Non-reference-based DNA read compression using machine learning techniques

Thomas Mortier

Supervisors: Prof. dr. Wesley De Neve, Prof. dr. ir. Joni Dambre
Counsellors: Ruben Verhack, Tom Paridaens

Master's dissertation submitted in order to obtain the academic degree of Master of Science in Computer Science Engineering

Biotechnology Department of Environmental Technology, Food Technology and Molecular,Vakgroep Elektronica en Informatiesystemen
Chair: Prof. dr. Jozef Vercruysse

Department of Electronics and Information Systems
Chair: Prof. dr. ir. Rik Van de Walle

Faculty of Engineering and Architecture
Academic year 2015-2016
Non-reference-based DNA read compression using machine learning techniques

Thomas Mortier

Supervisors: Prof. dr. Wesley De Neve, Prof. dr. ir. Joni Dambre
Counsellors: Ruben Verhack, Tom Paridaens

Master's dissertation submitted in order to obtain the academic degree of Master of Science in Computer Science Engineering

Biotechnology Department of Environmental Technology, Food Technology and Molecular, Vakgroep Elektronica en Informatiesystemen
Chair: Prof. dr. Jozef Vercruysse

Department of Electronics and Information Systems
Chair: Prof. dr. ir. Rik Van de Walle

Faculty of Engineering and Architecture
Academic year 2015-2016
Acknowledgments

Working on a masters dissertation comes contemporaneously with moments of joy and despair. This book is the report of a long process, which cannot express the long days spent in the Data Science Lab, filled with hope for good results and interspersed with sadness after each failed attempt. However, it is after great and long-term persistence that I can finally finish this book together with years of education, which were honestly the best years of my life.

I would like to thank my supervisors prof. dr. ir. Joni Dambre and prof. dr. Wesley De Neve for offering me this dissertation. Again, I would like to thank prof. dr. ir. Joni Dambre and prof. dr. ir. Tom Dhaene for the wonderful course on machine learning, during the first semester. During this course, my interests in machine learning started to grow. I am fully convinced that in the near future, machine learning will change the world as we know it today. A special thanks also goes to Jeroen Burms for the tremendous support throughout the machine learning course project, which provided me with fundamental knowledge in convolutional neural networks. I would also like to thank Ira Korshunova for the support in recurrent neural networks and other crucial aspects in this work. More importantly, a big thanks also goes to my counsellors Lionel Pigou, Ruben Verhack and Tom Paridaens for the unconditional support and enthusiasm throughout my research.

Of course, this dissertation would not have been possible without the love and support of family and friends. I would like to thank my parents for their trust and faith in realizing my own potential and showing me the great value of dedication and education. Without their support, I may never have become the person I am today. Thanks to my close friends, for supporting me throughout all these years and filling my current life with unforgettable memories. A special thanks also goes to Kiani, Jan and Jannick for the numerous moments of late night working. And – last but not least – thank you Charlotte for the love, motivation and consolation that you provided me during this dissertation. I couldn’t possibly imagine this year without you by my side.

Thomas Mortier, May 2016
Permission for consultation

“The author gives permission to make this master’s dissertation available for consultation and to copy parts of this master’s dissertation for personal use. In the case of any other use, the copyright terms have to be respected, in particular with regard to the obligation to state expressly the source when quoting results from this master’s dissertation.”

Thomas Mortier, May 2016
Summary

Since the latest evolutions in high-throughput sequencing technologies, a dramatic decline has been seen in genome sequencing costs together with a tremendous increase of genomic data. In order to process and store large amounts of genomic data, a lot of capacity is needed when it comes to storage and transmission. Therefore a recent surge of interest has been emerged for genomic (high-performance) data compression tools in order to allow efficient, low-cost storage and management for genomic data. In the past it has already been proven that video and other state-of-the-art techniques work well when it comes to genomic sequence and read compression. The goal for this masters dissertation is to perform extensive analysis on whether it is feasible to make a transition to machine learning techniques and more specific deep learning techniques. Hereby, different machine learning models are going to be studied. Currently a lot of compression techniques (e.g. image/sound/text compression) have already been realized by the use of various deep learning models such as recurrent neural networks, autoencoders, convolutional neural networks etc. The question remains if compression, using the latter techniques, is also achievable on genomic data.

Keywords: machine learning, deep learning, genomic read compression
(non-reference-based) DNA read compression using machine learning techniques

Thomas Mortier

Supervisor(s): prof. dr. ir. R. van de Walle, prof. dr. ir. W. De Neve, prof. dr. ir. J. Dambre, Ruben Verhack, Tom Paridaens, Lionel Pigou

Abstract—As of today, a dramatic decline has been seen in genome sequencing costs together with tremendously increasing genomic data, due to the latest evolutions in high-throughput sequencing technologies. In order to process and store large amounts of genomic data, a lot of capacity is needed. Therefore a recent surge of interest has been emerged for genomic (high-performance) data compression tools in order to allow efficient, low-cost storage and management for genomic data. In the past it has already been proven that video techniques and other state-of-the-art compression tools work well when it comes to genomic sequence and read compression. We perform extensive analysis on whether DNA read compression is also possible using machine learning techniques. An abstract novel framework for DNA read compression is proposed, using convolutional autoencoders (CAEs), where compression ratios well below 0.9 bits per base (bpb) are observed. We show that CAEs generalize well on unseen datasets and are able to highly compress, where for some datasets better compression ratios are observed in contrast to 7-zip and video techniques.

Keywords—Machine learning, deep learning, genomic read compression, reference-based encoding, video techniques and machine learning techniques (to a limited extent).

I. INTRODUCTION

High-throughput sequencing technologies led to a dramatic decline in genome sequencing costs together with a tremendous increase of genomic data. These evolutions led to ambitious and huge projects, e.g. the 1000 Genomes Project [1]. This is one of the first projects to sequence the genome of a large number of people, to provide a comprehensive resource on human genetic variation. In order to process and store large amounts of genomic data, a lot of capacity is needed when it comes to storage and transmission. As cause of this, a recent surge of interest has been emerged for genomic data compression tools in order to allow efficient, low-cost storage and management of genomic data.

The purpose of this work is to study the transition of various state-of-the-art techniques towards machine learning techniques. We analyze different machine learning models such as predictive neural networks (PNNs), flat autoencoders (FAEs) and convolutional autoencoders (CAEs). At the same time, we discuss various shortcomings which come with RNNs, PNNs and FAEs, together with the effectiveness and efficiency of CAEs. These models owe their success due to biologically inspired convolutional layers prior to and after an autoencoder network. Finally, we propose an abstract novel framework for DNA read compression.

II. RELATED WORK

We emphasize a lack of proper benchmarks for DNA read compression, using machine learning techniques. However, we can compare machine learning models to the current state-of-the-art techniques, which are divided in 5 general classes: naive bit encoding, dictionary-based encoding, statistical encoding, reference-based encoding, video techniques and machine learning techniques (to a limited extent).

A. Naive bit encoding

In a naive bit encoding scheme, multiple symbols or nucleotides (in an alphabet $\mathcal{L}$) are stored using one byte. The latter can be seen as simple compression, in contrast to the human friendly ASCII format which comes with a lot of memory overhead. In practice, different symbols can be defined for storing genomic data, next to the basic set $\{A, C, G, T\}$. For instance, to denote uncertainty which comes with sequencing technologies, one can introduce an additional symbol $N$ [2].

B. Dictionary-based encoding

Many dictionary-based algorithms currently exist, like CASToRe, which is a modification of the Lempel-Ziv compression. POMA, a particle swarm optimization-based algorithm for DNA read compression, is also an important dictionary-based algorithm. Differentiation between four different kinds of repeat patterns (i.e. direct, mirror, pairing and inverted repeats) is used where the most commonly repeated fragments are identified and included in a dictionary. Compression ratios of 1.909-2.056 bpb are found, using CASToRe for different genomes, while a compression ratio of 1.3 bpb is found when using POMA [2].

C. Statistical encoding

Cormack G. and Horspool N. [3] propose a statistical encoding algorithm, based on hidden Markov models (HMM). DNA sequences are approximated by a hidden Markov model, on which statistical inference can be applied. The order of the model defines the number of symbols, which can be taken in account to predict the next symbols. It is important to mention that the compression rate, for statistical algorithms, mostly depends on the distribution of input symbols and the available memory for construction of frequency distributions. The compression rates of statistical algorithms are usually between 4 bpb and 8 bpb for DNA sequences. This can be explained by the fact that nucleotides are on average equally distributed within a genome, thus yielding worse compression rates [2].

D. Reference-based encoding

Daily K., Rigor P., Christley S., et al. [4], present GenCompress. In this algorithm, sequences are aligned to a reference sequence with reference entries being composed of: starting position, match length and an optional difference list describing
Table I

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description (coverage)</th>
<th>#reads</th>
<th>#bases</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA12878, S1.fasta</td>
<td>Homo sapiens (26x)</td>
<td>69,430,054</td>
<td>101</td>
</tr>
<tr>
<td>homosapiens.fasta</td>
<td>Homo sapiens (2.3x)</td>
<td>56,463,236</td>
<td>100</td>
</tr>
<tr>
<td>K562.fasta</td>
<td>RNA data (16x)</td>
<td>246,476,391</td>
<td>76</td>
</tr>
<tr>
<td>MiSeq_Ecoli_DH10B.fasta</td>
<td>E. coli bacteria (44x)</td>
<td>13,175,350</td>
<td>150</td>
</tr>
</tbody>
</table>

Four different datasets which were used for the training and testing of our deep learning models.

---

Fig. 1. Example of a CAE architecture where the first part (encoding) is shown. The blue layer at the end represents the bottleneck layer which produces the codeword. Two conv-pool layers are shown for deep feature extraction whereafter the features are used as input to the encoder network (similar as in the FAE networks). The decoding part has been omitted in order to aid readability.

---

mismatches. The focus in GenCompress is on entropy encoding of integers via fixed or variable schemes, such as Huffman, Golomb or Elias. The algorithm only supports compression of the four nucleotides. Similarly, Kozanitis C., Saunders C., Kruglyak S., et al. [5] propose SLIMGENE, a lossless or lossy reference-based compression scheme focusing on how to find encodings of integers in order to minimize storage. Compression ratios of 9.8:1 – 22.7:1 are found for GenCompress, using three heterogeneous datasets, and 39:1 for SLIMGENE on human short reads.

E. Video techniques

Paridaens T., De Neve W., Lambert P., et al. [6][7] discuss the integration and optimization of Context-Adaptive Binary Arithmetic Coding (CABAC), within a block-based framework for genomic (read) compression. The latter framework allows for random access functionality, stream processing, adaptive streaming and encryption next to DNA read and sequence compression. The obtained compression ratios for the latter framework can be seen in Table II.

F. Machine learning techniques

Manuel J., Duarte and Armando J. Pinho [8] verify that artificial neural networks are useful in combination with other models for the compression of genomic data. In this work, artificial neural networks (ANNs) are used to reduce the high memory requirements, which come with the use of finite-context models (FCMs). The problem is that a lot of runtime memory is required to store high-order finite-context models. The authors are able to reduce these requirements by using ANNs to build probabilistic models in a compact way. Testing and training is mainly done on bacterial DNA, due to their high entropy and few repeating regions, which make it harder to encode or compress. Genomic sequences are used for training and testing instead of genomic reads, whereby we omit the obtained compression ratios.

III. METHODOLOGY

A. Data preprocessing

We use four available datasets, shown in Table I. Prior to training each model, we convert the ASCII formatted data to a one-hot encoded format. This introduces a lot of memory overhead, if we assume 32-bit integers, but can be generally solved by storing the latter data in binary format.

B. Choosing a model

B.1 Predictive neural network (PNN)

The first proposed model for DNA read compression is called a predictive neural network (PNN). The purpose of a PNN is learning to predict the next read, given the previous read. We can exploit the redundancy that exists between neighbouring reads. The model could learn that the next read will probably yield a similar structure. Important to know is that this requires knowledge about how and when reads are shifted or duplicated.

B.2 Flat autoencoder (FAE)

Flat autoencoders are special types of artificial neural networks, which are used for compression. It learns to transform inputs to a lower dimensional space, where the learned code-words are then used to reconstruct the original inputs [9][10]. Rather then predicting new reads, as seen in PNNs, we are going to transform each read into a smaller representation. These networks will have to gain insight into the structure of every read. Impliedly that this will require to have more understand-
<table>
<thead>
<tr>
<th>Model</th>
<th>Dataset</th>
<th>Reconstruction accuracy(µ)</th>
<th>bpb(µ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAE(10x10,1)</td>
<td>E. coli</td>
<td>89%</td>
<td>1.01</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>91%</td>
<td>0.87</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>82%</td>
<td>1.29</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>89%</td>
<td>0.94</td>
</tr>
<tr>
<td>CAE(30x30,1)</td>
<td>E. coli</td>
<td>89%</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>91%</td>
<td>0.61</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>80%</td>
<td>1.11</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>90%</td>
<td>0.63</td>
</tr>
<tr>
<td>CABAC¹</td>
<td>E. coli</td>
<td>-</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>-</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>-</td>
<td>1.09</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>-</td>
<td>0.12</td>
</tr>
<tr>
<td>no CABAC¹</td>
<td>E. coli</td>
<td>-</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>-</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>-</td>
<td>1.67</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>-</td>
<td>0.42</td>
</tr>
<tr>
<td>7-zip Ultra¹</td>
<td>E. coli</td>
<td>-</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>-</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>-</td>
<td>1.13</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>-</td>
<td>0.11</td>
</tr>
</tbody>
</table>

TABLE II
TEST RESULTS FOR DIFFERENT CAE MODELS (PATCH SIZE, HIDDEN UNITS) AND OTHER STATE-OF-THE-ART COMPRESSION TECHNIQUES.

¹ Obtained bpb’s merely represent the lowest seen bpb’s, during compression. This means that the mean values will be higher for a given dataset.

B.3 Convolutional autoencoder (CAE)

Inspired by the biological visual cortex, as seen in many organisms including humans, convolutional neural networks (or ConvNets, CNNs) are (deep) feature extraction models which are capable of visual pattern recognition, without any preprocessing, in data such as video or image [11]. Taking in account the visual cortex, being the most powerful visual processing system in existence, it seems reasonable to emulate its behaviour. CNNs try to achieve this behaviour by using hierarchical layers/building blocks: convolutional layers, pooling layers and dense layers for classification. In convolutional layers, for each feature map, a filter is convolved across the input where typically less parameters are necessary in contrast to fully dense layers. We say that these layers scale better to larger input sizes. When training and validating on a particular dataset, the latter two sets have to be separated by using a gap set, in order to avoid redundancy across the sets. After validating, we can train on three datasets and finally test on the remaining dataset, in order to assess generalization.

C. Validation and testing

When choosing an appropriate model, we also have to find a way to train, validate and test the model on the given types of datasets (see Table I). It is important to have a good training and testing strategy in order to yield valuable models (i.e. in terms of generalization). Nevertheless, we also want to have an idea on how the trained models have to be used. For instance, we could train a deep learning model on a particular type of dataset in order that it generalizes well on new datasets, originating from the same organism or biological concept. Or we could extend our models by choosing to train in such a way that it generalizes well on unseen data. Nevertheless, we take in mind that when training and validating on a particular dataset, the latter two sets have to be separated by using a gap set, in order to avoid redundancy across the sets. After validating, we can train on three datasets and finally test on the remaining dataset, in order to assess generalization.

D. Training details

Training and testing is done by using the Data Science Lab’s HPC with 131.92 GB of RAM and (40 cores) Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz. Training the models are done by using a slight $L_1$ regularization with penalty term $\lambda = 1e^{-5}$ and AdaGrad for optimization with learning rate $\eta = 0.01$. The weights and biases in the whole network are initialized by using Xavier/Glorot initialization. As nonlinear activation function, we use a sigmoid function together with the cross-entropy loss function as cost function. We use mini-batch training with size 32, where 2-folds are chosen for cross-validation.
We find that AdaGrad yields faster convergence in comparison to other optimization techniques such as Nesterov momentum. This can be explained by the fact that we are working with relatively sparse data whereby AdaGrad is ideally used in the latter case. We also use a slight $L_1$ regularization in the whole network, as it induces sparsity in the weight space and yields better performances for CAE models. The use of sigmoids is motivated by the fact that it saturates in a range $[0, 1]$, the same range as for the inputs and outputs. Due to the huge overhead that comes with one-hot encoded data, we are forced to use 166400 reads in the training set and 41600 reads in the validation set, separated by a gap set consisting of 20% of the dataset. For testing we use 187200 reads for training and 10000 reads for testing.

IV. RESULTS

We saw that PNNs behave as sequential compression algorithms where random access functionality is difficult to achieve. Besides that, it is important to mention that PNNs yield less scalability. With the latter we mean that if would use PNNs to predict consecutive multiple reads (given multiple previous reads), the input and parameter space would increase tremendously due to the fully connectivity property in artificial neural networks. Nevertheless, we obtained a maximum prediction accuracy of 73%, when using one layer with 1024 hidden units for the E. coli dataset.

FAE models suffer from the same scalability problem that we’ve seen in PNNs. Together with the latter issue, we observed relatively low reconstruction accuracies (at the decoder side) of 64% on the E. coli dataset. This can be explained by the fact that FAEs don’t exploit the redundancy across multiple reads, forcing them to learn the language of individual DNA reads – which is hard to accomplish.

