Master's thesis
Master's Programme in Data Science

Transformer Networks in Gene Prediction

Venla Viljamaa

February 14, 2022

Supervisor(s): Dr. Jarno Alanko, Professor Veli Mäkinen

Examiner(s): Dr. Jarno Alanko
Professor Veli Mäkinen

University of Helsinki
Faculty of Science

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Degree programme
Faculty of Science	Master's Programme in Data Science

Tekijä — Författare — Author

Venla Viljamaa

Työn nimi — Arbetets titel — Title

Transformer Networks in Gene Prediction

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidantal — Number of pages
Master's thesis	February 14, 2022	63

Tiivistelmä — Referat — Abstract

In bioinformatics, new genomes are sequenced at an increasing rate. To utilize this data in various bioinformatics problems, it must be annotated first. Genome annotation is a computational problem that has traditionally been approached by using statistical methods such as the Hidden Markov model (HMM). However, implementing these methods is often time-consuming and requires domain knowledge. Neural network-based approaches have also been developed for the task, but they typically require a large amount of pre-labeled data.

Genomes and natural language share many properties, not least the fact that they both consist of letters. Genomes also have their own grammar, semantics, and context-based meanings, just like phrases in the natural language. These similarities give motivation to the use of Natural language processing (NLP) techniques in genome annotation.

In recent years, pre-trained Transformer neural networks have been widely used in NLP. This thesis shows that due to the linguistic properties of genomic data, Transformer network architecture is also suitable for gene predicting. The model used in the experiments, DNABERT, is pre-trained using the full human genome. Using task-specific labeled data sets, the model is then trained to classify DNA sequences into genes and non-genes. The main fine-tuning dataset is the genome of the *Escherichia coli* bacterium, but preliminary experiments are also performed on human chromosome data.

The fine-tuned models are evaluated for accuracy, F1-score and Matthews correlation coefficient (MCC). A customized estimation method is developed, in which the predictions are compared to ground-truth labels at the nucleotide level. Based on that, the best models achieve a 90.15% accuracy and an MCC value of 0.4683 using the *Escherichia coli* dataset. The model correctly classifies even the minority label, and the execution times are measured in minutes rather than hours. These suggest that the NLP-based Transformer network is a powerful tool for learning the characteristics of gene and non-gene sequences.

ACM Computing Classification System (CCS):
Applied computing → Life and medical sciences → Computational biology → Computational genomics
Computing methodologies → Machine learning → Machine learning approaches → Neural networks

Avainsanat — Nyckelord — Keywords

Transformer, Deep learning, DNA, Genome, Escherichia coli, Classification, DNABERT, HMM

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction **1**

2 Background **3**

2.1 Basics of Genomics . . . 3

2.2 Semantics of DNA . . . 4

2.3 Related work . . . 6

3 Preliminaries **9**

3.1 Notations . . . 9

3.2 Transformer architecture . . . 10

3.2.1 Preprocessing . . . 10

3.2.2 Attention . . . 11

3.2.3 Feed Forward network . . . 12

3.2.4 Encoder and Decoder . . . 14

3.2.5 Training . . . 16

3.3 BERT and other Transformers . . . 18

3.4 DNABERT . . . 20

4 Experiments **23**

4.1 Collecting data . . . 23

4.1.1 *Escherichia coli* K-12 . . . 23

4.1.2 Human chromosome 18 . . . 24

4.1.3 Preprocessing . . . 25

4.1.4 Dataset variations . . . 26

4.2 Fine-tuning the DNABERT . . . 28

4.3 Evaluation setup . . . 29

4.4 Results . . . 32

4.4.1 Primary results . . . 33

4.4.2 Secondary results . . . 41

5 Conclusions and discussion **43**
5.1 Summary 43
5.2 Discussion 44
5.3 Future work 46

Bibliography **47**

Appendix A Fine-tuning **51**

Appendix B Preprocessing and evaluation scripts **53**

1. Introduction

The first complete genome was sequenced in 1995 [11], and the publishing speed of new genomes of bacteria, plants, animals and other organisms has accelerated since then [33]. Genomic data consists of alphabetical characters that denote the nucleotides. The order of those nucleotides forms the genome, and the gene sequences inside the genome contain information about different features of the organism. Therefore, taking advantage of all the published raw genomics data requires annotating those coding sections and their functions, which is a rather complicated task [22]. In addition to traditional methods based on rules and statistics, neural networks are a potential tool for predicting genes. Compared to statistics-based systems, they require less domain knowledge and are therefore easier to implement.

In this thesis, we investigate the possibility of utilizing a Transformer neural network in a gene detecting task. In addition to being a powerful tool in Natural language processing (NLP) [7, 34], Transformer networks can also help in detecting genes and other coding sections from genomes. The linguistic characteristics of genome data provide particular support for this approach. More specifically, this thesis focuses on the use of a Transformer-based pre-trained DNABERT model [18] in detecting genes of the *Escherichia coli* (later *E. coli*) bacterium [30]. We also performe preliminary experiments on the human genome [12] using the 18th chromosome.

