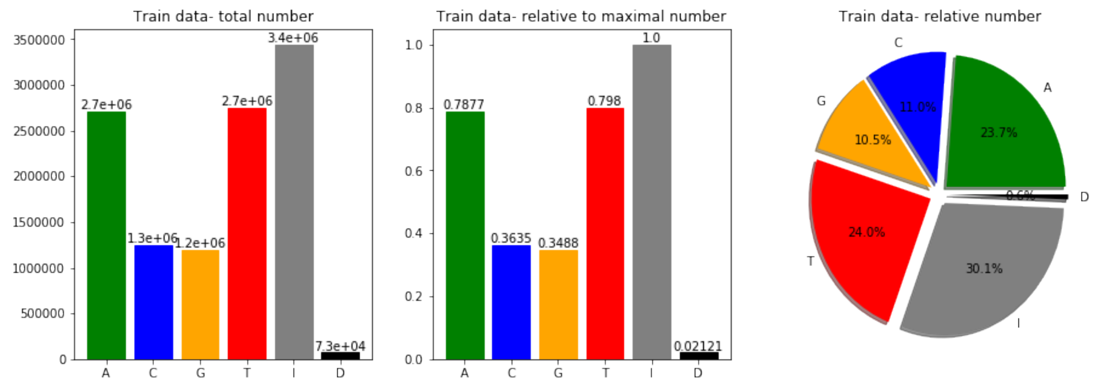


Figure 5.7: Visualization of *Fusobacterium* dataset with *neighbourhood_size* $= 15$ and *MSA supported overlap with indels* pileup version

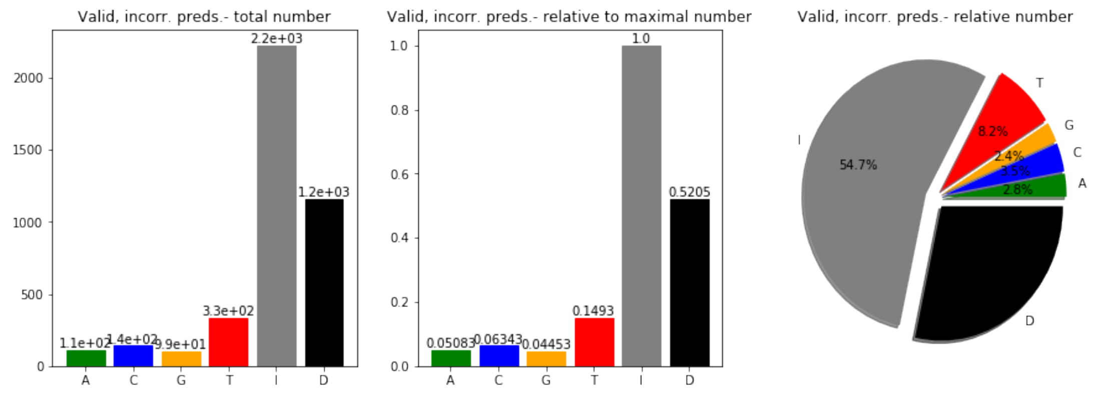
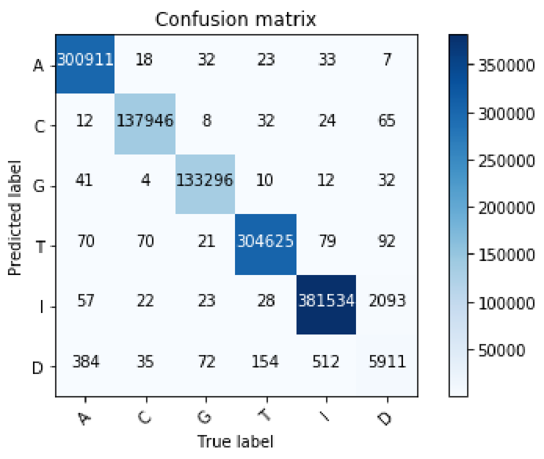


Figure 5.8: Visualization of incorrect predictions for model 11 on *Fusobacterium dataset* with *MSA supported overlap with indels* pileup version on a validation split



	A	C	G	T	I	D
A	300911	18	32	23	33	7
C	12	137946	8	32	24	65
G	41	4	133296	10	12	32
T	70	70	21	304625	79	92
I	57	22	23	28	381534	2093
D	384	35	72	154	512	5911