For the last model, we used two different architectures where 2 convolutional-max-pooling layers were used for the CAE(30x30,1) model. The number of feature maps in the subsequent convolutional layers were chosen as 64 and 256. Together with 3x3 filters in each convolutional layer and 2x2 filters for each max-pooling layer. The feature extraction part was then followed by 3 hidden layers with 1024, 128 and 1 hidden unit(s) respectively. The network is fed with 30x30 DNA read patches. Analogously, for the CAE(10x10,1) model, we used one convolutional-max-pooling layer with 256 feature maps and 10x10 DNA read patches as input. The dense part is defined similar as seen for CAE(30x30,1). The results, in terms of reconstruction accuracies and bpb, for each architecture are shown in Table II. It is clear that CAEs can successfully compress DNA reads originating from an organism or dataset, even when they were not used in training. CAEs have thus been proven to highly generalize on new datasets, even when they are originating from different species. When comparing the obtained results for the Homo sapiens (low coverage) dataset with other state-of-the-art compression techniques, we can see that our models yield better compression ratios in contrast to 7-zip and no-CABAC.

Altogether, we thus propose a novel abstract framework for DNA read compression depicted in Figure 2 and 3. The residue transform is necessary to allow for lossless compression (i.e. residues are correction words), whereas the entropy encoder and codeword encoder are used to further compress the obtained codewords and residues. For obtaining the bpb’s in Table II, we assumed a simple first-order entropy encoder. We neglected compression by the codeword encoder. However, we suggest to use quantization techniques to further compress the 32-bit integer codewords.

V. CONCLUSION

Under the assumption that reads are ordered, a sequential approach of compression can be achieved by using PNNs. This is done by predicting a read, given the previous read. We’ve shown that PNNs are less interesting due to the fact that random access is limited. Using this observation, we moved to FAEs which generally solved the random access issue in PNNs. However, this comes with an additional storage cost, by means of storing a codeword next to the correction term (i.e. residue). Besides the overhead in storage, we also encountered low compression rates since redundancy across consecutive reads are not exploited in FAEs. This motivated us to introduce CAEs, allowing for:

- **High compression:** due to the spatial feature extraction in
convolutional layers, CAEs allow to exploit redundancy across consecutive reads.

- **Input scalability**: scaling (i.e. increasing input) is done more efficiently due to the local connectivity property within convolutional layers, in contrast to fully connected dense layers.
- **Random access**: due to the nature of autoencoders, non-sequential based compression (in contrast to PNNs) is allowed, therefore allowing for random access.

It has been proven that CAEs yield remarkable results, due to the fact that they are able to successfully compress DNA reads, originating from an organism or dataset, even when it’s not used in training. Consequently, CAEs have thus been proven to highly generalize on new datasets, irrespective of whether they are originating from different or same species. For the low redundant Homo sapiens dataset, we obtained better bpb’s in contrast to other state-of-the-art techniques.

**REFERENCES**

## Contents

1 Introduction ........................................... 1
   1.1 Genome sequencing ........................................... 1
   1.1.1 Genome ........................................... 1
   1.1.2 Why genome sequencing? .................................... 3
   1.2 Genome sequencing technologies ......................... 4
   1.2.1 Sanger DNA sequencing ................................. 4
   1.2.2 Next-generation sequencing (NGS) ...................... 5
   1.3 Genomic data and the need for compression ............. 7
   1.3.1 Genomic data .................................... 7
   1.3.2 Storage of genomic data .............................. 8
   1.3.3 The need for compression of genomic data .......... 9
   1.4 Current compression techniques ......................... 10
   1.4.1 Naive bit encoding ................................ 11
   1.4.2 Dictionary-based encoding ............................ 11
   1.4.3 Statistical encoding ................................ 12
   1.4.4 Reference-based encoding ......................... 14
   1.4.5 Video techniques ................................. 15
   1.4.6 Machine learning techniques ..................... 16
   1.5 Conclusion ........................................... 16

2 Machine learning concepts .......................... 18
   2.1 Introduction ........................................... 18
   2.2 Types of data and learning ............................. 19
   2.2.1 Supervised learning ................................ 19
   2.2.2 Unsupervised learning .............................. 20
   2.2.3 Reinforcement learning ......................... 21
   2.2.4 Hybrid approaches ............................... 21
   2.3 Overfitting, underfitting and regularization .......... 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.1</td>
<td>Overfitting and underfitting</td>
<td>21</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Overfitting and underfitting: bias-variance</td>
<td>22</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Regularization</td>
<td>24</td>
</tr>
<tr>
<td>2.4</td>
<td>Steps in machine learning</td>
<td>24</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Problem and data analysis</td>
<td>24</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Feature extraction</td>
<td>24</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Select model and training approach</td>
<td>25</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Optimize model and evaluate</td>
<td>25</td>
</tr>
<tr>
<td>2.5</td>
<td>Shallow versus deep learning</td>
<td>26</td>
</tr>
<tr>
<td>3</td>
<td>Deep learning models for DNA read compression</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>Problem and data analysis</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>Deep learning models</td>
<td>30</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Artificial Neural Networks (ANNs)</td>
<td>30</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Predictive Neural Networks (PNNs)</td>
<td>40</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Flat Autoencoders (FAEs)</td>
<td>41</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Recurrent Neural Networks (RNNs)</td>
<td>42</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Convolutional Autoencoders (CAEs)</td>
<td>44</td>
</tr>
<tr>
<td>3.3</td>
<td>Training and testing</td>
<td>47</td>
</tr>
<tr>
<td>3.3.1</td>
<td>General strategy</td>
<td>47</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Evaluation metrics</td>
<td>49</td>
</tr>
<tr>
<td>3.4</td>
<td>Storage</td>
<td>50</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Residual storage</td>
<td>50</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Storage for PNNs</td>
<td>52</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Storage for FAEs and CAEs</td>
<td>52</td>
</tr>
<tr>
<td>4</td>
<td>Results</td>
<td>54</td>
</tr>
<tr>
<td>4.1</td>
<td>Software and hardware specifications</td>
<td>54</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Software</td>
<td>54</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Hardware</td>
<td>55</td>
</tr>
<tr>
<td>4.2</td>
<td>PNN models</td>
<td>55</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Architecture and (hyper)parameters</td>
<td>55</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Training and validation</td>
<td>57</td>
</tr>
<tr>
<td>4.3</td>
<td>FAE models</td>
<td>58</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Architecture and (hyper)parameters</td>
<td>58</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Training and validation</td>
<td>59</td>
</tr>
<tr>
<td>4.4</td>
<td>CAE models</td>
<td>61</td>
</tr>
</tbody>
</table>
4.4.1 Architecture and (hyper)parameters ........................................ 62
4.4.2 Training and validation ......................................................... 64
4.4.3 Testing ............................................................................... 65
4.4.4 Memory usage and time complexity ........................................ 66

4.5 Proposed novel framework for DNA read compression ............... 71
  4.5.1 Encoder ........................................................................... 71
  4.5.2 Decoder ........................................................................... 71
  4.5.3 Complexity ....................................................................... 72

5 Conclusions and future work ...................................................... 75
  5.1 Conclusions ......................................................................... 75
  5.2 Future work .......................................................................... 76

Bibliography .................................................................................. 80
List of Figures

1.1 The speed of genome sequencing has been increasing exponentially, due to lower costs [38]. ................................. 2
1.2 Simple representation of a double DNA strand (double helix), consisting of nucleotides [9]. ......................................... 3
1.3 The role of dNTP’s and ddNTP’s in sanger sequencing: a primer is added whereafter dNTP’s are used for filling and ddNTP’s for the termination of DNA strands (template strand) [3]. ........................ 5
1.4 Visual result after gel electrophoresis and autoradiography/UV light. The sequence is shown from bottom to top [4]. ....................... 6
1.5 Multiple fragmented sequence reads, aligned to their reference genome sequence. We can clearly observe a lot of redundancy across multiple reads [52]. ........................................ 8
1.6 Number of base pairs per haploid genome. In general, the genome size increases with the organism’s complexity. ..................... 8
1.7 A simple example of one (protein) sequence in FASTA format. ........ 9
1.8 Huffman tree. ....................................................... 13
2.1 Machine learning and overlap with different disciplines. ............... 19
2.2 Overfitting and underfitting in machine learning models. Complex models will tend to memorize the data, without learning effectively. The latter can be seen by a decreasing training error and increasing validation error. ... 22
2.3 Bias and variance in machine learning [28]. .......................... 23
2.4 Typical execution flow when performing machine learning [28]. ...... 25
3.1 First ten DNA reads of the MiSeq_Ecoli_DH10B.fasta file (FASTA description lines omitted). ................................. 28
3.2 Ten DNA reads of the homosapiens_filtered.fasta file (FASTA description lines omitted). ................................. 28
3.3 Biological neuron annotated. Dendrites can be seen as the inputs of the neuron, while axon terminals denote the outputs to other neurons [51]. 31
3.4 Diagram for the two-layer neural network as described in Section 3.2.1. The input, hidden and output variables are represented by nodes, while the weight parameters are shown as links between the nodes. Bias parameters are denoted by links coming from black nodes [8]. .......... 32
3.5 Sigmoid function .......................................................... 33
3.6 Cost function plotted as a hypersurface over the weight space. \( w_A \) denotes a global minimum, while \( w_B \) denotes a local minimum. At any point \( w_C \), the gradient is given by \( \nabla E \) [8]. ................................................................. 35
3.7 Example architecture for a PNN network where the input layer is denoted by green and output layer by red. Hidden layers are shown in black, with given number of hidden units. Note that RL stands for read length. .......... 40
3.8 Example architecture for a FAE network where the input layer is denoted by green and output layer by red. Hidden layers are shown in black, with given number of hidden units. Note that RL stands for read length and CL for the codeword length (number of hidden units in bottleneck layer). .. 42
3.9 Fully connectivity ......................................................... 44
3.10 Local connectivity .......................................................... 44
3.11 Example of a CAE architecture where the first part (encoding) is shown. The blue layer at the end represents the bottleneck layer which produces the codeword. Two conv-pool layers are shown for deep feature extraction whereafter the features are used as input to the encoder network (similar as in the FAE networks). The decoding part has been omitted in order to aid readability. ................................................................. 46
3.12 Unpooling (2x2) operation. .............................................. 47
3.13 How training and validation is done for a given deep learning model. .......... 48
3.14 How final testing is done for a given deep learning model. Every row represents a training and testing iteration. ................................................................. 48
4.1 Learning curve and accuracy in function of epochs for a PNN model with one hidden layer and 1024 hidden units. ................................. 57
4.2 Plotted gradients for different layers and different FAE architectures. Subfigures a, b and c represent the plotted gradients for a FAE(256-128-50) network, while Subfigure d represents the plotted gradients for a FAE(90) network. Comparing d with a, together with the scale of values, confirms the suspicion of the vanishing gradients issue in the FAE(256-128-50) network. 61
4.3 Learning curve and accuracy in function of epochs for a CAE model (30×30 patches and 32-bits codewords). Validated on the E. coli dataset. .... 68
4.4 Learning curve and accuracy in function of epochs for a CAE model (30×30 patches and 32-bits codewords). Validated on the Homo sapiens (high coverage) dataset. ................................................................. 68
4.5 Learning curve and accuracy in function of epochs for a CAE model (30×30 patches and 32-bits codewords). Validated on the Homo sapiens (low coverage) dataset. ................................................................. 69
4.6 Learning curve and accuracy in function of epochs for a CAE model (30×30 patches and 32-bits codewords). Validated on the RNA dataset. .... 69
4.7 Error distribution (normalized) for each base and file, obtained after validation. ................................................................. 70
4.8  Error distribution (normalized) for each base and file, obtained after testing.  70
4.9  Architecture for the encoding part of the proposed CAE compression framework.  73
4.10 Architecture for the decoding part of the proposed CAE compression framework.  74
### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANN</td>
<td>Artificial Neural Network</td>
</tr>
<tr>
<td>RNN</td>
<td>Recurrent Neural Network</td>
</tr>
<tr>
<td>DNA</td>
<td>Deoxyribonucleic acid</td>
</tr>
<tr>
<td>NGS</td>
<td>Next generation sequencing</td>
</tr>
<tr>
<td>ddNTP</td>
<td>Dideoxynucleotide</td>
</tr>
<tr>
<td>bpp</td>
<td>bits per base</td>
</tr>
<tr>
<td>HMM</td>
<td>Hidden Markov Model</td>
</tr>
<tr>
<td>CNN</td>
<td>Convolutional Neural Network</td>
</tr>
<tr>
<td>PCA</td>
<td>Principal Component Analysis</td>
</tr>
<tr>
<td>FA</td>
<td>Factor Analysis</td>
</tr>
<tr>
<td>PNN</td>
<td>Predictive Neural Network</td>
</tr>
<tr>
<td>FAE</td>
<td>Flat Autoencoder</td>
</tr>
<tr>
<td>CAE</td>
<td>Convolutional Autoencoder</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

High-throughput sequencing technologies led to a dramatic decline in genome sequencing costs together with a tremendous increase of genomic data (see Figure 1.1). These evolutions led to ambitious and huge projects, e.g. the 1000 Genomes Project [19]. This is one of the first projects to sequence the genome of a large number of people, to provide a comprehensive resource on human genetic variation. In order to process and store large amounts of genomic data, a lot of capacity is needed when it comes to storage and transmission. Therefore a recent surge of interest has been emerged for genomic (high-performance) data compression tools in order to allow for efficient, low-cost storage and management of genomic data. In the course of this masters dissertation, compression techniques will be studied by the use of machine learning on genomic (read) data. In this introduction we will discuss how genomes are currently sequenced, which types of genomic data are available and which techniques are currently used for compression. Finally, we will conclude this introduction with a strong emphasize on the need of good compression techniques.

1.1 Genome sequencing

1.1.1 Genome

In molecular biology and genetics, the term genome is used to refer to the genetic material of an organism. This material consists of deoxyribonucleic acid (DNA), or ribonucleic acid (RNA) in certain viruses, which includes non-coding sequences (these sequences do not encode proteins) and coding sequences.

DNA is a molecule which stores crucial biological information. It mostly contains genetic instructions for the development, functioning and reproduction of all known living organisms and many viruses. Most DNA molecules consist of two biopolymer strands
Figure 1.1: The speed of genome sequencing has been increasing exponentially, due to lower costs [38].

coiled around each other to form a double helix, as it can be seen in Figure 1.2. These two biopolymer strands are also known as polynucleotides due to the fact that they are composed of smaller units called nucleotides. Next to a monosaccharide sugar and a phosphate group, nucleotides are also defined by a nitrogenous nucleobase which can be one of the following chemical substances:

- Cytosine (C)
- Guanine (G)
- Adenine (A)
- Thymine (T)

When dealing with ribonucleic acid (RNA), which is also a nucleic acid like DNA and is used for translating instructions from DNA to make proteins in your body, we typically introduce a fifth chemical substance called uracil (U) which replaces the abovementioned nucleobase thymine. A chromosome is then a packaged and organized structure containing most of the DNA of a living organism.
1.1.2 Why genome sequencing?

In molecular biology and genetics, scientists use genomic sequences to study various concepts within living organisms. A genome or genomic sequence is represented by the order of DNA nucleotides (or nucleobases as described in Section 1.1.1). We can thus define genome sequencing as finding the order of nucleotides in a genome (i.e. the order of A,C,G and T that make up an organism’s DNA). It is important to emphasize that genome sequencing is not related to the “decoding” of a genome, i.e. it does not immediately explain the genetic secrets of an entire species. We still have to translate the resulting order of nucleobases into an understanding of how the genome works. This falls outside the subject of this masters dissertation.

At this moment it is only important to remember that genome sequencing is an important step towards understanding it:

1. Efficiently locate genes within the genome.
2. Understand how the genome works as a whole.
3. Identify different regulatory regions (i.e. regions that control how genes are turned on and off).
Genome sequencing allows us to digitally store genomic information whereafter further processing or compression can be performed.

1.2 Genome sequencing technologies

Throughout the last decennia, numerous of techniques have been raised to perform genome sequencing. We will only discuss the most important technologies and the evolution from what is called Sanger DNA sequencing towards the current next-generation sequencing (NGS) which led to much cheaper and faster sequencing of DNA. Again, it is straightforward that we will not put a lot of focus on the discussion of sequencing technologies but nevertheless it is important to have a decent understanding of how the data, on which compression is being performed, is being generated. A good understanding of how the data generation process works, is crucial to perform good data analysis and subsequent data compression.

1.2.1 Sanger DNA sequencing

Developed by Frederick Sanger and colleagues in 1977, the Sanger’s method, also referred to dideoxy sequencing or chain termination, is based on the use of dideoxynucleotides (ddNTP’s) which have a similar structure as normal nucleotides found in DNA. The only difference lies in the fact that they contain a hydrogen group on the 3’ ("three prime end") carbon instead of a hydroxyl group (OH). These special nucleotides, which can be placed at the end of a DNA chain, prevent that additional nucleotides are bounded which results in a termination of this chain, hence its name chain termination.

The method

Denaturation of our DNA is the first step prior to the actual sequencing. In this process, heat is applied to the sequence which converts the double stranded DNA (i.e. double helix) in single stranded DNA. The latter process results in two strands called the template strand and the complementary strand. A primer is then applied to the template strand which allows for the addition of nucleotides later on. The latter process is illustrated in Figure 1.3.