In our experiments, the network receives labeled genome sequences as input, with labels 1 or 0 indicating whether the sequence is part of a gene or not, respectively. From these sequences and labels, the model is supposed to learn the characteristics of genes and make predictions for previously unseen data sequences. Due to a novel approach to this task, we develop a new evaluation method that allows comparison at the nucleotide level. Therefore, it is challenging to compare our solution directly to existing systems. We still make a preliminary comparison to the ability of GeneMarkS [3] to predict *E. coli* genes. In addition, the evaluation metrics are compared to the so-called naive baseline model, where the most likely label is assigned to all nucleotide sequences.

The model performance is measured using various metrics, like accuracy and Matthews correlation coefficient (MCC) [23]. The latter is especially suitable for our data with highly unbalanced label distribution [5]. Our best models achieve better

metrics than the naive baseline model, indicating that the Transformer network can capture some of the underlying semantics of the genome. Therefore, they are a promising tool to help predict genes from bacterial genomes. However, more work is needed to bring Transformer neural networks to the level of GeneMarkS in gene prediction accuracy.

This thesis begins with the Background chapter, where relevant basics and semantics of genomics are presented, as well as related work in terms of the gene detecting problem. The next chapter, Preliminaries, explains the architecture and mechanisms of Attention-based Transformer neural networks in detail. Also some state-of-the-art Transformer models, including the model used in our experiments, DNABERT [18], are introduced. The procedure from data collection to model fine-tuning is explained in the Experiments chapter, as well as the results from different models trained. Finally, the Conclusions chapter covers the summary of those results and a discussion of the data and methods used. The thesis ends with suggestions for future work.

2. Background

This chapter provides relevant background information for the research methods and experiments of this thesis. The reader gets the required terms and explanations of genomics, as well as an overview of similarities between natural language and genome data. Finally some of the previous works related to the gene predicting task are covered.

2.1 Basics of Genomics

This thesis examines the Transformer neural network's ability to understand the underlying semantics of genomes from the data itself, without needing input from genomics experts. Therefore it is not crucial to emphasize genomics knowledge, but a basic understanding of genomics is sufficient to understand the composition of the genome data used in this study. This section gives an introductory overview of basic genomics as a recap of high school biology courses. The terminology of genomics used throughout this work will also be introduced.

Living organisms can be divided into three domains, bacteria, archaea and eukarya [13]. Bacteria and archaea consist of one cell with no nucleus and are together referred to as *prokaryotes*. All other organisms, such as plants, animals and fungi, belong to the *eukaryotes*. Their cellular structure is more complex containing a membrane-bounded nucleus. All organisms have their DNA in the cell structure which carries the genetic information of the organism [6]. The double-stranded structure of DNA consists of a sequence of nucleotides introduced next.

A *nucleotide* is a chemical structure with three parts: sugar, phosphate and a nitrogenous base [6]. The nitrogenous base is the most relevant in terms of genomics data processing since it determines the marker that refers to the nucleotide. Four nitrogenous bases – adenine, thymine, guanine and cytosine, and their corresponding letters A, T, G and C – form the alphabet of the data used in bioinformatics. The double-strand structure of DNA, illustrated in Figure 2.1, comes from nucleotides forming pairs and connecting. Adenines are connected with thymines, and guanines pair with cytosines. Nucleotides form *codons* in fragments of three consecutive nucleotides, which in turn define amino acids and protein biosynthesis, the core process inside the

cells.

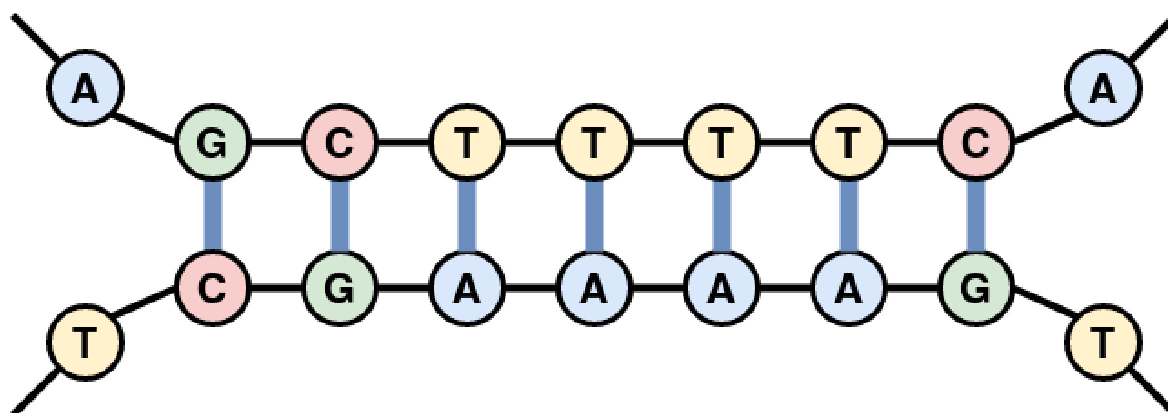


Figure 2.1: An example of a DNA strand showing how the nucleotides are connected.