Figure 5.9: Confusion matrix for model 11 on *Fusobacterium dataset* with *MSA supported overlap with indels* pileup version on a validation split

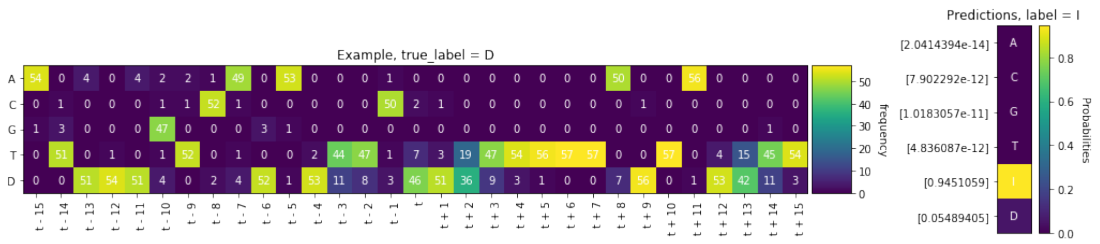


Figure 5.10: Visualization of random sample which was wrongly labeled as not a deletion for model 11 on *Fusobacterium dataset* with *MSA supported overlap with indels* pileup version on a validation split

6. Future work

The whole process is time consuming and there are a lot of places to improve. Hence, a couple few ideas were left untested and are guidences for future work.

6.1. Datasets

Creating a dataset in the same form as is described in other similar works is definitely worth of trying. Clairvoyante and VariantNet models are both convolutional models and they construct the dataset in the way superior to DeepVariant as the authors describe it. Since those models solve other problem (variant calling), dataset should be adopted for consensus polishing problem, but the main idea stays the same.

Those dataset provide more contextual information than just the neighbourhood of the current positino for which prediction is being made - they include the matrices which also stores the differences between the reads and references. Such addition of information could help to improve the polishing consensus process.

6.2. Class weights and sample weights

This project already tried the simplest form of class weights - the inverse frequency. Other class weights could be also tried as the inverse frequency was found to put too much weight on the underrepresented class - deletions. Possible solutions could be putting smaller weight to underrepresented class than the inverse frequency approach would do.

Other possible direction is to put the weights on the samples. This way the sample from the same class can get the different weights which could be especially helpful if particular samples from some class are hard to learn.

The idea of sample weights can be even more extended. In a situation where data is sequential (which is the case with samples with neighbourhood), weights could be defined for each time step of sequential data. Such modeling enables placing the information on the center of the sample. With the current learning configuration, models by themselves must figure out that the most information for predicting the right base for some sample is stored in the center of the sample. By using the weights for steps, models are suggested to put more focus on the center of the sample.

All described techniques are supported in Keras' ***fit()*** method (Charles (2013)).

6.3. Edit distance measure

Besides the average identity measure used in this project to check if polished consensus have improved, other good measure could be an edit distance between the reference and the polished consensus. Edlib (Sosic i Sikic (2016)) tool could be used to calculate the edit distance.

That tools was used in this project to try to calculate the edit distance, but it showed problems when it was used on data consisting of multiple contigs. Also, if data is circular (which is common for bacteria), data must be preprocessed in a way to make both data start at position in which are they aligned. To calculate the coordinates from which the alignment starts, Mummer tool could be used.

6.4. Inference - output size

Currently the size of the output is 1, since for each sample models predict one base. The question remains what would happen if the model didn't predict only one base, but multiple bases (e.g. 2 or 3 consecutive bases). In such scenario, the sample would have multiple centered positions along with *neighbourhood_size* flanking positions.

When pileups are being created, sliding window would have the size of *2*neighbourhood_size+x* where x is the number of centered positions (and, also the size of the output). If sliding window was then moved for x positions, the predictions wouldn't overlap and they would be just appended at the polished consensus. If the sliding windows was moved for less than x positions, subsequent predictions would overlap in some number of positions and those overlapping would have to be resolved in the postprocessing (e.g. the simplest overlap postprocessing could be the majority vote).

6.5. Inference postprocessing

Current inference postprocessing works in a way that for each prediction (which are posterior probabilities) appropriate base is appended onto polished consensus which is an empty at the beginning. If prediction is a nucleus base, then corresponding base is appended ('A', 'C', 'G' or 'T'). If the prediction is insertion, then nothing is appended since it means that reads have something which was not there. If the prediction is a deletion, then prediction is a symbol 'N' representing that nucleus base is missing from reads.