Next, four reaction mixtures are set up where the template strand, with attached primer, is being added. This is the DNA which will be sequenced. To this we add DNA polymerase to each reaction mixture, together with free nucleotides (dNTP’s). On top of that, one of these dNTP’s is radiolabeled (i.e. isotopic labeled in order the track the passage of the nucleotide through the reaction), which helps in the determination of the DNA sequence.
Finally, we add the aforementioned modified nucleotides or ddNTP’s in such a way that only one type of each is added to each mixture. As described above, the addition of the latter modified nucleotides then results in chain termination, which assures the blocking of further nucleotide additions to each chain. At the end, the resulting DNA fragments are heat denatured and separated by size using gel electrophoresis, as shown in Figure 1.4. This is done by using denaturing polyacrylamide-urea gel with each of the four reactions run in one of four individual lanes (lanes A, T, G, C). The DNA bands may then be visualized by autoradiography or UV light and the DNA sequence can be directly read off the X-ray film or gel image.

When it comes to sequence storage on a hard disk, the manual read off method is mostly omitted and working with laser technology, which automatically detects the order of nucleotides within the DNA sequence, is preferred [42][14][15].

### 1.2.2 Next-generation sequencing (NGS)

Using (electrophoresis-based) Sanger sequencing, the Human Genome Project took over ten years together with a cost of nearly $3 billion. With the attendance of NGS methods, large-scale whole-genome sequencing became more accessible and practical for the average researcher. Sanger sequencing is relatively slow and comes with a high cost for sequencing millions of nucleotides, e.g. sequencing of only a few thousand nucleotides takes a week. In contrast, NGS methods are fast, easy to operate and very cost-effective. NGS methods can easily sequence around 200 billion nucleotides in a week. To give the reader a more...
Figure 1.4: Visual result after gel electrophoresis and autoradiography/UV light. The sequence is shown from bottom to top [4].

The general idea of this high-level advancement: **we can now sequence the entire genome of any individual within a couple of hours.** The term next-generation sequencing denotes a family of different methods where the sequence process is heavily parallelized (i.e. sequencing-by-synthesis operations run in parallel), producing thousands or millions of sequences concurrently. The latter is responsible for the high-throughput characterization of NGS methods [18][30][48].

As of today, multiple NGS methods exist [36] [33]:

- Massively parallel signature sequencing (MPSS).
- Polony sequencing.
- 454 pyrosequencing.
- SOLiD sequencing.
- Ion Torrent semiconductor sequencing.
- DNA nanoball sequencing.
- Helicos single molecule sequencing.
- Single molecule real time (SMRT) sequencing.
Illumina (Solexa) sequencing.

The available sequence/genomic data, which has been used throughout this masters dissertation, originates from Illumina’s MiSeq sequencer machine where Illumina sequencing is used as NGS method. This method is based on sequencing-by-synthesis with increased parallelisation. The MiSeq (desktop) instrument allows for quick and inexpensive runs, which can be ideally used for smaller DNA samples [13].

1.3 Genomic data and the need for compression

1.3.1 Genomic data

We’ve discussed some basic aspects about genome sequencing and the evolution from Sanger sequencing to next-generation sequencing. More important, we want to focus on the data that is produced by various sequencing methods. This masters dissertation has been split in two different topics, due to two different types of genomic data that can be stored on a hard drive.

Genomic read data

Due to the nature of the most sequencing technologies, the output mostly consists of DNA fragments or reads. These reads are aligned directly to the reference genome (see Figure 1.5), and this alignment (which usually covers each position of a genome multiple times) is used for further processing. For instance, the next step can consist of assembling the reads together, based on their overlapping areas, to fully recreate the reference genome sequence. This process will then lead to whole genomic data, representing the complete genomic sequence. Many computational challenges exist, in order to achieve this, such as the evaluation of the raw sequence data which is done by programs and algorithms such as Phred and Phrap. The attentive reader will also notice the high data redundancy, which occurs within the total set of reads: a lot of DNA reads tend to overlap with each other. For this, other challenges exist to deal with these repetitive sequences that prevent complete genome assemblies (since they occur in many places of the genome). We can conclude that the production of raw sequence data is only the beginning of the subsequent bioinformatical analysis [23].

Whole genomic sequence data

After assembling genomic reads, using special algorithms, we obtain the complete genomic sequence, which can then be stored on a hard drive. In Figure 1.6, we can see the
Figure 1.5: Multiple fragmented sequence reads, aligned to their reference genome sequence. We can clearly observe a lot of redundancy across multiple reads [52].

Figure 1.6: Number of base pairs per haploid genome. In general, the genome size increases with the organism's complexity.

storage sizes of different haploid genomes (i.e., genomes which contain a single set of chromosomes). In general, the genome size increases with the organism's complexity.

1.3.2 Storage of genomic data

For the actual storage of genomic data, different formats exist. One of the most frequent used formats is called the FASTA format, which is used in this work. FASTA is a text-based format for representing sequences (protein or genomic), in which nucleotides are represented using single-letter codes (as described in Section 1.1.1). The format also allows for sequence names and comments to precede the sequences. The format originates from the FASTA software package, but has now become a standard in
the field of bioinformatics. It is important to note that the simplicity of FASTA format makes it easy to manipulate and parse sequences using text-processing tools and scripting languages like Python, Ruby, and Perl.

Using the FASTA format, a sequence starts with a single-line description, followed by lines of sequence data. The description line is distinguished from the sequence data by the “>” symbol in the first column. The word following “>” denotes the identifier of the sequence, and the rest of the line is the description. The sequence ends if an other line starting with a “>” appears; this indicates the start of a new sequence. For instance, in the above example (Figure 1.7) "gi|31563518|ref|NP_852610.1|
MKMRFFSSPCGKAAVDPADRCKEVQQIRDQHPKIVIPVIIIERYKGEKQLPVLDKTKFLVPDHVNMSELVKIIRRRLQLNPTQAFFLLVNQHSMVSVSTPIADITYEQKD"

is the name of the sequence.

1.3.3 The need for compression of genomic data

We end this section by stressing the need for compression, when it comes to the storage of genomic data. In the previous subsections, we made a distinction between two different types of genomic data:

1. Genomic read data.

2. Whole/complete genomic sequence data.

When it comes to genomic read data, one file can easily contain $10^8$ DNA reads, each containing 150 nucleotide characters (A,C,G,T). If we assign eight bits for one nucleotide character (storing in ASCII format), the size would be on average 1.9 GB, while test files are even much larger. Taking in account that the FASTA description lines have been excluded, the need for compression becomes quite naturally: compression would yield less bandwidth for sending and space for the storage of DNA data.

The goal of this masters dissertation (which has been split up in two parts, based on the different types of data) is to look at compression techniques, using machine learning, for both genomic read data and whole genomic sequence data. During the course of this dissertation we will focus on the compression of genomic read data without using reference genomes (i.e. compression purely on genomic reads without the aid of a reference genome).
1.4 Current compression techniques

In the previous section we pointed out the need for compression techniques for genomic data, due to the high-throughput sequencing methods. Currently, sequence databases contain more data than any scientist can handle. In the future, this situation will become even worse:

1. The decreasing sequencing cost per genome makes huge projects more feasible (e.g. the 1000 Genomes Project).

2. More individuals are interested in their genome and its genetic predispositions.

3. Sequencing platforms tend to produce more and longer reads, which increases the possible throughput.

A naive way of encoding genomic sequences would be to assign eight bits for each possible nucleotide character. We are using the word naive to stress out the overhead that comes along with this type of encoding. Indeed, let’s define the alphabet (in practice, other alphabets can be used) of all possible nucleotide characters as

\[ \mathcal{L} = \{A, C, G, T\} \]  

Since the cardinality of \( \mathcal{L} \) is 4, we would need

\[ n = \lceil \log_2 |\mathcal{L}| \rceil = 2 \]  

bits to encode each nucleotide character as

\[ A \rightarrow 00, \]
\[ C \rightarrow 01, \]
\[ G \rightarrow 10, \]
\[ T \rightarrow 11. \]  

Thus, six bits would be unused or in general 75% of the total storage for a genomic sequence would be wasted, making this naive way highly space-intensive. Notwithstanding the fact that this representation is wasteful in terms of space, because it does not compress the sequence at all, many scientists are still accustomed to this format: it allows applications and programming languages easy access to the data and it’s easy to read by human beings.

In order to allow efficient storage and transmission, the introduction of valuable compression techniques are necessary. The term compression is used to denote methods for storing
or encoding the information, in such a way that less space is required. Compression can be divided in two different classes:

1. **Lossy compression**: data is encoded in such a way that less space is required, possibly yielding some kind of information loss (e.g. lossy compression in JPEG).

2. **Lossless compression**: data is encoded in such a way that less space is required and such that the original data can be fully reconstructed (e.g. Lempel-Ziv-Welch-algorithm or LZW compression used in ZIP archive files).

In the following sections we will discuss some basic techniques, which are currently being used for the compression of sequenced genomes. Again, the distinction has to be made between two kinds of genomic data (as already explained in Section 1.3) which results in two kinds of genomic compression classes: whole genome compression and read compression. Again, due to the course of this dissertation, we are mainly going to focus on read compression.

### 1.4.1 Naive bit encoding

We started this chapter by explaining how a naive way of storing DNA sequences can be performed, using one byte for every nucleotide character, together with its disadvantage in terms of memory usage. We can improve this idea by looking at the number of bits available in one byte, which can be used to store multiple nucleotides. Again, taking into account the cardinality of the abovementioned $L$, one can easily conclude that we can store

$$k = \frac{8}{\lceil \log_2 |L| \rceil} = 4$$

(1.4)

nucleotides, using one byte. This binary encoding scheme yields 2.0 bpb (bits per base), which allows for a simple compression of four nucleotides in one byte. Note that during the further course of this dissertation, we will mostly talk about genomic sequence/read compression in terms of bits per base.

It is important to mention that in practice (for genomic sequences), $L$ is usually extended with an additional symbol $N$, which denotes an error. The most sequencing methods are prone to errors, i.e. some nucleotides can be undefined. In the latter case, we will mostly use one byte for the storage of two nucleotides whereby the remaining two bits are used as metadata [43].

### 1.4.2 Dictionary-based encoding

In dictionary-based or substitutional compression algorithms, repeatedly occurring substrings are replaced by references to a dictionary (i.e. this is a set of previously seen
or predefined common strings). This dictionary can be constructed at runtime or offline. In the case of at runtime construction, the dictionary itself does not have to be stored along with the compressed data. Well-known dictionary-based algorithms are the Lempel-Ziv-based compression algorithms, such as LZ77 or LZ78. The input sequence is sequentially parsed and examined for reoccurring substrings. Substrings that have not been encountered before are added to the dictionary in the form of a reference to a previously encountered substring plus one new character. In the case of integer sequences, one can use similar Golomb codes (which encode small numbers more efficiently than large numbers) and Fibonacci codes (which are more tolerant to failures). Many dictionary-based algorithms currently exist, like CASToRe, which is a modification of the above-mentioned Lempel-Ziv compression algorithm. The difference, in comparison to Lempel-Ziv compression, is that sequences are compared against the dictionary, whereby the registration of new dictionary entries are concatenations of two previously parsed subsequences in the dictionary. POMA, a particle swarm optimization-based algorithm for DNA read compression, is also an important dictionary-based algorithm. Differentiation between four different kinds of repeat patterns (i.e. direct, mirror, pairing and inverted repeats) is used where the most commonly repeated fragments are identified and included in a dictionary.

In Table 1.1, the resulting bpb rates are given for the two dictionary-based compression algorithms (given two different kinds of data sources): We can see that both algorithms, on average, outperform the naive bit encoding scheme (where 2.0 bpb is obtained) [43].

<table>
<thead>
<tr>
<th>Tool</th>
<th>Data source</th>
<th>Compression ratio (bpb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASToRe</td>
<td>14 genomes of different species</td>
<td>1.909-2.056</td>
</tr>
<tr>
<td>POMA</td>
<td>11 short gene sequences</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Table 1.1: Non-referential (dictionary-based) compression schemes for DNA reads.

1.4.3 Statistical encoding

Statistical encoding, also called entropy encoding, constructs a probabilistic model of the input. Using partial matches of subsets of the input, these models try to predict the next symbols in the sequence. Compression rates depend on the quality of the model as well as the existence of detectable patterns in the input. A simple example of such statistical encoding algorithms, is the well-known Huffman encoding. Using the estimated probabilities for each possible symbol, originating from the sequence, a binary tree is created in which the leaf nodes correspond to the symbols and edges are labeled with probabilities and the derived codes. It is important to mention that the resulting Huffman code table has to be stored, together with the compressed stream,
which to be taken in account for the determination of the compression ratio. Huffman encoding tends to get very practical, in terms of compression ratios, when the alphabet is large together with an uneven distribution for the symbols. On the other hand, one should be careful when using Huffman encoding when the above-mentioned conditions are not satisfied. In the latter case, worse compression ratios would be seen. For example, let’s define an arbitrary DNA sequence \( X \) with alphabet \( L \) as

\[
X = \text{AAAGGCAAAATA},
\]

\[
L = \{A, C, G, T\}.
\]

Looking at the distribution of every symbol \( S \in L \), we find

\[
P_A = \frac{2}{3},
\]

\[
P_G = \frac{1}{6},
\]

\[
P_C = \frac{1}{12},
\]

\[
P_T = \frac{1}{12}.
\]

Symbols with a higher probability will yield a bigger increase in the entropy rate, in comparison to other symbols, of the stochastic process \( X \) (the sequence can be seen as a stochastic process). Based on these findings we iteratively construct a binary tree, where in each step the lowest binary tree’s (in terms of probability) are merged. Note that, initially every symbol (and its corresponding probability) is essentially a binary tree with one (root) node. The latter construction process terminates when only one binary tree is left. This tree is also known as the Huffman tree, where every edge is labeled with either 1 or 0 and every leaf node represents a symbol. Leaf nodes with a higher depth will yield a lower probability.

Using the above-defined probabilities, for every symbol \( S \in L \), we find the resulting Huffman tree in Figure 1.8. We then find for every symbol the following binary translations
\[ A \rightarrow 0, \quad C \rightarrow 100, \quad G \rightarrow 11, \quad T \rightarrow 101. \quad (1.7) \]

We can easily compare this to the naive bit encoding scheme (where 24 bits would be necessary to encode the sequence \( X \)): only eighteen bits are required when Huffman encoding is used. Arithmetic encoding is also a statistical encoding algorithm, which encodes longer strings – or even whole input streams – as a single float \( x \in [0, 1] \).

Cormack G. and Horspool N. [17] proposed a statistical encoding algorithm, based on hidden Markov models (HMM). DNA sequences are approximated by a hidden Markov model, on which statistical inference can be applied. The order of the model defines the number of symbols, which can be taken in account to predict the next symbols. It is important to mention that the compression rate, for statistical algorithms, mostly depends on the distribution of input symbols and the available memory for construction of frequency distributions. The compression rates of statistical algorithms are usually between 4 bpb and 8 bpb for DNA sequences. This can be explained by the fact that nucleotides are on average equally distributed within a genome, thus yielding worse compression rates [43].

### 1.4.4 Reference-based encoding

Reference-based encoding algorithms are quite similar to dictionary-based techniques, as described above. Long substrings (within a sequence) are replaced by references to another string. With the latter references, we denote external sequences (which are not a part of the initial, to-be-compressed input sequence). Daily K., Rigor P., Christley S., et al. [32], presented GenCompress. In this algorithm, sequences are aligned to a reference sequence with reference entries being composed of: starting position, match length and an optional difference list describing mismatches. The focus in GenCompress is on entropy encoding of integers via fixed or variable schemes, such as Huffman, Golomb or Elias. The algorithm only supports compression of the four nucleotides.

Similarly, Kozanitis C., Saunders C., Kruglyak S., et al. [7] proposed SLIMGENE, a lossless or lossy reference-based compression scheme focusing on how to find encodings of integers in order to minimize storage. The algorithm is based on Huffman and arithmetic encoding [43].

Again, due to the topic of this masters dissertation (i.e. non-reference read compression), we will not further focus on reference-based encoding. The results for the above-mentioned algorithms can be seen in Table 1.2.
### Table 1.2: Referential compression schemes for sequence reads.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Data source</th>
<th>Compression ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenCompress</td>
<td>three heterogeneous datasets</td>
<td>9.8:1 – 22.7:1</td>
</tr>
<tr>
<td>SLIMGENE</td>
<td>human short reads</td>
<td>39:1</td>
</tr>
</tbody>
</table>

1.4.5 **Video techniques**

In the past it has been proven that video techniques also seem to yield valuable results when it comes to DNA read compression. In [49] and [25], the integration and optimization of Context-Adaptive Binary Arithmetic Coding (CABAC) is discussed within a block-based framework for genomic (read) compression. The latter framework allows for random access functionality, stream processing, adaptive streaming and encryption next to DNA read and sequence compression. Using the above-mentioned techniques, compression ratios were reported well below 1.90 bpb. The best achieved compression is 1.63 bpb for the complete human Y genome. It is however important to mention that these ratios are resulting from whole genome compression rather than genomic read compression. In the following chapters we will focus more on the comparison and analysis of machine learning techniques for genomic read compression, in terms of bits per base and reconstruction accuracies. The most important reconstruction accuracies and bits per base measurements, obtained in [49] and [25], on different read datasets are shown in Table 1.3 and 1.4. We will also use these datasets for the training and testing of our machine learning models.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Homo sapiens (low)</th>
<th>E. coli (low)</th>
<th>HS RNA</th>
<th>HS (high)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSP</td>
<td>78.06%</td>
<td>98.91%</td>
<td>98.69%</td>
<td>98.04%</td>
</tr>
<tr>
<td>S2SP</td>
<td>74.91%</td>
<td>95.38%</td>
<td>82.17%</td>
<td>84.75%</td>
</tr>
</tbody>
</table>

Table 1.3: Highest accuracies for different datasets (and given redundancy) for different tools within the CABAC framework.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Homo sapiens (low)</th>
<th>E. coli (low)</th>
<th>HS RNA</th>
<th>HS (high)</th>
</tr>
</thead>
<tbody>
<tr>
<td>without CABAC</td>
<td>1.672</td>
<td>0.282</td>
<td>0.419</td>
<td>0.532</td>
</tr>
<tr>
<td>with CABAC</td>
<td>1.096</td>
<td>0.134</td>
<td>0.124</td>
<td>0.292</td>
</tr>
<tr>
<td>7-zip Ultra</td>
<td>1.135</td>
<td>0.135</td>
<td>0.115</td>
<td>0.244</td>
</tr>
</tbody>
</table>

Table 1.4: Lowest bpb’s for different datasets (and given redundancy) for CABAC, no-CABAC and 7zip Ultra.