The complete set of an organism's DNA is called a *genome*, and for prokaryotes, the DNA inside the cell is the same as its genome. The genome of eukaryotes is a combination of the DNA from both the nucleus and mitochondria. Most of the genome is identical between different individuals in the same species, but genetic diversity does occur. Genomes can be read in both directions, forward and reverse.

A *gene* is a nucleotide sequence of the genome. The order of nucleotides works as the source code, or recipe, for the proteins and other molecules. There are typical codons that indicate the beginning or end of a gene, *start* and *end codons*. The codons inside a gene define the functions of the gene through protein biosynthesis. A bacterial gene is usually a contiguous nucleotide sequence that begins at one nucleotide and ends at another.

Eukaryotes have more complicated structures in their genes than prokaryotes. One gene consists of multiple *exons*, the coding sequences of a gene, and *intron* sequences between them. Exon sequences folded together in a particular order form the nucleotide sequence that defines the gene. As mentioned above, the genome has two reading directions, and therefore the orientation of genes can also be forward or backward. The genes can also overlap with each other, meaning that one gene can start from the middle of another, or even be entirely inside another [25].

2.2 Semantics of DNA

One inspiration for this thesis was the interest in linguistics and the possibility to use NLP techniques with genome data. Both language and DNA consist of letters, and therefore it should be technically possible and effortless to configure NLP systems to take genome data as an input. In this section, we will introduce the field of NLP and examine similarities and differences between natural language and DNA.

In NLP, computers are taught to understand and generate both spoken and written language. These tasks are easy for humans, but require a lot of training for a model to capture all the nuances of the language [19]. For instance, from the sentence ”The chicken crossed the street slowly because it was so wide” it is clear to the reader, that "wide" refers to the street, but in theory, it could as well refer to the chicken. Therefore, the meaning of the sentence is not trivial for the algorithm or NLP model. Often in NLP, the processed text is divided into sentences or words, and the connections between those pieces are formed, for instance, with rule-based systems or a neural network learning the connections. That way the system gathers an understanding of the text and is able to execute the given tasks, for example, telling whether the sentence has a positive or negative sentiment.

Language is widely used as a metaphor for DNA, but there are also concrete similarities between natural language and so-called cell language, which is based on DNA. This cell language has linguistic features like alphabet, lexicon, sentences, grammar, phonetics, semantics and articulation [17].

Genome data consist of only four nucleotides A, T, G and C, meaning that the size of the DNA alphabet is only 4. If we think of codons mentioned in Section 2.1, we can equate them into words, and therefore the size of the lexicon is $4^3 = 64$. Most NLP applications handle much larger alphabets and vocabularies.

It is also possible to treat genes as words, and that approach makes the task instantly more complicated. Our example genome of the *E. coli* bacterium has genes of length between 14 to 7077 nucleotides. This is a much wider range of lengths than lexicons in normal NLP applications. Instead, equating genes to sentences in natural language makes the task more comparative to NLP tasks, such as labeling sentences as positive or negative depending on their content.

The DNA of organisms evolves through time. Small incidental changes, also known as mutations, occur constantly. Therefore genome data is more complicated than strictly the size of its alphabet or lexicon. Certain rules that locate coding sequences can be used to define the grammar of DNA. However, these rules tend to have many exceptions rendering the resultant grammar very complicated. As mentioned earlier, this possibility of interpretation is also typical of natural language.

Transformer networks have been popular in NLP applications for recent years. Their attention mechanism, explained in Section 3.2.2, is said to capture the semantics of the text by emphasizing important parts and connections in the text [34]. In our experiments, we use the Transformer-based DNABERT model that has been pre-trained with the human genome. DNABERT is said to ”bring new advancements and insights to the bioinformatics community by bringing advanced language modeling perspective to gene regulation analyses” [18]. In the following chapters, we examine in practice the

NLP tools' ability to read sentences written in the DNA language.

2.3 Related work

Traditionally, genes are detected with methods that use Open reading frames (ORFs) and statistics. By definition, ORF is a continuous sequence of nucleotides from a start codon to the following end codon, excluding the end codon itself [22]. ORF sequences are potential coding sections to be translated into proteins, but not all of these sections are coding sections, and not every gene is between consecutive start and end codons. For prokaryotes, a great proportion of the genes can be found using ORFs, but overlapping genes cause problems due to the mixed order of start and end codons.

The exon and intron structure of eukaryotic genes requires more complicated methods. An attempt can be made to identify exon and intron sequences by resolving the portions of nucleotides since G and C occur in exons more frequently than the other two nucleotides. Features derived from already known gene sequences of different organisms can also be used as model examples when annotating new genomes [22].

Markov models utilize conditional probabilities for the gene prediction task. Essentially, the model learns the probability of a certain nucleotide for different types of genome sequences, based on k previous nucleotides. Markov models and more complicated Hidden Markov models (HMM) [3] are widely used in gene predicting, although implementing them might be time-consuming and require domain understanding. Yet they are unable to completely predict the location of genes in genomes, and they also make false predictions.[1].

Neural networks bring a new perspective to the task by not being limited to human-provided rules and statistics [2]. Computational neural networks consist of layers of algorithms that perform computations for the input. The layers have multiple weights, which are adjusted during the training to result a model that can predict the desired output. Especially convolutional neural networks (CNN) are used for genome annotation tasks [2]. However, they only have a limited view on the input, and therefore challenges to capture insights from long genome sequences [18].