Putting the symbol 'N' could confuse the metrics and potentially lower the score. Edlib tool has support to provide which bases are the same when calculating the edit distance, hence one could provide that all other bases are the same as 'N'. Other try could be, if the prediction is a deletion, too also append nothing, but it's not clear how this will affect the metrics since in this situation the final length of polished consensus is being shortened when it should not be.

6.6. Multiple output branches or specialized models

In order to better handle the underrepresented class (deletions), specialized branch of the network, of whole new network, could be used to predict the deletion class. If this branch, or network, predicts the that input is not a deletion, normal output is considered to resolve whether it is a base or an insertion. Such approach is used in Clairvoyante XY (put reference) to separate the outputs which predict the bases and the outputs which predict other useful information (zygosity, variant type and so on).

6.7. Homopolymers

At the end of the work, in order to find the clues which part of the workflow could be improved, *MSA supported overlap with indels* pileup version was investigated (MSA was done by Racon tool). Especially, deletion class was investigated. When neural network polishes the consensus, if it predicts a deletion, it will output symbol 'N' meaning that some base is missing here. If the model knew what was missing and instead of symbol 'N' appended the true base to the polished consensus, that will definitely rise the final polished consensus quality.

It turns out that there are some regularities when the deletion happens. Looking at the PacBio pileups, it can be seen that most of the time there are two cases. In one case

Datasets	total deletions	homopolymers before deletions	homopolymers after deletions	deletion bursts with length 2	deletion bursts with length 3	deletion bursts with length 4	deletion bursts with length >4
All becteria	77787	15901	13865	4983	2197	1080	2212
S. cerevisiae	216684	47766	88398	24559	4607	631	152
Fusobacterium	301368	71621	164367	18112	5136	1943	2589

Table 6.1: Total number of deletions, homopolymers after and before deletions and length of deletion bursts per dataset

a deletion is surrounded by sequence of the more deletions (burst of deletions). These are the parts of the subsequent pileup labels for this case: ...*GDDDDIGITDDGDDD DDDCDDDDDDCDDDCDDCDDDDDDTG*..., ...*ITITIITDDDCDDDDD DDDDDDDDDDDDDCTDTIDDDDCAD*..., ... In the other case, a deletion is intertwined with other bases and insertions. These are the parts of the subsequent pileup labels for this case: ...*AAAITGIGAIAICICIITIIDGAIIGICCIIIGCIG ITIGI*..., ...*CCTIGICIITGGTIIGIAIIDIIDCIAIGIGITIIAIIATT*..., ...

Looking at the Oxford Nanopore pileups, it can be seen that the mentioned two cases also occur but slightly less frequent, while there is also a third case. In the Oxford Nanopore pileups, deletions often come at the beginning of the homopolymer regions - sequences of the same bases one after the another. These are the parts of the subsequent pileup labels for this case: ...*IIIIIIAAIIIIIGAAAATDAAAAACAGATGIIIA ATAGI*..., ...*ICIICTDCCCITIITIGGTIDGGGGGCACTTITGGAIGIGA*..., ...*IIIIITTTDIITITTTDDDDTTTTTTTTTGACGACITICT*....

All those statistics are summarized in the table 6.1. It can be seen that in datasets created from Oxford Nanopore reads deletions surrounding were homopolymers. Hence, it would be reasonable to try ,in the inference postprocessing, to put the homopolymer base in the place of symbol 'N' when working with Oxford Nanopore reads to improve the polished consensus quality. Other aproach taking this findins into the account would be to make an additional prediction branch (like described in the previous section) which would predict the length of deletions. By knowing the deletion length, model could try to replace such region with appropriate homopolymer base.

6.8. Possible incompatibility of optimization processes

This last section does not need to be considered as an idea to try - rather an observation. While training a deep neural network, model is forced to optimize the accuracy on the samples which are only a tiny part of the polished consensus. While one can hope that with optimizing this problem it would also increase the performance on the other problem (creating a better polished consensus), there is currently no a strong proof for such behaviour.

To make these two optimization processes match, instead of using an cross-entropy error for each sample in the dataset, more accurate loss information would be the average identity (or edit distance) of polished consensus created with the model which contains the same weights as in the current step of the training process. But, calculating the whole polished consensus and it's metric before every parameter update is time consuming. Also, it is questionable if such information would be derivable (maybe the edit distance could be the derivative?) which might make it unusable in backpropagation and it might overfit to that particular reference.