The different datasets will be further described in the following chapters.
1.4.6 Machine learning techniques

At this moment, machine learning techniques are not widely used for DNA compression and more specific for DNA read compression. The lack of appropriate benchmarks makes it hard for the comparison and testing of various machine learning models in this dissertation. However, in [37] it was verified that artificial neural networks are useful in combination with other models for the compression of genomic data. In this work, artificial neural networks (ANNs) are used to reduce the high memory requirements, which come with the use of finite-context models (FCMs). The problem is that a lot of runtime memory is required to store high-order finite-context models. The authors were able to reduce these requirements by using ANNs to build probabilistic models in a compact way. Testing and training was mainly done on bacterial DNA, due to their high entropy and few repeating regions, which make it harder to encode or compress. Again, genomic sequences where used for training and testing instead of genomic reads. Some of the most important results are shown in Table 1.5.

<table>
<thead>
<tr>
<th>Encoder Name</th>
<th>bpb</th>
</tr>
</thead>
<tbody>
<tr>
<td>NN1</td>
<td>1.8491</td>
</tr>
<tr>
<td>NN2</td>
<td>1.8551</td>
</tr>
<tr>
<td>NN3</td>
<td>1.8395</td>
</tr>
<tr>
<td>7-Zip</td>
<td>2.1402</td>
</tr>
<tr>
<td>PAQ12a</td>
<td>2.0205</td>
</tr>
</tbody>
</table>

Table 1.5: Simulation results of three encoders used, over different bacterial sequences, together with some well-known compression standards.

1.5 Conclusion

In the previous section, different state-of-the-art compression techniques (which can be divided in six classes) and their performances against different types of data has been discussed. One can conclude that there is, at this moment, a lack of proper benchmarks for DNA compression, which makes the comparison of different techniques relatively hard. It is important that test sequences should come from different species and cover different sizes. Next to compression rate, other metrics like (de)compression time and maximum main memory usage during (de)compression, may also be useful to make an extensive comparison and analysis of different techniques. In the following chapters, we will mainly focus on compression techniques using machine learning or deep learning in general. As it will be described later, machine learning has already proven to be useful when it comes to compression such as image compression, sound compression etc. The question, which
will be discussed in the next chapters, remains if deep learning could also be useful for DNA read compression.
Chapter 2

Machine learning concepts

In the previous chapter we started off with a brief introduction to genome sequencing and the need for compression. Different compression techniques where discussed and a seperation was made between whole genome compression and genomic read compression. In this chapter we are going to discuss important machine learning concepts. These are crucial in the study of different machine learning/deep learning models for DNA read compression. In the next chapter we will then discuss the obtained models and techniques, used throughout our dissertation.

2.1 Introduction

Machine learning emerged as a subfield of computer science [6], from the study of pattern recognition and artificial intelligence. One of the most accurate descriptions for machine learning, given by Arthur Samuel in 1959, goes as follows: "Field of study that gives computers the ability to learn without being explicitly programmed" [40]. In general, machine learning deals with the study and creation of algorithms, which can learn from and make predictions on data. As Arthur Samuel already stated, machine learning is able to make data-driven predictions or decisions expressed as outputs, based on example inputs, without the need of strictly static program instructions. In Figure 2.1, we can see that machine learning has not only overlap with computer science but also probability theory and optimization theory. This will be further explained in detail, during the following two chapters. Machine learning is often confused with data mining [21], which are not directly related. The latter focuses more on data analysis and can be seen as unsupervised learning (we refer to Section 2.2). However, it is impossible to imagine data analysis without the use of machine learning, as this discipline allows to devise complex models and algorithms which return predictions. These analytical models allow researchers, data scientists and engineers to reason about decisions and results and allow them to gain "hid-
den insights” through learning from relationships and trends within (big) data [26][45].
We will further focus on various machine learning key concepts, which are important to
asses the performances of different machine learning models and to gain more insight into
machine learning itself. More advanced techniques and deep learning models, which were
used in this masters dissertation, will be discussed in the next chapter. The purpose of
this chapter is rather to give the reader more insight into machine learning.

2.2 Types of data and learning

In general, we can define four different types of learning classes based on the type of data
that we are dealing with (i.e. supervised learning, unsupervised learning, reinforcement
learning and hybrid aproaches).

2.2.1 Supervised learning

When performing machine learning on labeled training data, we call this task supervised
learning. The training data consists of a set $X$ of $N$ training examples, where each
example exist of a vector/array $x_i$ and a desired output label $t_i$

$$X = \{(x_1, t_1), \ldots, (x_N, t_N)\}. \quad (2.1)$$
For instance, in the domain of pattern recognition like traffic sign recognition, every $x_i$ could denote an image with an assigned class or output label $t_i$ which stands for a certain traffic sign class. Thus, in supervised learning, we are trying to model the relationship between input $x_i$ and output $t_i$. It is important to mention that in the most machine learning models, a transformation has to be performed on the training samples $\{x_1, \ldots, x_N\}$. This transformation projects the original samples in a new space, also known as the feature space. This is due to the fact that some machine learning models tend to learn better from transformed samples, represented in some feature space, rather than the original (sample) space. We can further divide supervised learning in two different classes, based on the type of labels that we are dealing with:

1. **Classification tasks**: labels $t_i$ are defined in a discrete space (e.g. traffic sign recognition/classification, digit recognition etc.).

2. **Regression tasks**: labels $t_i$ are defined in a continuous space (e.g. data fitting, sound compression etc.).

### 2.2.2 Unsupervised learning

When training examples do not include output labels (i.e. if the data is unlabeled), we can use machine learning to infer a function to describe hidden structure from this data. We call this task unsupervised learning. Due to the nature of unlabeled data, there is no error or reward that can be calculated. This distinguishes unsupervised learning from supervised learning.

Unsupervised learning techniques can generally be divided in two classes:

1. **Clustering tasks**: k-means clustering, mixture models, hierarchical clustering, etc.

2. **Dimensionality reduction tasks**: principal component analysis (PCA), factor analysis (FA), etc.

For instance, if every training sample $x_i$ is defined in a high-dimensional feature space, one can use unsupervised learning techniques like PCA in order to reduce the number of dimensions (i.e. reduce the size) while keeping the most important information. These techniques can be useful, prior to unsupervised learning, in order to eliminate the “curse of dimensionality” in machine learning. The latter can be described by the fact that if we have data samples in a in a high-dimensional feature space, with each feature having a number of possible values, an enormous amount of training data is required to ensure that there are several samples with each combination of values. This is necessary if we want to accurately model the relationship between input and output [20].

2.2.3 Reinforcement learning

Inspired by behaviorist psychology, reinforcement learning is concerned with software agents, which have to take actions in an environment in order to maximize some cumulative reward. Reinforcement learning is used in many disciplines, such as game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, statistics, and genetic algorithms.

2.2.4 Hybrid approaches

It is obvious that in this approach different learning approaches, as defined above, are combined. For instance, when PCA is used prior to supervised learning, we can generally say that both supervised and unsupervised learning was used [28].

2.3 Overfitting, underfitting and regularization

When a machine learning model is trained, it is important to have a good understanding on how accurate our model is on unseen data, or how well or model generalizes on new data. The choice of training data, validation data and test data are thus important in order to perform valuable analyses of our models. In general, we train a machine learning model on a given training set, while validating on a validation set. Cross-validation techniques can be used, to iteratively train and validate on different portions of a complete dataset and where the obtained error measurements are averaged. The rationale behind this idea is that each sample in a dataset would have been used as a training sample and validation sample. After training a particular model, we typically want to have an idea of whether we are overfitting or underfitting.

2.3.1 Overfitting and underfitting

When a model is too complex for our training set, the chance increases that our model is learning the training set by heart and is capturing the noise of the data. When this model would be validated on the validation set for a given error measurement (e.g. number of misclassifications in a classification task), we would see a much higher error in comparison to the error on our training set.

At the other hand, if a model is too simple, it would not use all the important information that the data contains and we would typically see higher error rates on both the
Figure 2.2: Overfitting and underfitting in machine learning models. Complex models will tend to memorize the data, without learning effectively. The latter can be seen by a decreasing training error and increasing validation error.

training set and validation set. The model is not learning well and yields an underperformance on the training and validation set.

The above-mentioned aspects can be visually assessed by plotting the relation between a chosen error measurement and the model complexity, as illustrated in Figure 2.2. When the optimal model complexity is chosen, we can finally test our model on a final test set, by training on both the validation and training set. The reason that we use an extra test set, is to make sure that we are not overfitting on the validation set. It is of great importance that the test and validation sets are chosen in a way, that the data contains unseen samples. This to optimally assess the generalization (i.e. how well does our model perform on unseen data) ability of our model. In order to ensure generalization we use a proper evaluation method, as described above, and use regularization techniques.

2.3.2 Overfitting and underfitting: bias-variance

In Figure 2.3, we can see four different models and their prediction accuracies denoted as blue dots. Correct predictions are located in the red center circle. With increasing error, the blue dots will be located further away from the center circle.

The error due to bias is given by the difference between the expected or average prediction of our model and the correct value that a model must predict. When bias is high (Figure
2.3, left column), we will typically have models which are not powerful enough. This can be due to a lack of features (i.e. recall feature space, where each dimension is seen as a feature) or features which are not useful, as well as the given data which is not informative enough.

The error can also be explained by high variance, when the variability of the predictions is too high (Figure 2.3, right column). This can be generally solved by using more data or by model averaging techniques [28]. Finally, we can talk about the bias-variance decomposition by looking at a predictive model. For instance, if we want to predict some outputs $Y$, given inputs $X$, we assume a (noisy) relationship between the inputs and outputs, given as

$$Y = f(X) + \epsilon,$$

$$\epsilon \sim \mathcal{N}(0, \sigma^2),$$

with $\epsilon$ the error term, which is normally distributed with zero mean and $\sigma^2$ variance.

We want to find a function $\hat{f}(X)$, that approximates the true function $Y = f(X)$, for instance by using a learning algorithm like linear regression. When training such algorithms, we will calculate the mean squared error between $Y$ and $\hat{f}(X)$ and try to minimize this error measurement $(Y - \hat{f}(X))^2$. Due to the noise introduced in (2.2), we will not expect to find a perfect approximation $\hat{f}(X)$. No matter which learning algorithm is used, it turns out that for a given unseen sample $x$, the expected error evaluates as

$$E[(y - \hat{f}(x))^2] = \text{Bias}[\hat{f}(x)]^2 + \text{Var}[\hat{f}(x)] + \sigma^2$$

(2.3)
where

\[
\text{Bias}[\hat{f}(x)] = E[\hat{f}(x)] - f(x) \tag{2.4}
\]

and

\[
\text{Var}[\hat{f}(x)] = E[(\hat{f}(x) - E[\hat{f}(x)])^2]. \tag{2.5}
\]

We call \( \sigma^2 \) the irreducible error. All the three terms in (2.3) are non-negative, so that \( \sigma^2 \) forms a lower bound on the expected error on unseen samples [50] [24].

### 2.3.3 Regularization

In order to reduce the effect of overfitting, one can choose to lower the complexity of the machine learning model. In general, this is called regularization. We also talk about regularization, when it comes to the following techniques: adding noise to data, \( L_1/L_2 \) regularization (will be explained in the next chapter, Section 3.2.1), data augmentation, early stopping. Most of these techniques will be described in the next chapter, when we will talk about advanced machine learning techniques for DNA read compression. It is however important to mention that different models will need different regularization techniques, and highly depends on the nature of the learning problem, data etc [28].

### 2.4 Steps in machine learning

#### 2.4.1 Problem and data analysis

Performing machine learning in practice is ideally done in an iterative fashion as shown in Figure 2.4. The first step consists of performing adequate problem and data analysis. Before using machine learning models, it is crucial to have a good understanding of the problem and data that is provided.

#### 2.4.2 Feature extraction

In the next step we are going to identify useful features which will be eventually used by our machine learning model. For instance, when it comes to signal processing and pattern recognition, useful features can be obtained when transforming signals from the time domain to frequency domain or by means of extracting \( k \)-th order statistics such as mean, standard deviation, maximum or minimum of a signal etc. However, the need for feature extraction highly depends on the nature of the used models, due to the fact
that some models (e.g. deep learning models like convolutional neural networks) perform feature extraction implicitly.

### 2.4.3 Select model and training approach

An appropriate model and training approach is then selected. This also highly depends on the nature of the data and problem description. Recall that one must always start with a simple model and iteratively increase model complexity when underfitting occurs. This to reduce the risk of overfitting (see Section 2.3). The training approach includes the choice of a cost function (i.e. error measurement). For instance, when it comes to classification tasks, one could use hinge loss (used in support vector machines) or cross-entropy loss, while in regression tasks the mean squared error function can be used.

### 2.4.4 Optimize model and evaluate

We finally optimize our model, in terms of used parameters, and evaluate the model on a given test set. This is required to gain insight in the generalization capability of our model (see Section 2.3). Based on these outcomes we can iteratively repeat from step one until decent performances are evaluated, hence the feedback loop in Figure 2.4.
2.5 Shallow versus deep learning

To conclude this chapter, we will discuss the difference between shallow and deep learning. It is important to mention that nowadays, many different definitions exist in literature. Let us look again at a simple classification problem, where we have a set of given training examples $X$ and training labels $t$ as

$$X = \{x_1, \ldots, x_n\}, \quad (2.6)$$
$$t = \{t_1, \ldots, t_n\}.$$  

We define a prediction model $f_\theta(x)$ with respect to some model parameters $\theta$. Using this prediction model and a given sample $x_n$, we obtain a prediction

$$y_n = f_\theta(x_n), \quad (2.7)$$

whereby we optimize $f_\theta$ in respect to $\theta$, so that $y_n \approx t_n$.

This can be seen as shallow learning, since we are dealing with one model $f_\theta$. However, it seems reasonable that we can define more “submodels” where each result of one model is taken as input for the next model in order to allow the existence of more complex models. Indeed, let’s define a function/model composition of $k$ submodels as

$$f_\varnothing := f_{\theta_k} \circ (\ldots (f_{\theta_2} \circ f_{\theta_1}), \quad (2.8)$$

such that for a given sample $x_n$ the composite model prediction yields

$$f_\varnothing(x_n) = f_{\theta_k} \circ (\ldots f_{\theta_2}(f_{\theta_1}(x_n))) = y_n. \quad (2.9)$$

This model would then be optimized or trained in respect to every $\theta_i$. For instance, every $f_{\theta_i}$ can be seen as a single neural network layer (see Section 3.2.1) where $f_\varnothing$ represents the fully neural network model. When $k$ in (2.8) is large, we are talking about deep learning. Deep learning models are more complex models in contrast to shallow learning models, which allows the capturing of “deep” information within data. Deep learning techniques are ideally used when a lot of data is available and when more complex (learning) tasks are be performed. The complex nature of the latter models, makes them more vulnerable to overfitting as described in Section 2.3. Regularization techniques are thus of great importance when it comes to deep learning models.

We will come back at specific deep learning models, in the next chapter, when we will talk about DNA read compression using deep learning.
Chapter 3

Deep learning models for DNA read compression

By giving an introduction in genome sequencing, genomic data and current compression techniques together with a brief introduction in machine learning techniques, we can now further focus on advanced machine learning models and the actual DNA read compression. Note that due to the complex nature of these models and the abundant data, we will further use the term deep learning rather than machine learning models. Before discussing the actual models, training approaches and final storage, we start by giving a recapitulation of the problem and given datasets that have been extensively used throughout this masters dissertation.

3.1 Problem and data analysis

In general, we have been working with four different files, each containing DNA reads of different lengths, originating from different species. Due to the sequencing technology and the way of storage, a file can contain a lot of redundancy across multiple reads. For instance, Figure 3.1 and 3.2 represent parts of two different files (i.e. DNA reads originating from E. coli and Homo sapiens respectively). It is not difficult to see that for the first file a lot of redundancy occurs, while in the other file less redundancy can be seen.

The used datasets, as shown in Table 3.1, contain different number of reads and bases per read. For each file, the reads are ordered on location within a genomic sequence, hence the potential redundancy across neighbouring reads.

As already explained in Section 1.4.1, most sequencing technologies are prone to uncertainties whereby an additional symbol $N$ is introduced in the alphabet $\mathcal{L} = \{A, C, G, T\}$. 

27
Figure 3.1: First ten DNA reads of the MiSeq_Ecoli_DH10B.fasta file (FASTA description lines omitted).

Figure 3.2: Ten DNA reads of the homosapiens_filtered.fasta file (FASTA description lines omitted).