The performance of the neural network is dependent of the amount of learning data available. Often the task-specific data is limited, and this leads to a poor performance [18]. Transformer network architecture [34], explained in Section 3.2.2, supports a semi-supervised learning, in which the model is first fine-tuned with a massive amount of general genomic data [18]. It is then fine-tuned to the specific task, such as gene prediction, with a small labeled dataset. Transformers are already used

[1] Our reference system, GeneMarks, performs the prediction of *E. coli* genes with 82% accuracy, and 10% of all the gene predictions are false predictions

for other genomics related tasks, such as finding starting points for genes [9, 18]. In the following sections, we introduce a novel gene predicting approach, where Transformer neural network learns to read genes from genomic language.

3. Preliminaries

The main objective of this chapter is to give a detailed explanation of the Transformer network [34], the neural network used in the experiments. After defining the notations used in this thesis, the architecture of the network is described in Section 3.2. The following Sections 3.3 and 3.4 introduce some state-of-the-art Transformer models as well as the model used in the experiments detailed later in Chapter 4.

3.1 Notations

The following notations are used throughout the thesis, unless specified otherwise:

1. All numbers are denoted with lower-case letters and unless indicated otherwise, they belong to natural numbers (for instance $k \in \mathbb{N}$). Matrices are denoted with capital letters: $M \in \mathbb{N}^{m \times n}$. The notation $m \times n$ denotes the dimensionality of a matrix having m rows and n columns.

2. Superscripts are references to the corresponding element and used mostly in connection to weight matrices. For instance the weight matrix W linked to queries Q is denoted W^Q. Subscript i indicates the ith element of a list, such as input vectors $x_1, x_2, \ldots, x_m$.

3. Indexing starts from number one – not zero, as is common in the context of programming and coding. Intervals and ranges of integers are denoted with parenthesis and square brackets, $(m, n]$ meaning the range of integers from $m+1$ to n. Value m is not included as parenthesis $(\ldots)$ denote an open interval, and square brackets $[\ldots]$ indicate a closed one.

4. The names of the layers and elements of the network architecture, such as Attention or Encoder, are capitalized, to distinguish them from the general versions of the corresponding words.

3.2 Transformer architecture

The Transformer is a neural network architecture proposed in 2017 [34] and it has since become popular especially in the NLP field [7, 10], but also in other sectors like computer vision [29]. This section explains the architecture of the Transformer as well as the steps and parameters related to the training of the network.

As with neural networks in general, the architecture of Transformer networks consists of layers. The first layer handles the prepossessing of the input. After that, the main building blocks are called Attention, Feed Forward, Encoder and Decoder layers. The layers listed above are explained in that order, strictly following the original implementation of Transformers [34]. Explanations of different layers and the style of figures are adapted from the blog post "The Illustrated Transformer" [1].

3.2.1 Preprocessing

The input of the model, supposing it is natural language sentences, is first tokenized to a list of smaller parts, such as words, word parts, or punctuation. Every token is then embedded into a numerical form of word vectors. Several tools are available for tokenization and embedding[1], and the details of these steps are beyond the scope of this work. In the case of genome data, these preprocessing steps differ from the natural language approach and are discussed in more detail in Section 3.4.

The Positional Encoding layer ensures that the model knows the order and positions of the input vectors. The intuition behind the layer is to add a different kind of weight to every word vector, from which the model can derive the position of each word. Let there be m words in the input. For every word vector w_m with length n, and $i \in [1, n]$, the positional vector t_m is calculated as follows:

$$\text{Position}(w_m)_i = \begin{cases} \sin\left(\frac{m}{10000^{2i/n}}\right), & \text{when } i \text{ is even} \\ \cos\left(\frac{m}{10000^{2i/n}}\right), & \text{when } i \text{ is odd} \end{cases}$$

Each index $i \in [1, n]$ of the word vector gets a corresponding value in its positional vector. These values are sine wave values and therefore belong to the range $[-1, 1]$ giving each positional vector a unique representation. The output of the layer is simply vectors w_m and t_m added together.

[1] Natural Language Toolkit [4] for the tokenization, and word2vec [24] for the word embedding, for instance.

3.2.2 Attention

The Attention used in the original Transformer implementation is called Scaled Dot-Product Attention [34]. Technically, the actual version is Multi-Head Attention, which gives the model a broader understanding of the input. We will first examine the simple attention mechanism, and use elements of that further to describe the Multi-Head feature.