Hence, keeping the two optimization processes separate in the theory is not the same, but in the practice it is shown that by optimizing a model performance on the samples, both optimization processes are being optimized.

7. Conclusion

This work shows that polishing a consensus with neural network in order to increase the metric score (e.g. average identity) is possible, but the whole process still needs the improvements.

Process is highly fragile as it consists of few workflows with a lot of steps. Those fragile parts are pileups creation, dataset construction, model training, inference, inference post processing and so on. Among three represented pileup versions, one showed having on average a better contextual information than the others, but there is no reason that there is no another pileup version which wouldn't do better. There is no standard dataset on which models could be trained and compared as in other fields where deep learning techniques are being applied (e.g. image processing). The existence of such standard dataset would definitely improve the applications of deep learning techniques in this field.

Forming a samples from a dataset and then using those samples for training was shown as a rather simple problem even for slim neural networks since a lot of models achieved a validation accuracy over 99%. The main problem which occurs for implemented pileup creation and dataset is the class imbalance. The number of deletions is well underrepresented, hence making it hard for models to properly learn to predict deletion samples.

Nevertheless, despite all those problems, this work hopefully provides a way to a further improvement of the method. Sometimes resulting in a slightly better or slightly worse results and most of the times giving the same result, confirms that it might be on a good track.

BIBLIOGRAPHY

Bashari Rad Babak, John Bhatti Harrison, i Ahmadi Mohammad. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security*, 2017. URL `http://search.ijcsns.org/07_book/html/201703/201703027.html`.

Bhagyashree Birla. Pacbio vs. oxford nanopore sequencing, 2017. URL `{https://blog.genohub.com/2017/06/16/pacbio-vs-oxford-nanopore-sequencing/}`.

P.W.D. Charles. Project title. `https://github.com/charlespwd/project-title`, 2013.

Jason Chin. Simple convolutional neural network for genomic variant calling with tensorflow, 2017. URL `https://towardsdatascience.com/simple-convolution-neural-network-for-genomic-variant-calling-with-`

CometML developers. Cometml, 2018. URL `https://www.comet.ml/`.

Pysam developers. Pysam, 2009. URL `https://pysam.readthedocs.io/en/latest/api.html`.

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

Sergey Ioffe i Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL `http://arxiv.org/abs/1502.03167`.

Diederik P. Kingma i Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL `http://arxiv.org/abs/1412.6980`.

Günter Klambauer, Thomas Unterthiner, Andreas Mayr, i Sepp Hochreiter. Self-normalizing neural networks. *CoRR*, abs/1706.02515, 2017. URL `http://arxiv.org/abs/1706.02515`.

William Koehrsen. A conceptual explanation of bayesian hyperparameter optimization for machine learning, 2018. URL `https://towardsdatascience.com/a-conceptual-explanation-of-bayesian-model-based-hyperparameter-opt`

Stefan Kurtz, Adam Phillippy, Arthur L. Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, i Steven L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(2):R12, Jan 2004. ISSN 1474-760X. doi: 10.1186/gb-2004-5-2-r12. URL `https://doi.org/10.1186/gb-2004-5-2-r12`.

Scikit learn developers. Compute class weight, 2007. URL `http://scikit-learn.org/stable/modules/generated/sklearn.utils.class_weight.compute_class_weight.html`.

Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34 (18):3094–3100, 2018. doi: 10.1093/bioinformatics/bty191. URL `http://dx.doi.org/10.1093/bioinformatics/bty191`.

Nicholas James Loman, Joshua Quick, i Jared T Simpson. A complete bacterial genome assembled de novo using only nanopore sequencing data. *bioRxiv*, 2015. doi: 10.1101/015552. URL `https://www.biorxiv.org/content/early/2015/03/11/015552`.

Ruibang Luo, Fritz J Sedlazeck, Tak-Wah Lam, i Michael Schatz. Clairvoyante: a multi-task convolutional deep neural network for variant calling in single molecule sequencing. *bioRxiv*, 2018. doi: 10.1101/310458. URL `https://www.biorxiv.org/content/early/2018/04/28/310458`.

Alistair Miles. Pysamstats, 2012. URL `https://github.com/alimanfoo/pysamstats`.

Myers E.W. et al. A whole-genome assembly of drosophila. *Science*, (287):2196–2204, 2000.

Oxford Nanopore. Update: New ‘r9’ nanopore for faster, more accurate sequencing, and new ten minute preparation kit, 2016.