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description (coverage)</th>
<th>#reads</th>
<th>#bases</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA12878_S1.fasta</td>
<td>Homo sapiens (26x)</td>
<td>69,430,054</td>
<td>101</td>
</tr>
<tr>
<td>homosapiens.fasta</td>
<td>Homo sapiens (2.3x)</td>
<td>56,463,236</td>
<td>100</td>
</tr>
<tr>
<td>K562.fasta</td>
<td>RNA data (16x)</td>
<td>246,476,391</td>
<td>76</td>
</tr>
<tr>
<td>MiSeq_Ecoli_DH10B.fasta</td>
<td>E. coli bacteria (44x)</td>
<td>13,175,550</td>
<td>150</td>
</tr>
</tbody>
</table>

Table 3.1: Four different datasets which were used for the training and testing of our deep learning models.

A simple compressed storage could be realised by using \(\lceil \log_2 |C| \rceil = 3\) bits for the storage of each base, instead of using eight bits (ASCII character encoding). This compression ratio can be seen as an upperbound for the average obtained bpb, for our deep learning models.

Further data analysis on both the datasets gave us a fairly equal distribution for \(A, C, T\) and \(G\):

\[
P_A \approx 0.25, \quad P_C \approx 0.25, \quad P_G \approx 0.25, \quad P_T \approx 0.25, \quad P_N \ll 0.01.
\]

The number of occurrences for \(N\) was very small for both sets.
Before training deep learning models, it is of great importance to define a data transformation which transforms the original data to a new representation which can be ideally exploited by the subsequent deep learning models. One could use for instance the following mappings

\[ A \rightarrow 0, C \rightarrow 1, G \rightarrow 2, T \rightarrow 3, N \rightarrow 4, \]  

so that numerical values are assigned to every base, which can then be used by mathematical models. However, by using this mapping we are transforming \( \mathcal{L} \) to a strict total ordered set \( \mathcal{L}_< \) where an ordering is defined between \( A, C, G, T \) and \( N \). For the most readers it is self-evident that no ordering exists between every symbol or base. We thus have to introduce a mapping which retains the categorical property of the extended DNA set. The most appropriate way to do this is by using a so called one-hot or one-of-K encoding, defined as follows

\[
\begin{align*}
A &\rightarrow [0, 0, 0, 0, 1], \\
C &\rightarrow [0, 0, 0, 1, 0], \\
G &\rightarrow [0, 0, 1, 0, 0], \\
T &\rightarrow [0, 1, 0, 0, 0], \\
N &\rightarrow [1, 0, 0, 0, 0],
\end{align*}
\]  

where a 1 is introduced in a column which stands for the appropriate base. The advantage of this technique is that by the use of this encoding, we also automatically generated a valid target set \( t \) (see Section 2.2.1) where each target \( t_i \) is a one-hot vector denoting the correct base or label. On the other hand, when storing these mappings in standard format (i.e. integers are used for representing 1 and 0), storage will increase due to the overhead that comes with these mappings.

A read \( r = \{b_1, ..., b_n\} \) with \( b_i \in \mathcal{L} \), consisting of \( n \) bases is transformed to a “relatively sparse” matrix \( \mathbf{R} \) with dimensions \( 5 \times n \). After transforming the genomic read data to a one-hot representation, we can finally use this data for our deep learning models. It is important to mention that no further data analysis or data preprocessing was done on our datasets, due to the limited number of symbols. We will also discuss some deep learning models where automatic feature extraction is performed.
3.2 Deep learning models

We will discuss the following potential candidate deep learning models:

1. Predictive Neural Network (PNN)
2. Flat Autoencoder (FAE)
3. Recurrent Neural Network (RNN)
4. Convolutional Autoencoder (CAE)

Before describing every model in detail, we will first start with an extensive but crucial study of artificial neural networks (ANNs). This because of the fact that every above-mentioned model, that has been used in this masters dissertation, is based on ANNs. In the next sections and chapter, we will extensively refer to some theoretical aspects which will be discussed in the next subsection.

3.2.1 Artificial Neural Networks (ANNs)

Inspired by biological neural networks (i.e. central nervous systems of animals), artificial neural networks form a family of models, used in machine learning and cognitive science. Similar to other machine learning models, neural networks have been used to solve a wide variety of tasks, like computer vision and speech recognition, that are hard to solve using ordinary rule-based programming.

As seen in central nervous systems or more specific the brain, neural networks mainly consist of interconnected neurons whose activations define recognizable linear pathways. Using axon terminals, connected via synapses to dendrites on other neurons, signal or message passing can occur between different neurons. When the sum of the input signals (in some neuron) exceeds a certain threshold, the neuron transmits an electrical signal along the axon. A simple representation with annotations of a biological neuron can be seen in Figure 3.3.
Feed-forward property

Similar to biological neural networks, artificial neural networks are also presented as fully connected (neural) layers, each consisting of an arbitrary number of artificial neurons. With fully connected we mean that every neuron is connected to every neuron in the next layer. For instance, let’s define a neural network with an input layer of $D$ inputs, followed by one layer, consisting of $M$ neurons and an output layer of $K$ outputs. We denote the input variables as

$$x_1, \ldots, x_D.$$  \hspace{1cm} \text{(3.4)}

To give a more general idea of the input, we could see the above input variables as all pixel values of some image $\mathbf{x} = \{x_1, \ldots, x_D\}$. We start with $M$ linear combinations of the input variables in the form

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)},$$  \hspace{1cm} \text{(3.5)}

$$j = 1, \ldots, M.$$
Figure 3.4: Diagram for the two-layer neural network as described in Section 3.2.1. The input, hidden and output variables are represented by nodes, while the weight parameters are shown as links between the nodes. Bias parameters are denoted by links coming from black nodes [8].

With the superscript \(^{(1)}\) we mean that the corresponding parameters are in the first layer of the network. In practice, we mostly talk about weights for \(w^{(1)}_{ji}\), and biases for \(w^{(1)}_{j0}\). The resulting linear combinations \(a_j\) are often referred to as activations.

These activations are then transformed by using a differentiable, nonlinear activation function \(h(.)\) to give

\[
z_j = h(a_j). \tag{3.6}
\]

In literature, \(z_j\) is also called hidden unit (i.e. the output of a neuron).

The outputs in (3.6) are then passed to the subsequent layer where we iteratively calculate

\[
a_k = \sum_{i=1}^{M} w^{(2)}_{ki} z_j + w^{(2)}_{k0}, \tag{3.7}
\]

\(k = 1, \ldots, K\),

with \(K\) the total number of outputs. We now calculated the transformation for the second layer of the artificial neural network. Finally, the output unit activations are transformed using a chosen activation function to give a set of network outputs \(y_k\). The corresponding network diagram can be seen in Figure 3.4. It is important to note that these functions (in the output layer) will mostly depend on the nature of the data (i.e. regression problem, classification problem etc.). For instance, the output unit activations could be transformed
by using a logistic sigmoid function

\[ y_k = \sigma(a_k), \quad k = 1, \ldots, K, \tag{3.8} \]

where

\[ \sigma(a) = \frac{1}{1 + e^{-a}}. \tag{3.9} \]

A plotted sigmoid function can be seen in Figure 3.5. It is easy to see that the range of this function yields correct probabilities. Within the domain of neural computing, sigmoid functions are also referred to as saturating nonlinearities. Saturating can be seen as a term to denote the squeezing of inputs. Indeed, from Figure 3.5 we know that sigmoid functions squeeze their inputs to the range \([0, 1]\). Important to mention is that one has to be careful when using a saturating nonlinearity as activation function, due to the fact that the derivative will tend to be very small in regions where the squeezing occurs. Consequently, we can say that for \(\sigma(a)\) the following property applies

\[ \frac{\partial \sigma}{\partial a} \to 0, \quad a \to \infty. \tag{3.10} \]

Later in this chapter, we will discuss the potential issues related to the use of saturating nonlinearities. A multiclass generalization of the logistic sigmoid function is called the **softmax function**. This function will transform a \(K\)-dimensional input \(y\) to a \(K\)-
dimensional vector, consisting of values in the range $[0, 1]$ and add up to one. If we take
the artificial neural network, depicted in Figure 3.4, we can for instance pass an input $x$ to
the network where the outputs $y$ are used as new input to a softmax activation function,
given by
\[
p(C_j | x) = \sigma(y)_j = \frac{e^{y_j}}{\sum_{k=1}^{K} e^{y_k}}. \tag{3.11}
\]
With $p(C_j | x)$, the posterior class probabilities (i.e. the probability of a class $C_j$ given a
sample $x$). Many nonlinearity functions can be used as activation function, as long as it
is continuously differentiable.
Due to the fact that message passing is done in a forward manner (i.e. layer to layer), we
sometimes use the term feed-forward neural networks. During this chapter we will also
talk about specific artificial neural networks, where the feed-forward property does not
yield due to directed cycles that are formed between hidden units. We call these models
recurrent neural networks [8].

**Network training**

We’ve introduced the mathematical concepts behind artificial neural networks, in terms
of the feed-forward message passing behaviour. However, we still need a way to train
these models in respect to every parameter.
Assume that we trained an artificial neural network $y(x_n, w)$ on a dataset $X = \{x_n\}$
with labels or target vectors $\{t_n\}$ defined in a continous space. We denote the term $w$
as all the parameters used by our model. Given our target vectors, we know that (see
Section 2.2.1) this translates to a regression problem. We want our model to fit the data
as accurate as possible and we minimze the sum-of-squares cost function given by
\[
E(y(x_n, w), t_n) = \frac{1}{2} \sum_{n=1}^{N} \| y(x_n, w) - t_n \|^2. \tag{3.12}
\]
For the sake of simplicity, we will further use the notation $E(w)$ for $E(y(x_n, w), t_n)$. In
the case that we are dealing with a classification problem where every target is a binary
class label $t_k \in \{0, 1\}$, the above-mentioned cost function would translate to
\[
E(w) = -\sum_{n=1}^{N} \{t_n \ln(y_n) + (1 - t_n) \ln(1 - y_n)\}, \tag{3.13}
\]
\[
y_n = y(x_n, w)
\]

which is also called the cross-entropy cost function. Or for multiclass classification problems

\[ E(w) = - \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \ln(y_k), \]  

(3.14)

where the class labels \( t_n = \{t_k\} \) are encoded in a one-hot fashion, similar as in (3.3), and the neural network model \( y(x_n, w) \) yields \( K \) outputs \( \{y_k\} \), by using the softmax activation function (3.11). The latter cost function would then be called the multiclass cross-entropy cost function.

Let’s continue with an arbitrary cost function \( E(w) \), defined for some model \( y(x_n, w) \). We want to optimize the cost, in respect to \( w \). The cost function can be seen as plotted in Figure 3.6. Given that \( E(w) \) is continuously differentiable, we know that its smallest value will occur at some point in the weight space such that the gradient of the cost function evaluates to

\[ \nabla E(w) = 0. \]  

(3.15)

Thereby we could make a small step in the “right” direction given by \(-\nabla E(w)\), making the error smaller. However, it is important to mention that, depending on the cost function, we have to keep in mind that we can get stuck in a local minimum (e.g. as seen in Figure 3.6 for point \( w_B \)). Seeking a global minimum is not always feasible and appropriate optimization techniques are necessary. For instance, one technique that is widely known

---

Figure 3.6: Cost function plotted as a hypersurface over the weight space. \( w_A \) denotes a global minimum, while \( w_B \) denotes a local minimum. At any point \( w_C \), the gradient is given by \( \nabla E \) [8].

35
today is gradient descent or steepest descent optimization, given by

$$w^{(τ+1)} = w^{(τ)} - η\nabla E(w^{(τ)}),$$

(3.16)

where the parameter $η > 0$ is known as the learning rate. After each update, the gradient is re-evaluated for the new weight vector and the latter process is repeated. Using a high learning rate can cause the optimization to easily escape from a local minimum, while using a smaller rate, we will mostly get stuck in a local minimum. A higher learning rate can be seen as taking bigger steps in the cost surface and smaller steps for lower learning rates respectively. In practice, it can be useful to use a combination of high and low rates. For instance, to train a neural network for some time, using a relatively high learning rate (i.e. initially try to escape from local minima) and eventually to decrease the learning rate in order to stay within a hopefully global minimum. Of course, this may depend on the nature of the data and cost function. We can also perform gradient descent in an on-line fashion. This is also known as sequential gradient descent or stochastic gradient descent. Updates to the weight vector are performed on one data point at a time. This update is repeated by cycling through the dataset sequential or by selecting random points with replacement. Finally, we can also use batch optimization where gradient descent is performed on a set of data points, rather than the full dataset. Generally, it is more efficient to work with mini-batch gradient descent since the calculation of $E(w)$ is time-consuming (i.e. the error has to be calculated for every input sample).

AdaGrad (adaptive gradient algorithm) is a modified optimization technique of (stochastic) gradient descent, which is widely used today [27] [29]. In general, the algorithm increases the learning rate for more sparse parameters and decreases for less sparsity respectively. This strategy improves convergence performance over standard stochastic gradient descent when data is relatively sparse and sparse parameters are more informative. In AdaGrad we define

$$G = \sum_{τ=1}^{t} g_τ g_τ^T,$$

(3.17)

where $g_τ = \nabla E(w^{(τ)})$ represents the gradient at iteration $τ$. The diagonal of $G$ is given by

$$G_{j,j} = \sum_{τ=1}^{t} g_{τ,j}^2.$$

(3.18)
This vector is updated at every iteration. Finally, the update function is now given as

$$w^{(\tau+1)} = w^{(\tau)} - \frac{\eta}{\sqrt{G_{j,j}}} g_r.$$  \hspace{1cm} (3.19)

Meantime, the attentive reader will have noticed that the calculations of $\nabla E(w)$ are not straightforward. An efficient way for evaluating the gradient of the cost function, for a feed-forward neural network, is by using a local message passing scheme in which error information is sent backwards through the network. We also call this technique error backpropagation. As we’ve already seen in Section 3.2.1, for a given feed-forward neural network, each unit computes a weighted sum of its inputs of the form

$$a_j = \sum_i w_{ji} z_i.$$  \hspace{1cm} (3.20)

Recall that $z_i$ is the activation of a unit that sends to unit $j$ and $w_{ji}$ is the weight associated with that particular connection between unit $i$ and $j$. Again, a nonlinear activation function $h(.)$ is used to find the activation $z_j$

$$z_j = h(a_j).$$  \hspace{1cm} (3.21)

In general a network will propagate an input vector $x_n$ throughout the network, where the latter process is called forward backpropagation. Now consider the error term $E_n$ associated with the latter input vector where

$$E(w) = \sum_{n=1}^N E_n(w).$$  \hspace{1cm} (3.22)

The derivative of $E_n$ with respect to a weight $w_{ji}$ is then given by

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}.$$  \hspace{1cm} (3.23)

The chain rule for partial derivatives is used because of the fact that $E_n$ depends on the weight $w_{ji}$ only via the summed input $a_j$ to unit $j$. From now on we introduce the following notation

$$\delta_j = \frac{\partial E_n}{\partial a_j},$$  \hspace{1cm} (3.24)

where $\delta$ is referred to as an error. Using (3.20) we then find

$$\frac{\partial a_j}{\partial w_{ji}} = z_i.$$  \hspace{1cm} (3.25)
and substituting (3.24) and (3.25) into (3.23), gives us

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i. \quad (3.26)$$

This means that the derivative in respect to $w_{ji}$ is simply obtained by multiplying the value of $\delta_j$ for the unit $j$ at the connection end (with weight $w_{ji}$) by the value of $z_i$ for the unit $i$ at the beginning of the connection. Thus in order to evaluate the derivatives, we only need to calculate the value of $\delta_j$ for each hidden and output unit in the network, and then use (3.26). In general, for each hidden unit the errors are defined (by using the chain rule for partial derivatives) as

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \quad (3.27)$$

and using the above findings we finally have

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k, \quad (3.28)$$

where the value of $\delta$ is obtained by propagating the $\delta$’s backwards from units higher up in the network, hence the name backpropagation.

The training of a neural network is now done using the following steps

1. Use an input sample $x_n$ and forward propagate through the fully neural network by using (3.20) and (3.21) to find the activations of all the hidden and output units.
2. Calculate $\delta_k$ for the output units (depends on the used cost function).
3. Backpropagate the $\delta$’s using (3.28) to obtain $\delta_j$ for each hidden unit.
4. Use (3.26) to evaluate the derivatives.

For batch methods, the derivative of the cost function or total error can then be obtained by repeating the above steps for every input vector in the batch and then summing over all the inputs. Summarized, the training of an artificial neural network is mainly done by passing the input data through the network, updating the parameters and iteratively repeat. A full pass of the data is also called an epoch. The evolution of training in terms of training and validation error can be visualized, for instance in Figure 2.2. The x-axis would then denote the number of epochs or the number of parameter updates.

After many parameter updates, the network will eventually start overfitting for which regularization techniques can be useful. A second problem that has to be addressed and goes by the name vanishing gradients, is when the calculated delta’s $\delta_j$ get too small and are passed to the next layer while backpropagating. The gradients (which are used
for updating all the weights) become significantly small, during propagation through the network, which in turns result in negligible updates of the weights within the first layers of the artificial neural network. Thus, the network will not be able to fully learn a relationship between inputs and outputs, yielding low performances. When using saturating nonlinearities as activation function in the whole network, one has to take in mind the latter issue due to the property (3.10) that we’ve similarly explained for sigmoid functions. This can be solved by using non-saturating nonlinearities such as rectified linear unit (ReLU) activation functions, which will not be further explained [8].

Regularization

It is generally known that deep neural networks (e.g. when using big networks in terms of hidden layers and units) are very sensitive to overfitting (see Section 2.3) due to the huge number of free parameters that can be used. For ANNs different regularization techniques exist (next to the general techniques in Section 2.3.3):

- **L₁, L₂ regularization**: by adding a specific penalty term to the cost function of an artificial neural network, large values of \( w \) are penalized. If one uses L₁ regularization, the cost function would evaluate as

\[
E(w) = \frac{1}{N} \sum_{n=1}^{N} E_n(w) + \lambda \|w\|_1,
\]

(3.29)

where \( \lambda \) is called the *penalty term* which controls the strength of the regularizer. Due to the \( \ell_1 \) norm, it can be shown that sparsity is induced for \( w \).