The first definition needed is a Softmax function [15] used in the Attention. The input of Softmax is a vector $x \in \mathbb{R}^n$, and it consists of elements $x_1, x_2, \ldots, x_n$. The exponential function of a single element is normalized by dividing by the sum of all exponents. When repeated to each element, the result is a vector length of n where all the elements are in the range $[0, 1]$ and sum up to 1.

$$\text{Softmax}(x)_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} \tag{3.1}$$

To calculate the Scaled Dot-Product Attention, one sentence worth of preprocessed word vectors is packed into a matrix and cloned into three matrices called queries, keys and values, marked with Q, K and V, respectively. During the training process explained in upcoming Section 3.2.5, the model learns corresponding weight matrices W^Q, W^K and W^V. The Attention, which helps the model to find the associations between input tokens, can be calculated using these six matrices:

$$\text{Attention}(QW^Q, KW^K, VW^V) = \text{Softmax}\Big(\frac{QW^Q \cdot (KW^K)^T}{\sqrt{d}}\Big)VW^V \tag{3.2}$$

The first step is to take a dot product of weighted queries and keys to give each input token a score against other tokens. The resulting matrix is scaled down with $\frac{1}{\sqrt{d}}$ to prevent situations where the dot product gets large and gradients of the Softmax get extremely small. Here d is the length of the input vectors.[1] The Softmax function, Equation (3.1) is then calculated for each row vector so that the values are scaled and sum up to 1. These values represent the score of how much every token is expressed at that particular position. Finally, those scores are multiplied by weighted values to get the Attention as an output. The steps from the input matrix X to the output Z are illustrated in Figure 3.1

The Attention function outputs a matrix which tells how much each input token weights related to other tokens. In natural language, these weights can represent semantic, syntactic or morphological dependencies, for instance. The intuition behind Multi-Head Attention is to capture multiple different types of these dependencies instead of one [8]. In Multi-Head Attention, Equation (3.2) is executed h times using the

[1] The scaling factor can also be something else, or even left out.

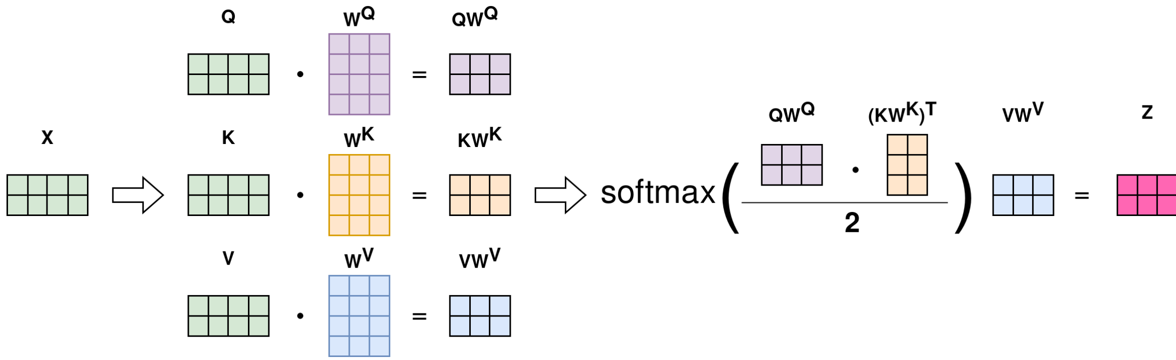


Figure 3.1: Data flow from the input vector X to calculated attention matrix Z. Rows in the matrix X represent input word vectors. Their length in this demonstration is 4, therefore the scaling factor here is $\frac{1}{\sqrt{4}} = \frac{1}{2}$.

same number of different weight matrices W_i^Q, W_i^K and W_i^V, where $i \in [1, h]$. Outputs are then concatenated into one matrix, and multiplied with the weight matrix W^O formed in the learning phase:

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \\ \text{where head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \end{aligned}$$

The resulting matrix is the output of the Multi-Head Attention layer. The dimensions of all matrices must be suitable for multiplication operations, and the final output of the attention layer must also be in the same form as the input. Therefore, if we have m input vectors of length n, weight matrices W^Q, W^K and W^V must have dimensionality $n \times \frac{n}{h}$ where h is the number of the heads. Weighted queries, keys and values then form $m \times \frac{n}{h}$ matrices. When all h Attention heads are concatenated, the resulting matrix has $h \cdot \frac{n}{h} = n$ columns, and therefore its dimensionality is $m \times n$. Lastly, the output weight matrix W^O has dimensionality $n \times n$ and the final output of the Multi-Head Attention layer is an $m \times n$ matrix denoted with Z. Multi-Head Attention as well as dimensions of associated matrices are illustrated in Figure 3.2.

There are three different versions of Attention in the Transformer implementation, which are described in the context of the overall architecture in Section 3.2.4.

3.2.3 Feed Forward network

When the Attention layer tries to emphasize important meanings and connections from the input, Feed Forward network (FFN) layer is said to capture the patterns and distributions of the input [14]. Intuitively, it tries to approximate or generalize the input and learn to predict the desired output from it. In the training phase, explained in more detail in Section 3.2.5, the parameters of FNN are tuned layer by layer based