URL https://nanoporetech.com/about-us/news/update-new-r9-nanopore-faster-more-accurate-sequencing-and-new-ten-

Nanoporetech. Medaka, 2018. URL https://nanoporetech.github.io/medaka/.

NVIDIA. Nvidia-docker, 2018. URL https://github.com/NVIDIA/nvidia-docker.

PacificBiosciences. Genomicconsensus, 2018. URL https://github.com/PacificBiosciences/GenomicConsensus.

Ryan Poplin, Pi-Chuan Chang, David Alexander, Scott Schwartz, Thomas Colthurst, Alexander Ku, Dan Newburger, Jojo Dijamco, Nam Nguyen, Pegah T. Afshar, Sam S. Gross, Lizzie Dorfman, Cory Y. McLean, i Mark A. DePristo. Creating a universal snp and small indel variant caller with deep neural networks. *bioRxiv*, 2018. doi: 10.1101/092890. URL https://www.biorxiv.org/content/early/2018/03/20/092890.

Genome Research. Samtools, 2009. URL http://www.htslib.org/.

Martin Sosic i Mile Sikic. Edlib: A c/c++ library for fast, exact sequence alignment using edit distance. *bioRxiv*, 2016. doi: 10.1101/070649. URL https://www.biorxiv.org/content/early/2016/08/23/070649.

S. Michelle Todd, Robert E. Settlage, Kevin K. Lahmers, i Daniel J. Slade. Fusobacterium genomics using minion and illumina sequencing enables genome completion and correction. *mSphere*, 3(4), 2018. doi: 10.1128/mSphere.00269-18. URL https://msphere.asm.org/content/3/4/e00269-18.

Robert Vaser, Ivan Sovic, Niranjan Nagarajan, i Mile Sikic. Fast and accurate de novo genome assembly from long uncorrected reads. *bioRxiv*, 2016. doi: 10.1101/068122. URL https://www.biorxiv.org/content/early/2016/08/05/068122.

JL Weirather, M de Cesare, Y Wang, P Piazza, V Sebastiano, XJ Wang, D Buck, i KF Au. Comprehensive comparison of pacific biosciences and oxford nanopore technologies and their applications to transcriptome analysis [version 1; referees: 2 approved with reservations]. *F1000Research*, 6(100), 2017. doi: 10.12688/f1000research.10571.1.

Wikipedia. Binary alignment map, 2016a. URL https://en.wikipedia.org/wiki/Binary_Alignment_Map.

Wikipedia. Sam (file format), 2016b. URL https://en.wikipedia.org/wiki/SAM_(file_format).

Assembly Improvement Using Deep Learning Methods

Abstract

Development of tools for genome assembling with enough precision to be useful in practice is still an open problem. Third generation sequencers allow genome assembly with smaller fragmentation and high accuracy. In this work, we try to improve the final accuracy of the assembled genome using deep learning techniques. Trained convolutional deep network polishes errors in the assembled genome by learning the correct bases, insertions and deletions patterns. Networks are prepared to be used with Pacific Biosciences and Oxford Nanopore data. Current results show that sometimes models slightly improve, sometimes slightly degrade consensus quality, but there is still a room for progress since there are a lot of untested ideas which are described in the end of the thesis.

Keywords: bioinformatics, consensus, genom assembly, deep learning

Poboljšanje sastavljenih genoma metodama dubokog učenja

Sažetak

Razvijanje alata koji sastavljaju genom s dovoljnom točnošću da bi bili upotrebljivi u praksi još je otvoren problem. Uređaji treće generacije za sekvenciranje genoma omogućuju sastavljanje genoma s manjom stopom fragmentiranosti te visokom točnošću. Konačnu točnost stastavljenog genoma u ovom radu pokušavamo popraviti metodama dubokog učenja. Trenirana konvolucijska duboka mreža na temelju naučenih frekvencija pojedinih baza, umetanja i brisanja popravlja konačni genom ispravljajući pronađene pogreške. Mreže su pripremljene za Pacific Biosciences te Oxford Nanopore podatke. Trenutni rezultati pokazuju da modeli na nekim konsezusuzima neznatno poprave, dok na drugima neznatno snize druge točnost. No, modele je moguće još popraviti s nizom neisprobanih ideja opisanih na kraju rada.

Ključne riječi: bioinfromatika, konsenzus, sastavljanje genoma, duboko učenje