- **Dropout**: by disabling hidden units at random within layers, less “learning collaboration” will occur between neighbouring hidden units. Every artificial neuron will deliver its own important contribution to the learning process, rather than collaborative groups of neurons [22].

- **Data augmentation**: overfitting can be solved by adding more (training) data. However, in some cases this is infeasible. Data augmentation solves this by transforming samples on-line during training (e.g. rotating of input images in image classification in order that the network becomes rotation invariant) [44].

- Use smaller number of hidden units or layers (i.e. reduce the complexity of a model).

- **Early stopping**: when the model starts to overfit, after some training epochs, simply stop the training.
3.2.2 Predictive Neural Networks (PNNs)

The first proposed model for DNA read compression is called a predictive neural network (PNN). The purpose of a PNN is learning to predict the next read, given the previous read. Recalling the structure of each dataset (see Section 3.1), the rationale for PNNs become straightforward. Indeed, we can exploit the redundancy that exists between neighbouring reads. The model could learn that the next read will probably yield a similar structure. Important to know is that this requires knowledge about how and when reads are shifted or duplicated. Looking at Figure 3.1, we can for instance see that the read shown on line two is equal to the read on line one. While read five is left shifted, whereby an extra base $G$ is introduced, to form read six. Shifts or duplications are resulting from the sequencing technology and its underlying chemical processes. We therefore have no certainty of a possible underlying trend.

A possible architecture which represents a PNN is shown in Figure 3.7. As described in Section 3.1, every read is transformed in one-hot arrays of dimensions $RL \times 5$, where $RL$ denotes the read length and depends on the dataset. The training set $X$ is constructed in the following way

$$X = \{(x_1, t_1), \ldots, (x_N, t_N)\}$$
$$= \{(x_1, x_2), \ldots, (x_{N-1}, x_N)\},$$

where $x_i$ denotes the $i$-th read (one-hot encoded). The network is thus trained, by learning to map a given read to its consecutive read. As we’ve already discussed in Section 3.1, due to the categorical property of our targets $\{t_n\}$, the PNN models can be categorized

![Figure 3.7: Example architecture for a PNN network where the input layer is denoted by green and output layer by red. Hidden layers are shown in black, with given number of hidden units. Note that $RL$ stands for read length.](image-url)
under supervised learning, where classification is performed. The network has to predict RL bases, where each base can be seen as a categorical variable. We can thus use a softmax activation function (3.11) in the output layer, which returns five probabilities (i.e. normalized or summed up to one) for each base it has to predict. As cost function we thus use cross-entropy (3.13), for training the PNN models. For the obtained results and details about the used architectures, we refer to the next chapter (Section 4.2).

### 3.2.3 Flat Autoencoders (FAEs)

Flat autoencoders (autoassociators or Diabolo networks [53]) are special types of artificial neural networks, which are used for compression. It learns to transform inputs to a lower dimensional space, where the learned codewords are then used to reconstruct the original inputs [34][35]. Due to the fact that these models are trained to reconstruct their own inputs, autoencoders can be seen as unsupervised learning models and more specific for dimensionality reduction (Section 2.2.2). Autoencoders have already been succesfully used, for instance in image compression where better compression ratios where found in comparison to JPEG or other state-of-the-art compression techniques [5]. In order to prevent the autoencoders to learn the identity function, different variations exist which can be useful to force the autoencoder network to capture important information and learn better codewords:

1. **Denoising autoencoders**: as we’ve already seen in Section 2.3.3, one can add noise to the inputs in order to prevent overfitting [39].

2. **Sparse autoencoders**: by using $L_1$ regularization (3.29), we already know that sparsity is imposed on the hidden units within the network. This allows for sparse representations of inputs and to learn useful structures in the input data.

3. **Variational autoencoders (VAE)**: these models make strong assumptions concerning the distribution of latent variables (variables that are not observed and have to inferred). In order to allow latent representation learning, they use variational approaches [12].

Rather then predicting new reads, as seen in PNNs, we are going to transform each read into a smaller representation. It can be seen that these networks will have to gain insight into the structure of every read. This will require to have more understanding of the genomic “language”, rather than exploiting the redundancy across the reads, in order to achieve good compression. However, if the network learns to compress a certain read, we may expect that for consecutive (redundant) reads, the network would yield same compression accuracies. Notwithstanding, the models have to generalize well on unseen
Figure 3.8: Example architecture for a FAE network where the input layer is denoted by green and output layer by red. Hidden layers are shown in black, with given number of hidden units. Note that $RL$ stands for read length and $CL$ for the codeword length (number of hidden units in bottleneck layer).

data, so the question remains if a decent generalization can be achieved using FAEs.

A general architecture for FAEs can be seen in Figure 3.8. The hidden units in the bottleneck layer (blue) represent the codeword of an input sample. Training is done similar as in PNNs, where weights in the decoder part are the inversed weights of the encoder part. It is therefore only necessary to train the encoder part, in order to have all the hyperparameters of the model. Reads are transformed in one-hot arrays, similar as in Section 3.2.2, where the training set $X$ is constructed in the following way

$$X = \{(x_1, t_1), \ldots, (x_N, t_N)\}$$

$$= \{(x_1, x_1), \ldots, (x_N, x_N)\}. \quad (3.31)$$

Again, we use a softmax activation function (3.11) in the output layer and cross-entropy (3.13) as cost function. For the obtained results and details about the used architectures, we refer to the next chapter (Section 4.3).

### 3.2.4 Recurrent Neural Networks (RNNs)

Up to now, we’ve been working with non-sequential data and models. However, we could concatenate every read or construct the (partial) genomic sequence as described in Section 1.3.1. In sequential data, we assume that there is a relationship between subsequent samples or that its evolution can be modeled by a stationary process. While in non-sequential data every sample is assumed to be independent of each other. If the
genomic sequence is constructed using the reads, we have

\[ S = b_1 b_2 \ldots b_n, \]
\[ b_i \in \{A, C, G, T, N\}. \]  

(3.32)

Special types of neural networks can be applied, and more specific recurrent neural networks. These type of networks allow connections between units which form a directed cycle. The use of directed cycles allow to create an internal state of the network, in order to exhibit dynamic temporal behavior. In contrast to feed-forward networks, RNNs use the aforementioned internal memory to process arbitrary sequences of inputs and are able to capture long-term dependencies within sequences.

Different variations exist, such as:

1. Long short term memory (LSTM) networks: published by Hochreiter and Schmidhuber [47], these deep RNNs can learn “very deep learning” tasks that normally require memories of events that happened thousands or even millions of discrete time steps ago [31]. LSTM networks became very popular in the domain of natural language processing where they, unlike HMMs and other concepts, can learn to recognise context-sensitive languages [16].

2. Bi-directional RNNs: bi-directional RNNs or BRNNs use a finite sequence to predict or label each element of a sequence based on both the past and future context of the element [46].

3. Continuous-time RNNs: or CTRNNs are dynamical systems models, based on biological neural networks. These have been frequently used in the field of evolutionary robotics for tasks such as vision, co-operation and minimally cognitive behaviour [41].

RNNs could be used to predict a base \( s_i \), given a sequence of past bases \( s_{i-N} \ldots s_{i-1} \) where \( N \) is called the number of time steps. The network could then be unfolded in time which allows to train the network, similar as in feed-forward neural networks, with \( N \) hidden layers. Using LSTMs, the capturing of long-term dependencies (if enough time steps are considered) or important repetitive structures within a genomic sequence, is possible. We've chosen not to implement these models since the reconstruction of a genomic sequence would overlap with the second part of the masters dissertation (i.e. whole genome compression).
3.2.5 Convolutional Autoencoders (CAEs)

The last models that have been studied are called convolutional autoencoders. We will put a lot of focus on these models, due to their success in terms of DNA read compression (see next chapter, Section 4.4)

Convolutional neural networks

Inspired by the biological visual cortex, as seen in many organisms including humans, convolutional neural networks (or ConvNets, CNNs) are (deep) feature extraction models which are capable of visual pattern recognition, without any preprocessing, in data such as video or image [10]. Taking into account the visual cortex, being the most powerful visual processing system in existence, it seems reasonable to emulate its behaviour. CNNs try to achieve this behaviour by using hierarchical layers/building blocks:

1. Convolutional layers

In a convolutional layer, two-dimensional filters (or kernels) are discrete convolved, across the width and height of an input. For complex-valued functions $f$ and $g$, defined on $\mathbb{Z}$, the one-dimensional discrete convolution of $f$ and $g$ is defined as

$$(f * g)[n] = \sum_{u=-\infty}^{\infty} f[u]g[n - u].$$  \hspace{1cm} (3.33)
For $f, g$ defined on $\mathbb{Z}^2$, we find

$$(f \ast g)[m, n] = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f[u, v]g[m - u, n - v]. \quad (3.34)$$

Assume that we have an input $x$ with dimensions $X \times Y \times C$, where $X, Y$ are width, height and $C$ denotes the number of channels (e.g. $c_1$ is equivalent to the R channel of an RGB image). A filter, in a 2D convolutional layer, for feature map $k$ and channel $c$, is determined by $w^{kc}$ and $b^{kc}$. Note that every filter, that is convolved across the input, results in a feature map. A convolutional layer can include several feature maps (i.e. a filter bank) and are hyperparameters of a CNN. The obtained filter bank, for the above-mentioned input, is then given as

$$z^{(k)}[x, y] = h \left( \sum_{c=1}^{C} \sum_{x'=1}^{X} \sum_{y'=1}^{Y} w^{(kc)}[x', y']x[x - x', y - y', c] + b^{(kc)} \right), \quad (3.35)$$

with $h(.)$ a chosen activation function. The obtained equation (3.35) can also be extended for 3D convolutional layers (e.g. when video data is used as input). After calculating the activations $z^{(k)}$, we can then use this as new input for the next convolutional layer. The new input would then yield $X \times Y \times K$ dimensions, with $K$ channels.

In some cases, using convolutional layers result in less parameters, in contrast to fully connected dense layers (as seen in Section 3.2.1). The local connectivity property in CNNs (which are similar to receptive fields within the visual cortex) are depicted in Figure 3.10, while Figure 3.9 shows the fully connectivity in dense layers. The two hidden units in Figure 3.10 can be seen as two partial results of one filter (with some parameters) that is convolved across the input. For instance, when filters of size $10 \times 10$ for an input of $100 \times 100$ are used, one would need 100 parameters in contrast to 10000 parameters for a fully connected dense layer. Therefore, CNNs are ideally used for image and video data.

Following (3.35), we know that a kernel $\{w^{(kc)}, b^{(kc)}\}$ is slid over every pixel $x^{(c)}[x_i, y_j]$. This means that the convolution would yield an output with same dimensions $X \times Y$ as the input $x^{(c)}$. By defining a stride $(s_1, s_2)$, we can choose to make steps of $s_1$ in height $Y$ and $s_2$ in width $X$, rather than sliding over every pixel. The output for the latter strided convolution would then result in an output with $[\frac{X}{s_1}] \times [\frac{Y}{s_2}]$ dimensions. Although, this also depends on the padding (i.e. if kernels are fully overlapping with the input, while sliding) and size of the used kernels.
2. Pooling layers

In practice, every convolutional layer is completed by a pooling layer. In this layer the input is partitioned in “pooling” regions. For each region, one can choose to use a certain operation, for instance max-pooling. In the latter case, the maximum value is extracted from every region. Using pooling layers, the input will be reduced while the most important information is kept. For instance, max-pooling layers are used in image classification in order to ensure translation-invariance (i.e. translations of objects, appearing in different images, are removed after max-pooling).

3. Fully connected dense layers

After a sequence of convolutional-pooling layers, the convolutional neural network extracted a set of useful “deep” features which can then be passed to a fully connected dense part (ANNs), where for instance classification can be performed in the case of image classification. For the mathematical concepts and operations concerning these layers, we refer to Section 3.2.1.

Convolutional autoencoders

Using the above definitions and concepts, we can now define a convolutional autoencoder (CAE). These models are similar to FAEs, except for the fact that the input is first sent through a deep feature extraction part. The purpose is to exploit spatial redundancy across different reads and to find (deep) features by interpreting multiple reads as a three dimensional image, where each channel denotes a base plane. In Figure 3.11 the input is presented as an image, where every row represents a read. In general, the input would yield $H \times W \times 5$ dimensions, encoded in a one-hot format. Two conv-pool layers are then...
used for feature extraction, yielding much smaller “images”. And finally, these images are passed through a FAE network for actual compression. Within the FAE part, decoding is done in a similar fashion as described in Section 3.2.3. When leaving the FAE network, information will be sent through the inverse deep feature extraction network by using the following layers:

1. Unpooling layer 1
2. Deconvolutional layer 1
3. Unpooling layer 2
4. Deconvolutional layer 2

The unpooling operation is described in Figure 3.12. The deconvolutional operation is performed similar as in normal convolutional layers, except that the kernels are flipped horizontally and vertically by means of transposing. Finally, similar as in FAE, the DNA image or patch is reconstructed by using a softmax layer as output layer. We refer to Section 4.4, for the experimental results.

3.3 Training and testing

3.3.1 General strategy

When choosing an appropriate model, we also have to find a way to train, validate and test the model on the given types of datasets (see Section 3.1). It is important to have a good training and testing strategy in order to yield valuable models (i.e. in terms of generalization). Nevertheless, we also want to have an idea on how the trained models have to be used. For instance, we could train a deep learning model on a particular type of dataset in order that it generalizes well on new datasets, originating from the same organism or biological concept. Or we could extend our models by choosing to train in
such a way that it generalizes well on unseen data. In general, we use a training-validation strategy depicted in Figure 3.13. As we can see, for each file, we extract three different sets:

1. **Training set**: used for training and takes around 70% of the whole file.

2. **Gap set**: unused, in order to avoid redundancy across the training and validation set. 20% was chosen as factor (i.e. under the assumption that reads are ordered and redundancy exist across consecutive reads).

3. **Validation set**: the trained model is validated on this set, which takes about 20% of all the data within the file.

![Figure 3.13: How training and validation is done for a given deep learning model.](image)

![Figure 3.14: How final testing is done for a given deep learning model. Every row represents a training and testing iteration.](image)
The four obtained training sets (Figure 3.13) are then concatenated and used for training, whereafter the trained model is used for the validation on four different validation sets. Recalling Section 2.3, we know that the use of a test set is of great importance (i.e. in order to avoid overfitting on the validation set). To comply with the latter requirement, we use an additional testing strategy depicted in Figure 3.14. Using four iterations, a model is trained on three datasets and tested on the remaining dataset. It is easy to see that in this way, every dataset can be used as test set and each test set contains totally unseen samples.

### 3.3.2 Evaluation metrics

Using the above-mentioned strategy, we now define proper metrics which express how good our models are in predicting (PNNs) or reconstructing (FAEs and CAEs) reads.

**Cross-entropy loss**

In Section 3.2.1, we saw that the (multiclass) cross-entropy cost (or loss) function is ideally used as cost function for classification problems, with \( K \gg 2 \) classes. Let’s say that the model reconstructs a certain base, by the use of softmax (3.11), yielding the following normalized output

\[
o_i = [P_A, P_C, P_G, P_T, P_N],
\]

\[
= [0.1, 0.05, 0.05, 0.8, 0.0].
\]

And assume that the true label is defined as

\[
t_i = [0, 0, 0, 1, 0].
\]

Given (3.36), we know that the model successfully predicts T, with a decent probability of 0.8. However, let’s assume the output

\[
o_i = [0.2, 0.05, 0.05, 0.5, 0.2].
\]

Looking at the highest probability, the model successfully predicts the correct base. On the other hand, when it comes to probabilities, the model is not very accurate and using the cross-entropy cost function, would yield a higher (or worse) score. **Cross-entropy can thus be used to have an idea on how accurate the model predicts an output, in terms of probabilities.** Note that optimizing or training a model, using a misclassification cost function (i.e. correct prediction or not), is not
possible due to the fact that it’s not continuously differentiable. Therefore, we use the cross-entropy cost function for training our models.

**Reconstruction accuracy**

When speaking of compression, we are often not interested in how certain our model is but whether the model yields correct reconstructions or predictions. We thus also use a second evaluation metric, called the *reconstruction accuracy*. This metric is only used for evaluation, instead of training.

### 3.4 Storage

Now that we have discussed all the possible deep learning models, we only have to look at the actual storage itself (i.e. how to store compressed information). This chapter ends by discussing two different possibilities, depending on the model that is used. But first we will look at how lossless compression (i.e. predictions or reconstructions are prone to errors) can be achieved using the above-described deep learning models and residual storage. We refer to Section 4.5, for a proposed framework based on CAEs and entropy encoding.