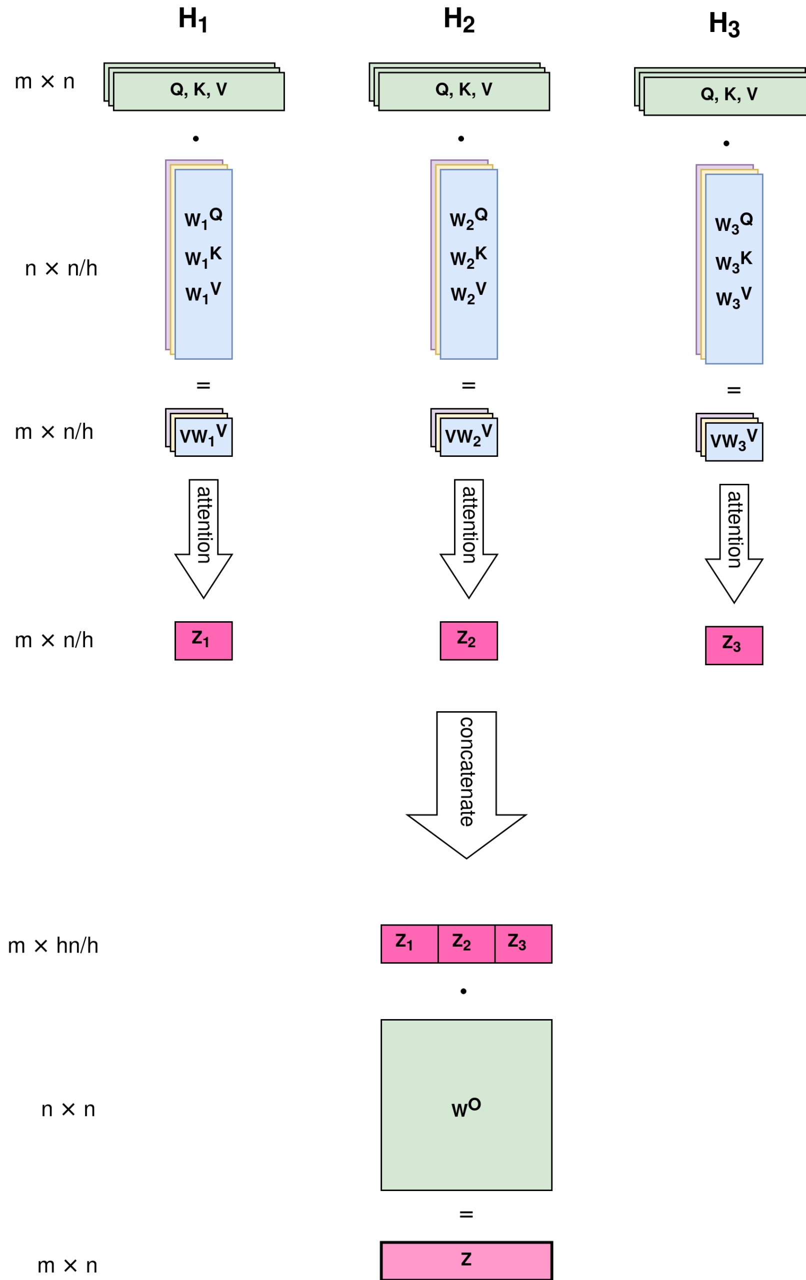


Figure 3.2: Simplification of Multi-Head Attention with three heads ($h = 3$). Dimensions of the matrices are given on the left.

on the feedback of error function (3.3). The FFN function is applied independently to every vector z. It uses weight matrices W_1^F and W_2^F learned in the training phase, as well as bias terms b_1 and b_2:

$$\text{FNN}(z) = \max(0, zW_1^F + b_1)W_2^F + b_2$$

More specifically, the function first has one affine transformation $y_1 = zW_1^F + b_1$ followed by Rectified Linear Unit (ReLU) activation $y_2 = \max(0, y_1)$, and finally another affine transformation $y_3 = y_2W_2^F + b_2$. In the original implementation of

the Transformer [34], the length of the inner layer vector is 2048 when the input and output are both length 512. Therefore W_1 and W_2 have to be 512×2048 and 2048×512 matrices, respectively.

3.2.4 Encoder and Decoder

Attention and Feed Forward layers are used as sublayers in two types of main layers, Encoders and Decoders. The overall architecture of the Transformer consists of a stack of Encoders, and an equal number of Decoders[1]. The original Transformer implementation has six layers of Encoders and Decoders.

After tokenization, embedding and positional encoding, the input first flows once through the Encoders sequentially. The first Encoder E_1 takes the preprocessed input, and the input of Encoder E_i is the output of the previous Encoder E_{i-1}, when $i > 1$. After that, a total of m iterations are needed through all the Decoder layers, where m is the desired amount of final output vectors. The final output is thereby printed out token by token, or word by word.

Both the Encoder and Decoder have a Multi-Head Attention layer described in 3.2.2. Because queries, keys and values are cloned from the same vectors, it is referred to as a Self-Attention layer in this context. The Decoder uses a masked version of Self-Attention, wherein the ith flow, the input vectors $x_m, m \geqslant i$ are masked with non-significant values, such as zeros. That forces the model to make predictions based only on the previous inputs. This so-called Auto-regressive feature is needed, for example, in machine translation or language generation tasks.

After the Masked Self-Attention layer, the Decoder has another type of Attention called Encoder-Decoder Attention. It differs from the Self-Attention in its input: The outputs of the final Encoder layer are used as keys K and values V and Masked Self-Attention outputs are used as queries Q. With values from the Encoder layer, the Attention function (3.2) can pay attention to all the positions of the original input sequence.

The final sub-layer for both Encoders and Decoders is the Feed Forward layer explained in Section 3.2.3. The FFN function is applied separately to each row vector of the Attention layer's output. In a sense, these row vectors represent individual tokens in the input, but the Attention function has already captured information about all the tokens to every row vector through the Softmax operation (3.1).

Figure 3.3 illustrates Encoders, Decoders and their sub-layers as well as shapes of inputs and outputs of the layers. In addition to these steps, every sub-layer output is connected to its input by adding them together at the end of the sub-layer. After

[1] There are also Transformers with only Encoders or Decoders, as it turns out in Section 3.3

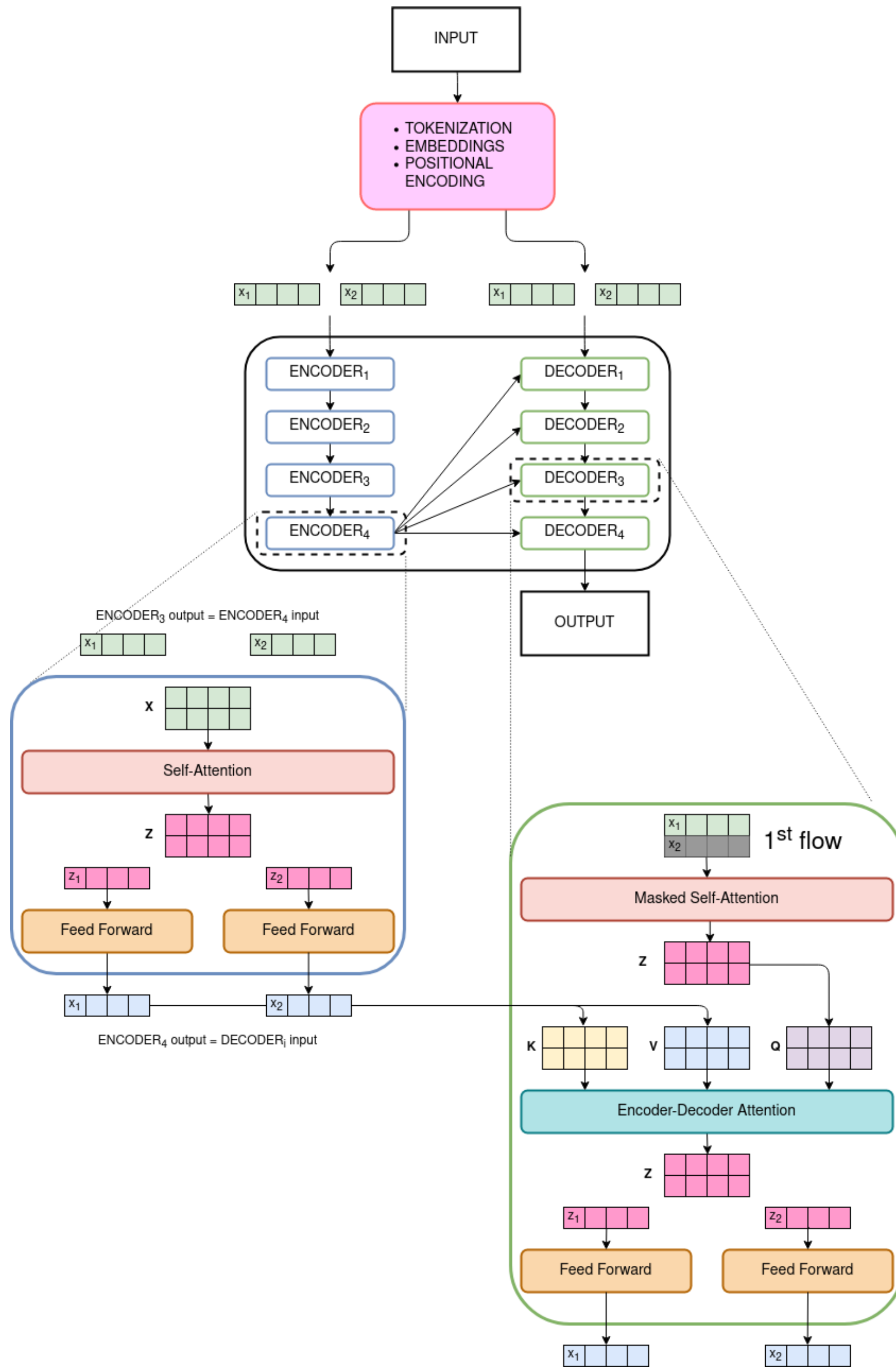


Figure 3.3: Architecture of Encoders and Decoders in Transformer network.

this so-called residual connection step, the sub-layer result is normalized [20]. These steps are applied to Feed Forward layers as well as to all three kinds of Attention layers: Self-Attention, Masked Self-Attention and Encoder-Decoder Attention. Residual connection and normalization are visualized in Figure 3.4.