#### 3.4.1 Residual storage

Notwithstanding the method that is used, a prediction (PNN) or reconstruction (FAE, CAE) will always introduce errors which have to be corrected and stored, in order to achieve lossless compression. Assume that we have an erroneous output \( o^\epsilon \) and correct output \( o \) (e.g. a prediction and label). In order to store the prediction together with the necessary correction we introduce the term *residue*. We can define two different ways of storing a residue:

1. **Residue with minimal symbols**

   Use a bitmap address where 0 is introduced on the position of a correct predicted or reconstructed base. A correction symbol \( s \in \{A,C,G,T,N\} \) is then introduced on every position which yields an error. For instance assume that

   \[
   \]
The residue $e_i$ for $o^i_j$ is then defined as

$$e_i = [0, C, 0, 0, 0, G, 0, G, 0, 0], \quad (3.40)$$

where the alphabet for every residue is

$$\mathcal{L} = \{A, C, G, T, N, 0\}. \quad (3.41)$$

2. Residue with extended symbols

We extend the alphabet by introducing an additional symbol so that

$$\mathcal{L} = \{A, C, G, T, N, 0, 1\}. \quad (3.42)$$

We now store a bitmap address where a 1 is introduced on every erroneous position. The bitmap address is then followed by the correction symbols. For the above example the residue would yield

$$e_i = [0, 1, 0, 0, 0, 1, 0, 1, 0, 0, C, G, G]. \quad (3.43)$$

Further compression of the residues

When a batch of residues is obtained, that is very sparse, we could further compress the batch by taking into account the distributions of every symbol in $\mathcal{L}$. Indeed, assume that we have $N$ residues with $|e_i| = 100$ and ten errors each. Further assuming that each error is made for some base $A$, the symbols are defined as $\mathcal{L} = \{A, 0\}$. We thus need $n = \lceil \log_2 |\mathcal{L}| \rceil = 1$ bit(s) to store each symbol. The effective number of bits per base, for the latter batch, would be equal to

$$N_{bpb} = 1. \quad (3.44)$$

However, when taking in account the distribution of each symbol, we could easily use entropy encoding (e.g. Huffman encoding in Section 1.4.3). For the latter batch, we find the following distributions:

$$P_0 = \frac{90N}{100N} = 0.9, \quad (3.45)$$
$$P_A = \frac{10N}{100N} = 0.1.$$
Using the Shannon entropy, given by

$$H = - \sum P_s \log_2(P_s),$$

(3.46)

which gives a lower bound on the number of bits required to encode each symbol $s$, we then find the following number of bits per base required to compress the batch

$$N_{bpb} = H = - (0.9 \log_2(0.9) + 0.1 \log_2(0.1)) = 0.47.$$  

(3.47)

When the reconstruction accuracy tends to be high, entropy encoding can thus be useful to compress the obtained residues. Nevertheless, it is important to mention that the latter entropy also depends on the distribution of the correction symbols.

### 3.4.2 Storage for PNNs

Following Section 3.2.2, we know that a PNN is able to predict a read $r_{i+1}$, given a previous read $r_i$. We use the following iterative storage strategy:

1. Store an uncompressed read $r_i$.
2. Use read $r_i$ to predict the next read $r_{i+1}$. This yields the residue $e_{i+1}$ which is stored after $r_i$.
3. Use $r_i$ and $e_{i+1}$ to obtain read $r_{i+1}$, and use this to predict $r_{i+2}$, which then yields $e_{i+2}$.
4. Repeat step 3.

After obtaining some residues $\{e_i\}$, we then use entropy encoding for final compression. To allow random access, one should use a block-based approach where every $n$-th read is used as starting point and where for every block $n - 1$ residues are stored using the latter strategy. We can conclude that PNNs, for DNA read compression, are less interesting due to the sequential approach of compression (and decompression). Especially when it comes to random access functionality.

### 3.4.3 Storage for FAEs and CAEs

While PNNs are used for the prediction of a subsequent read, FAEs and CAEs will be used for the compression and reconstruction of a read $r_i$. The difference between FAEs and CAEs is that reads are used, while for CAEs images/patches of DNA read data are used. Nevertheless, both are defined as compressing and reconstructing DNA read data.
We will explain the strategy for these models, by assuming that reads $r_i$ are compressed and reconstructed. We use the following strategy:

1. Transform a read $r_i$ into a smaller representation, which yields a codeword $c_i$, using the encoder.

2. Obtain the reconstructed read $r_i^r$ by using the codeword $c_i$ and the decoder.

3. Calculate the residue $e_i$ using $r_i^r$ and $r_i$.

4. Store the residue and codeword $[e_i, c_i]$.

5. Repeat steps 1–3 for the next read $r_{i+1}$.

We can compress the obtained residues $\{e_i\}$, using entropy encoding techniques. Very important to mention is that we also need the codewords $\{c_i\}$, in order to allow decompression. The obtained codewords are the hidden unit values of the bottleneck layer in the autoencoder (see Sections 3.2.3 and 3.2.5) and are stored as floating points. Compression will thus depend on the reconstruction accuracy, the number of hidden units in the bottleneck layer (i.e. codeword length) and the precision of the floating points.
Chapter 4

Results

In the previous chapter we discussed different machine/deep learning models for the compression of DNA read data, together with a theoretical view on the subsequent storage of the compressed data. In what follows, we will provide the reader with more practical results that have been obtained throughout our research. Initially, we discuss the used software and hardware, whereafter all the results are discussed for PNN, FAE and CAE. We then conclude with a final proposed framework for DNA read compression.

As seen in Section 3.2.5, more focus will be put on discussing CAE models, due to better performances in terms of compression. Note that not every model has been trained and tested, using the general strategy as described in Section 3.3.1, due to various shortcomings described in Section 3.2. Nevertheless, we recommend to use this general strategy for the comparison and assessment of different models.

4.1 Software and hardware specifications

4.1.1 Software

In terms of implementation, we used the following software:

- Python 2.7.6 (NumPy, SciPy etc.)
- Theano 0.8.0.dev-RELEASE
- Lasagne 0.2.dev1

Theano is a Python library that allows to define, optimize and evaluate mathematical expressions involving multi-dimensional arrays efficiently. This is done by using a tight
integration with NumPy, transparent use of a GPU, efficient symbolic differentiation, speed and stability optimizations and dynamic C code generation [2].

In order to allow efficient implementation of our machine learning models, we also used Lasagne. This is a lightweight library to build and train neural networks in Theano. Its main features consist of supporting various feed-forward networks such as CNNs and RNNs (or combinations), many optimization methods, freely definable cost functions (no need to derive gradients due to Theano’s symbolic differentiation) and of course a transparent support of CPUs and GPUs due to Theano’s expression compiler [1]. Other Python libraries where omitted. For the implementations we used a floating-point precision of 32 bits.

### 4.1.2 Hardware

Due to the huge amount of data and deep learning models, we used the Data Science Lab’s HPC cluster with the following hardware specifications:

- **RAM:** 131.92 GB
- **CPU:** (40 cores) Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz

Theano is focusing on computations at an abstract level, in order that the internal function compiler has a lot of flexibility about how to carry out those computations. Due to the parallel nature of graphics processing, theano allows carrying out calculations on graphics cards in order to allow faster processing. For instance, data-intensive calculations can be performed 140x faster, in comparison to the CPU. For allowing GPU calculations, one can use NVIDIA’s GPU-programming toolchain (CUDA) which has been integrated in Theano. However, since we had already access to a lot of computing power, we chose not to use CUDA.

### 4.2 PNN models

This model predicts the next read given a previous read, hence the name predictive neural network. We refer to Section 3.2.2 for the theoretical aspects.

#### 4.2.1 Architecture and (hyper)parameters

For this model, we’ve been testing with different number of hidden layers, each containing an arbitrary number of hidden units. For instance, in Figure 3.7, an architecture is shown for three hidden layers with 4096, 2048 and 1024 hidden units respectively. We use the following (hyper)parameters for our PNN models:
• **Regularization**: $L_1$
  
  - Penalty term $\lambda$: $1e - 5$

• **Optimization**: AdaGrad
  
  - Learning rate $\eta$: 0.01
  - $\epsilon$: $1e - 06$

• **Weight and bias initialization**: Xavier/Glorot initialization

• **Nonlinear activation function**: sigmoid function

• **Cost function**: cross-entropy loss function

• **Batch size**: 32

• **K-fold cross-validation**: 2-fold

• **Training set size**: 100,000 reads

• **Validation set size**: 10,000 reads

• **Gap set size**: 50,000 reads

The reader should understand all the above-mentioned concepts (Section 3.2.1). The penalty term, learning rate and $\epsilon$ are found by testing different values. We find that AdaGrad yields faster convergence in comparison to other optimization techniques such as Nesterov momentum. This can be explained by the fact that we are working with relatively sparse data which is, as seen in Section 3.2.1, ideally used in the latter case. We also use a slight $L_1$ regularization in the whole network, as it induces sparsity in the weight space (Section 3.2.1). However, this is not an improvement in comparison to no regularization or slight $L_2$ regularization, as it can be seen in Table 4.1. The reason behind this is that no compression is needed in PNNs, since these models need to learn or extract sufficient information to predict the next read. In Section 4.4, we will emphasize that $L_1$ regularization can be useful for CAEs, since these models need to learn valuable information from the inputs and reduce the amount of given information to a smaller representation. The introduction of sparsity in the weight space can thus be useful in the latter case.

<table>
<thead>
<tr>
<th>Regularization</th>
<th>Reconstruction accuracy((\mu))</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1$ regularization ($\lambda = 1e - 5$)</td>
<td>73%</td>
<td>34%</td>
</tr>
<tr>
<td>$L_2$ regularization ($\lambda = 1e - 5$)</td>
<td>74%</td>
<td>34%</td>
</tr>
<tr>
<td>None</td>
<td>73%</td>
<td>33%</td>
</tr>
</tbody>
</table>

Table 4.1: Results for PNN(1024) with different regularization. $L_2$ regularization yields better reconstruction accuracies.
4.2.2 Training and validation

We analyze the training-validation loss error and reconstruction accuracy. As we can see in Figure 4.1, the reconstruction accuracy converges to an estimated 73% while both training and validation loss error converge to 0.62. Interesting to see is that no overfitting occurs on the training set, since training does not influence the validation loss error (i.e. both loss errors tend to decrease and converge). This can be explained by the fact that $L_1$ regularization is used while training and evaluating on the training set, thus yielding a higher training loss error compared to the validation loss error. Analogously, we can say that the data is not memorized by the model. Further, we can see the influence of different architectures (in terms of number of hidden layers and units) depicted in Table 4.2. It seems that using more layers is not effective, even when we adapt the learning rate $\eta$ and penalty term $\lambda$. We refer to the next section for a more detailed explanation concerning this rather unusual phenomena. Using one hidden layer, starting from 1024 hidden units, yields better performances. We can also see high standard deviations for the obtained reconstruction accuracies for PNN(1024) and PNN(2048). The reconstruction accuracy was obtained by calculating the mean reconstruction accuracy over the validation set. We chose not to use an additional test set (which will also be the case for FAE), due to the less interesting properties for DNA read compression. However, we stress out the need for the latter sets, to assess generalization and overfitting of a deep learning model. Nevertheless, we used the general strategy (discussed in Section 3.3.1) where 70% of the E. coli dataset was used for training, 20% for the gap set and 10% for validation.

Figure 4.1: Learning curve and accuracy in function of epochs for a PNN model with one hidden layer and 1024 hidden units.
<table>
<thead>
<tr>
<th>Hidden layers [#units,]</th>
<th>Reconstruction accuracy(µ)</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[128]</td>
<td>54%</td>
<td>8%</td>
</tr>
<tr>
<td>[1024]</td>
<td>73%</td>
<td>34%</td>
</tr>
<tr>
<td>[2048]</td>
<td>73%</td>
<td>34%</td>
</tr>
<tr>
<td>[2048, 1024]</td>
<td>26%</td>
<td>3%</td>
</tr>
<tr>
<td>[4096, 2048, 1024]</td>
<td>25%</td>
<td>3%</td>
</tr>
</tbody>
</table>

Table 4.2: Results for PNN with different number of hidden layers. The use of more layers has a bad influence on the reconstruction accuracies.

The reader should notice the potentiality of PNNs for the compression of DNA read data. Indeed, a relatively high reconstruction accuracy, together with no need for storing codewords (Section 3.4) could make PNNs a worthwhile compression model. However, as we’ve already seen in Section 3.4.2, PNNs behave as sequential compression algorithms where random access functionality is difficult to achieve. Besides that, it is important to mention that PNNs yield less scalability. With the latter we mean that if would use PNNs to predict consecutive multiple reads (given multiple previous reads), the input and parameter space would increase tremendously due to the fully connectivity property in artificial neural networks. The use of convolutional layers and their local connectivity property could be a solution to the scalability problem (as seen in Section 3.2.5).

### 4.3 FAE models

As seen in Section 3.2.5, FAEs are used for the compression of single reads by the use of autoencoder networks (Figure 3.7). In these networks, the number of hidden layers in the encoder can be varied, together with the number of hidden units in the bottleneck layer (i.e. codeword length).

#### 4.3.1 Architecture and (hyper)parameters

We use the following (hyper)parameters:

- **Regularization:** $L_1$
  - Penalty term $\lambda$: 0

- **Optimization:** AdaGrad
  - Learning rate $\eta$: 0.01
  - $\epsilon$: $1e - 06$

- **Weight and bias initialization:** Xavier/Glorot initialization
• **Nonlinear activation function**: sigmoid function

• **Cost function**: cross-entropy loss function

• **Batch size**: 32

• **K-fold cross-validation**: 2-fold

• **Training set size**: 100,000 reads

• **Validation set size**: 10,000 reads

• **Gap set size**: 50,000 reads

We thus use the same fixed parameters as for PNNs, except that no regularization is used.

### 4.3.2 Training and validation

For different configurations of the encoder network, the obtained results are depicted in Table 4.3. Again, using more layers results in lower reconstruction accuracies. The latter observation could be explained due to the vanishing gradients issue, which occurs in (deep) artificial neural networks, combined with a saturating nonlinearity activation function (i.e. sigmoid function). We’ve already addressed this issue in Section 3.2.1. The gradients for each hidden layer and for the two architectures are depicted in Figure 4.2.

When analyzing the gradients for the first layer in the FAE(256-128-50) network, we can observe small values between \(-6 \times 10^{-7}\) and \(6 \times 10^{-7}\). The latter does clearly not hold for the subsequent layers. In fact, we can say that weights will adapt more in the last two layers, due to a higher magnitude of the gradients. This is exactly what vanishing gradients is all about: gradients decrease tremendously in magnitude, while backpropagating through the network, resulting in a network that is not able to generally learn. Hence the low reconstruction accuracies.

When further analyzing the gradients for the first layer in the FAE(90) network, we can observe higher gradients in contrast to the first layer of the previously mentioned network. In the case of a FAE(90) network, weights will adapt more thus yielding better reconstruction accuracies. Again, using a ReLU as activation function can be useful, however it is not recommended due to the inputs and outputs which are limited to \([0, 1]\).

We predict that the FAE model would need more training, together with a slower learning rate. Nevertheless, more research is advised and we refer to this as future work. It is clear that FAEs clearly underperform PNN networks. We can conclude that compression of single reads is thus hard to accomplish, due to the fact that the latter model is not able to extract useful information for reducing and reconstructing the input space. At this moment, the best reconstruction accuracy of 64% is obtained, using one hidden layer in
Table 4.3: Results for FAE with different number of hidden layers and units in the encoder part. Similar as for PNNs, using more layers is not recommended.

<table>
<thead>
<tr>
<th>Hidden layers [#units,]</th>
<th>Reconstruction accuracy($\mu$)</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>34%</td>
<td>4%</td>
</tr>
<tr>
<td>[90]</td>
<td>64%</td>
<td>3%</td>
</tr>
<tr>
<td>256, 128, 20</td>
<td>26%</td>
<td>3%</td>
</tr>
<tr>
<td>256, 128, 50</td>
<td>26%</td>
<td>4%</td>
</tr>
</tbody>
</table>

both encoder and decoder part with 90 hidden units each. However, important to mention is that using the latter amount of hidden units in the bottleneck layer, results in codewords which does not yield compression at all. Indeed, since 32 bits where used for floating-point precision, we know that the codeword would take 2880 bits. This exceeds the upperbound of 450 bits (150 times three bits for each base) for each E. coli read. Besides that, FAE suffers from the same scalability problem that we have described in Section 4.2, due to the fully connected dense layers. Recalling Section 3.1 and Figure 3.1, we suggest using autoencoders which exploit the redundancy across consecutive reads. For the latter, we propose convolutional autoencoders, which allow scalability and random access functionality.
4.4 CAE models

Up to now, we’ve discussed two different models together with their various shortcomings such as poor reconstruction accuracies, scalability issues (i.e. increasing the input size results in more model parameters) and limited random access functionality. To solve
the latter two, we already know that autoencoders together with convolutional layers are preferred. We will further study the average reconstruction accuracies and bits per base metric which has been obtained, taking in account the storage strategy in Section 3.4.3. We denote $B_e$ as the number of bits necessary to store the residue for a one-hot encoded read patch $P_{10}$, with dimensions $PS \times PS \times 5$, and $B_c$ as the number of bits to store the corresponding codeword. $B_e$ can be calculated using two different ways (as described in Section 3.4.1 and 3.4.1) together with entropy encoding, while $B_c$ is calculated as the number of hidden units in the bottleneck layer multiplied with the used floating-point precision. The resulting bpb for a patch $P_{10}$ is then defined as

$$\text{bpb}_P = \frac{B_e + B_c}{PS^2}. \quad (4.1)$$

When using batch training (with batch size $BS$) for CAEs, the latter will represent the average bpb over the whole batch.

### 4.4.1 Architecture and (hyper)parameters

We have been training and testing two architectures, allowing to compress 10x10 and 30x30 patches, where the bottleneck layers yield one hidden unit (i.e. codeword length of 32 bits) each.

1. **CAE(10x10,1) with input dimensions** $BS \times 5 \times 10 \times 10$
   1. **convolutional layer**: 256 feature maps with 3x3 filters, giving an output with dimensions $BS \times 256 \times 8 \times 8$
   2. **max-pooling layer**: 2x2 filters, giving an output with dimensions $BS \times 256 \times 4 \times 4$
   3. **dense layer**: 1024 hidden units
   4. **dense layer**: 128 hidden units
   5. **dense layer**: 1 hidden unit

2. **CAE(30x30,1) with input dimensions** $BS \times 5 \times 30 \times 30$
   1. **convolutional layer**: 64 feature maps with 3x3 filters, giving an output with dimensions $BS \times 64 \times 28 \times 28$
   2. **max-pooling layer**: 2x2 filters, giving an output with dimensions $BS \times 64 \times 14 \times 14$
   3. **convolutional layer**: 256 feature maps with 3x3 filters, giving an output with dimensions $BS \times 256 \times 12 \times 12$
4. **max-pooling layer**: 2x2 filters, giving an output with dimensions \(BS \times 256 \times 6 \times 6\)

5. **dense layer**: 1024 hidden units

6. **dense layer**: 128 hidden units

7. **dense layer**: 1 hidden unit

The above architectures only represent the encoder part. The decoder part is the inversed encoder, which can be obtained as described in Section 3.2.5. We omitted the decoder part, in order to maintain readability.

For both architectures we find the following optimal (hyper)parameters:

- **Regularization**: \(L_1\)
  - Penalty term \(\lambda\): \(1e^{-5}\)

- **Optimization**: AdaGrad
  - Learning rate \(\eta\): 0.01
  - \(\epsilon\): \(1e^{-06}\)

- **Weight and bias initialization**: Xavier/Glorot initialization

- **Nonlinear activation function**: sigmoid function

- **Cost function**: cross-entropy loss function

- **Batch size**: 32

- **Validation (2-fold cross-validation)**:
  - Training set size: 166,400 reads
  - Gap set size: 20%
  - Validation set size: 41,600 reads

- **Testing**:
  - Training set size: 187,200 reads
  - Test set size: 10,000 reads

Training and validation took around 17 hours for CAE(10x10,1) and 39 hours for CAE(30x30,1), where ten epochs were chosen. We motivate this number, by looking at the learning curves in Figure 4.3, 4.4, 4.5 and 4.6. After ten epochs, we can clearly see that the CAE model is already converging.
Again, we use sigmoid activation functions in order to allow hidden units to take values in the same range as for the inputs (i.e. recall one-hot encoding), together with AdaGrad for faster convergence. As it can be seen in Table 4.4, it seems that using a slight $L_1$ regularization in the whole network increases performance in terms of both mean reconstruction accuracies and bpb. For instance, looking at the Homo sapiens (high coverage) dataset, we can see an increase of 9% in the mean reconstruction accuracy when using $L_1$ regularization, in contrast to no regularization. Using $L_2$ regularization also yields better performances, but still underperforms $L_1$ regularization. The latter results thus confirm the effectiveness of introducing (slight) sparsity in the weight space.

<table>
<thead>
<tr>
<th>Regularization</th>
<th>Dataset</th>
<th>Reconstruction accuracy(µ)</th>
<th>bpb(µ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>E. coli</td>
<td>78%</td>
<td>1.14</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>82%</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>69%</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>83%</td>
<td>0.99</td>
</tr>
<tr>
<td>$L_1(λ = 1e-5)$</td>
<td>E. coli</td>
<td>89%</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>91%</td>
<td>0.61</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>80%</td>
<td>1.11</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>90%</td>
<td>0.63</td>
</tr>
<tr>
<td>$L_2(λ = 1e-5)$</td>
<td>E. coli</td>
<td>85%</td>
<td>0.90</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>86%</td>
<td>0.80</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>75%</td>
<td>1.31</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>90%</td>
<td>0.64</td>
</tr>
</tbody>
</table>

Table 4.4: Test results for different regularization parameters for CAE(30x30,1).

### 4.4.2 Training and validation

Plotting the training-validation loss errors together with the obtained accuracies in Figure 4.3, 4.4, 4.5 and 4.6, shows similar results as we’ve seen in PNNs. Both training and validation loss errors decrease and converge but no sign of overfitting is observed. Again, we can see a higher training loss error due to the regularization penalty term that is included. The obtained validation results for the four datasets are shown in Table 4.5. Nevertheless which model is used, we can see similar reconstruction accuracies, while the resulting mean bpb’s decrease with increasing complexity of architecture (i.e. in terms of number of convolutional layers and patch sizes). Using higher patch sizes, together with more convolutional layers, allows for more compression since inputs are mapped on the same size of codewords (i.e. 32 bits). One can thus further increase the patch sizes, together with the complexity of the CAE model to increase compression ratios (i.e. lower bpb’s). However, more data and training time (i.e. more training epochs) will be crucial.
An alternative to more complex models would be to further focus on the compression or encoding of the obtained codewords. Indeed, taking into account the definition of $bpb_P$ (4.1), we know that the required number of bits to store a codeword $B_c$ (for a corresponding patch $P_{10}$), takes a significant portion of the total required bits. We thus recommend to further compress the codewords, allowing for better compression ratios. For instance, a simple way could be to use quantization techniques where 32-bit codewords are constrained to 16 bits. Of course, the question remains to what extent we can further reduce the codeword sizes without decreasing the compression accuracy of a particular CAE model.

<table>
<thead>
<tr>
<th>Model</th>
<th>Dataset</th>
<th>Reconstruction accuracy ($\mu$)</th>
<th>bpb ($\mu$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAE(10x10,1)</td>
<td>E. coli</td>
<td>90%</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>90%</td>
<td>0.94</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>88%</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>92%</td>
<td>0.82</td>
</tr>
<tr>
<td>CAE(30x30,1)</td>
<td>E. coli</td>
<td>90%</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>90%</td>
<td>0.64</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>88%</td>
<td>0.74</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>89%</td>
<td>0.67</td>
</tr>
</tbody>
</table>

Table 4.5: Validation results for different CAE models (patch size, hidden units).

### 4.4.3 Testing

Using our final testing strategy, we can observe (see Table 4.6) that the CAE models yield similar results as we’ve seen for validation. It is clear that both models achieve best compression (e.g. 0.61 bpb with CAE(30x30,1)), on the Homo sapiens dataset, where a lot of redundancy is seen. **This is quite remarkable, since the latter results prove that CAEs can successfully compress DNA reads originating from an organism or dataset, even when they were not used in training.** CAEs have thus been proven to highly generalize on new datasets, even when they are originating from different species. When comparing the obtained results for the Homo sapiens (low coverage) dataset with other state-of-the-art compression techniques, we can see that our models yield better compression ratios in contrast to 7-zip and no-CABAC.

The (normalized) error distributions, for each base and dataset, are shown in Figure 4.7 and 4.8. For validation, it seems that CAEs are more prone to 'A' nucleotides while this is not the case for 'N'. However, we can see different error distributions if testing is performed, which shows that CAEs are not directly biased to a subset nucleotides. Based on these outcomes, we recommend to use the residual storage technique described in Section 3.4.1, by limiting the number of symbols in $\mathcal{L}$.
<table>
<thead>
<tr>
<th>Model</th>
<th>Dataset</th>
<th>Reconstruction accuracy(μ)</th>
<th>bpb(μ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAE(10x10,1)</td>
<td>E. coli</td>
<td>89%</td>
<td>1.01</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>91%</td>
<td>0.87</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>82%</td>
<td>1.29</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>89%</td>
<td>0.94</td>
</tr>
<tr>
<td>CAE(30x30,1)</td>
<td>E. coli</td>
<td>89%</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>91%</td>
<td>0.61</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>80%</td>
<td>1.11</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>90%</td>
<td>0.63</td>
</tr>
<tr>
<td>CABAC*</td>
<td>E. coli</td>
<td>-</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>-</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>-</td>
<td>1.09</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>-</td>
<td>0.12</td>
</tr>
<tr>
<td>no CABAC*</td>
<td>E. coli</td>
<td>-</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>-</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>-</td>
<td>1.67</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>-</td>
<td>0.42</td>
</tr>
<tr>
<td>7-zip Ultra*</td>
<td>E. coli</td>
<td>-</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>HS (high)</td>
<td>-</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td>HS (low)</td>
<td>-</td>
<td>1.13</td>
</tr>
<tr>
<td></td>
<td>RNA</td>
<td>-</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Table 4.6: Test results for different CAE models (patch size, hidden units) and other state-of-the-art compression techniques.

*Obtained bpb’s merely represent the lowest seen bpb’s, during compression. This means that the mean values will be higher for a given dataset.

### 4.4.4 Memory usage and time complexity

We conclude with a brief overview of the memory consumptions and time complexities, that we’ve encountered during training and testing of CAEs. Looking at Table 4.7, shows us the high memory requirement that comes with CAE models. This is not entirely surprising, since we are using one-hot encoded data. We point out the need for further memory optimization if one needs more training data and thus more powerful models in terms of data compression. We provide the reader with a potential optimization strategy, to solve the latter issue.

Assuming a dataset which contains $N$ bases, we know that storing the dataset in ASCII format yields $8N$ bits. However, when converting to a one-hot encoded format, the latter dataset would yield $160N$ bits (if we assume 32-bit integers). Since the resulting dataset can be seen as binary data (due to the one-hot encoding), we can easily store the dataset in binary format, yielding only $5N$ bits. We can finally train our models, by using on-line mini-batch training (see data augmentation in Section 3.2.1). When a binary batch is extracted from the dataset, we simply retransform the batch to the appropriate one-hot encoded format. Using this strategy allows a memory optimization depicted in Table 4.8.
<table>
<thead>
<tr>
<th>Model</th>
<th>Memory (MB)</th>
<th>Training (hrs)</th>
<th>Testing (mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAE(10x10,1)</td>
<td>454.4</td>
<td>4.16</td>
<td>6.25</td>
</tr>
<tr>
<td>CAE(30x30,1)</td>
<td>4089.6</td>
<td>9.72</td>
<td>14.58</td>
</tr>
</tbody>
</table>

Table 4.7: Memory usage (data) and running times for training and testing two CAE models. We used 187000 samples for training and 10000 for testing.

<table>
<thead>
<tr>
<th>Model</th>
<th>Memory usage (MB) (non-optimized)</th>
<th>Memory usage (MB) (optimized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAE(10x10,1)</td>
<td>454.4</td>
<td>14.2</td>
</tr>
<tr>
<td>CAE(30x30,1)</td>
<td>4089.6</td>
<td>127.8</td>
</tr>
</tbody>
</table>

Table 4.8: Memory usage (data) prior to and after memory optimization.
Figure 4.3: Learning curve and accuracy in function of epochs for a CAE model (30 × 30 patches and 32-bits codewords). Validated on the E. coli dataset.

Figure 4.4: Learning curve and accuracy in function of epochs for a CAE model (30 × 30 patches and 32-bits codewords). Validated on the Homo sapiens (high coverage) dataset.
Figure 4.5: Learning curve and accuracy in function of epochs for a CAE model (30 × 30 patches and 32-bits codewords). Validated on the Homo sapiens (low coverage) dataset.

Figure 4.6: Learning curve and accuracy in function of epochs for a CAE model (30 × 30 patches and 32-bits codewords). Validated on the RNA dataset.
Figure 4.7: Error distribution (normalized) for each base and file, obtained after validation.

Figure 4.8: Error distribution (normalized) for each base and file, obtained after testing.
4.5 Proposed novel framework for DNA read compression

In the previous section, it has been proven that CAEs are useful in terms of read compression and can thus be further used in a more general framework. We propose a novel abstract compression framework, depicted in Figure 4.9 (i.e. encoder) and Figure 4.10 (i.e. decoder), which will be briefly explained in the remaining chapter. The included components should be clear to the reader, as it has already been fully explained in Section 3.2.5 and 3.4.

4.5.1 Encoder

DNA reads $X$ are fed to the *one-hot encoder* which transforms the original data to an appropriate one-hot binary encoded format $X_{10}$. We continue extracting DNA images or patches, given a patch size (e.g. $30 \times 30$), which yields $P_{10}$. The latter information is then given as input to the *CAE encoder* and the *residue transform*. After compressing the patch $P_{10}$, yielding a codeword $c$, we then use the *CAE decoder* for obtaining the reconstructed patch $\tilde{P}_{10}$. The original patch $P_{10}$ and reconstruction $\tilde{P}_{10}$ are then used for calculating the residue $e$, whereafter further compression is realized using the *entropy encoder*. After assembling the final compressed residue $\hat{e}$ and codeword $\hat{c}$, we finally obtain the compressed input. The resulting encoder is illustrated in Figure 4.9. In practice, the *patch transform* will output a batch of patches, due to the fact that CAEs were trained using mini-batch rather than single patch inputs.

4.5.2 Decoder

After information is stored or transmitted, one can reconstruct the original data by using the decoder, illustrated in Figure 4.10. Conversely, we obtain the fully reconstructed residue $\hat{e}$ and codeword $\hat{c}$, by using the *codeword decoder* and *entropy decoder*. The *CAE decoder* then reconstructs the patch $\tilde{P}_{10}$, prior to the calculation of the original patch $\tilde{P}_{10}$ using $\tilde{P}_{10}$ and $\hat{e}$ as input for the *residue transform*. Decoding is finally obtained by converting the latter output to the original DNA read image $X$ by using the *patch transform* and *one-hot decoder* subsequently.
4.5.3 Complexity

When it comes to the time complexity for encoding, we assume complexity $O(n)$ for the one-hot encoder and patch transform. In general, a bottleneck can be assumed for the CAE encoder/decoder. After training the network, the calculation of codewords (using the encoder) and reconstructions (using the decoder) will depend on:

1. Number of samples/batches: each sample or batch has to be fed to the network.

2. Structure and number of layers in the encoder/decoder: the number of multiplications needed to compute the activation of all neurons. The latter multiplications will depend on the number of parameters, used in the network.

Again, the complexity for the residue transform and codeword encoder is defined as linear time $O(n)$. We conclude with the entropy encoder by assuming Huffman encoding. When the input probabilities are sorted, the complexity of constructing is linear $O(n)$. One can also choose for a more greedy approach, where the latter complexity evaluates to $O(n \log(n))$ [11]. However, since $n$ depends on the number of symbols, we can neglect the construction complexity, due to the small alphabet $\mathcal{L}$. We can analogously find the above-mentioned complexities for the decoder.
Figure 4.9: Architecture for the encoding part of the proposed CAE compression framework.
Figure 4.10: Architecture for the decoding part of the proposed CAE compression framework.
Chapter 5

Conclusions and future work

5.1 Conclusions

The goal in this masters dissertation was to study whether a transition is feasible from various state of the art compression techniques towards a machine learning based approach for (non-referenced based) DNA read compression. We started this book with expressing the need for highly effective compression techniques, to reduce the enormous data storage and bandwidth needs, when it comes to DNA read data. Besides that, as seen in Chapter 1, there is an urgent need for proper benchmarks in order to allow extensive analysis and comparison of different compression techniques. We have studied different machine learning and deep learning models such as predictive neural networks (PNNs), recurrent neural networks (RNNs), flat autoencoders (FAEs) and convolutional autoencoders (CAEs).

We stated that RNNs are more useful for sequence compression rather than DNA read compression, due to the non-sequential nature of DNA read data. However, the (partial) reconstruction of the reference genome sequence (using its DNA reads) offers the possibility to use the latter model. Under the assumption that reads are ordered, a sequential approach of compression can be achieved by using PNNs. This is done by predicting a read, given the previous read. We’ve shown that PNNs are less interesting due to the fact that random access is limited. Using this observation, we moved to FAEs which generally solved the random access issue in PNNs. However, this comes with an additional storage cost, by means of storing a codeword next to the correction term (i.e residue). Besides the overhead in storage, we also encountered low compression rates since redundancy across consecutive reads are not exploited in FAEs. This motivated us to introduce CAEs, allowing for:

- **High compression**: due to the spatial feature extraction in convolutional layers, CAEs allow to exploit redundancy across consecutive reads.
• **Input scalability:** scaling (i.e. increasing input) is done more efficiently due to the local connectivity property within convolutional layers, in contrast to fully connected dense layers.

• **Random access:** due to the nature of autoencoders, non-sequential based compression (in contrast to PNNs) is allowed, therefore allowing for random access.

It has been proven that CAEs yield remarkable results, due to the fact that they are able to successfully compress DNA reads, originating from an organism or dataset, even when it’s not used in training. Consequently, CAEs have thus been proven to highly generalize on new datasets, irrespective of whether they are originating from different or same species. For the low redundant Homo sapiens dataset, we obtained better bpb’s in contrast to other state-of-the-art techniques.

Using CAEs, we finally introduced a novel candidate framework for DNA read compression, consisting of an encoder and decoder part. We emphasized that an appropriate encoding of residues and codewords can further increase compression ratios.

### 5.2 Future work

We will finish this book by providing the reader with the most important future work:

• It is clear that an overhead, in storing codewords for CAEs, still has to be solved. As already mentioned, we could further **decrease memory usage for the obtained codewords** by means of quantization or floating-point compression techniques.

• Although a lot of data was provided for training deep learning models, we were bounded to memory restrictions. Indeed, converting the original DNA read data to a one-hot encoded format comes with a high memory requirement. A possible solution that was given in Section 4.4.4, could be to save the latter information in binary format, together with conducting **more general (memory) optimizations**.

• Consequently, this would allow **further analysis and improvements**, in terms of training and testing deep learning models.

• We are confident that, contingent upon extensive memory optimization, more powerful compression can be achieved when **increasing input or patchsizes together with deeper CAE models**.
Bibliography


77


[38] National Human Genome Research Institute. Cost per genome. 2012.