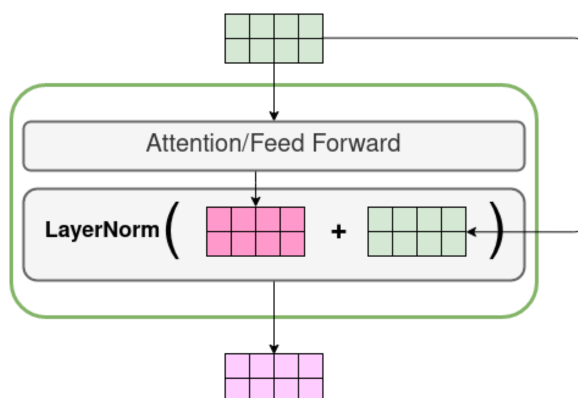


Figure 3.4: The residual connection and normalization in every Attention and Feed Forward layer first adds the layer output to its input, and then normalizes it.

3.2.5 Training

While the architecture is a feature that every Transformer network shares at some level, there are multiple different adaptations for training the network. Some of the typical practices and concepts related to the training phase are examined here, and more specific training parameters are introduced in association with the DNABERT model in Section 3.4.

Training in essence means learning the most optimal weight matrices so that the outputs are as close as possible to the desired outputs. The Transformer has weights W_i^Q, W_i^K, W_i^V and W^O related to Attention layers, as well as W_1^F and W_2^F in Feed Forward layers. Before training, all the weight matrices are initialized to contain random numbers forming a normal distribution. Bias terms used in the FFN function are set to $b_1 = b_2 = 1.0$. Every time the weights are used in Attention or FNN functions, updated versions of bias terms are calculated to be used in the next iteration. The predictions of these functions are compared to the desired outputs, and the difference between them is measured with an error function denoted here by e. The cross entropy loss function is used here as an example of e:

$$e = \sum_{i=1}^{n} -\hat{Y}_i \log(Y_i) \tag{3.3}$$

where $\hat{Y}$ and Y are desired and predicted outputs respectively, and n is their total amount. The result of the error function is used for updating the weights. One method for that is the gradient descent method, where the weight for next iteration $k+1$ is calculated using the previous weight W^k, learning rate η and a gradient of the error e with respect to the previous weight:

$$W^{k+1} = W^k - \eta \frac{\partial e}{\partial W^k}$$

The learning rate η is a small number that acts as a scaling factor, determining how much the derivative affects the weights at one step. Optimal weight updating often requires other methods than simply scaling the gradient at the learning rate. Many optimizing techniques are developed on top of the gradient descent method. DNABERT, for instance, uses an Adaptive Moment Estimation with Weight Decay (AdamW) optimizer [21] that utilizes momentum and weight decay methods with separate learning rate updates. These methods and their mathematical formulas are explained next.

First, to simplify the notations, the gradient with respect to the weight parameter W and time step t is denoted by g_t:

$$g_t = \frac{\partial e}{\partial W_t}$$

The purpose of the momentum terms is to help the model converge faster by strengthening the movement towards the promising directions.

When the alignment of the gradients' directions increases, the momentum terms increase and decrease correspondingly as the gradients' alignment decreases. AdamW uses two momentum terms, m and v as well as decay rates β_1 and β_2 connected to them respectively. Term m can be seen as an estimate of the mean of the gradients, and v as an estimate of the uncentered variance of the gradient. The terms are updated using the following formulas:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

β_1 and β_2 are usually set to 0.9 and 0.999 respectively by the authors' recommendation. Both terms are then corrected because they tend to bias toward zero.:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Lastly, the update formula of the weights is formulated, using the learning rate η, bias-corrected momentum terms $\hat{m}$ and $\hat{v}$ and weight decay value w:

$$W_{t+1} = W_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t - \eta w_t W_t$$

Index t refers to the step time as the parameters are updated separately for every step. To avoid dividing by zero, ε is added to the denominator of the division. The authors of the AdamW optimizer propose setting it to 10^{-8} or a similar extremely small value.

The weight decay mechanism works as regularization and therefore helps the model to avoid overfitting. Weight decay values, as well as learning rates, can also be updated separately for individual steps and parameters. This approach allows emphasizing their importance based, for instance, on the input frequency. The learning rate η is often updated with a process called *warmup* [16]. The weights are first initialized as random numbers, and therefore they might diverge in the wrong direction at the beginning of the training. The warmup method starts with a lower learning rate and increases it step by step until the desired value is achieved. The learning rate, weight decay and warmup are revisited in Section 4.4 which covers the hyperparameter optimization of our model.

3.3 BERT and other Transformers

Since introduced in 2017, there have been many successful implementations of the attention-based Transformer network [34]. Three of them are introduced in this section, two briefly and the last one, BERT, is examined more closely as a preliminary introduction to the DNABERT model used in the experiments.

OpenAI research lab has developed their Generative Pre-trained Transformer 3 (GPT-3) model with a somewhat similar Attention-based architecture than the original Transformer [7]. The key difference is that the GPT-3 has no Encoder layers [7, 28]. Instead, it is trained with 175 billion weight parameters and is thus one of the largest neural networks ever trained. It generalizes well in different NLP tasks and is known for the ability to generate fluent texts, such as complete articles, given just a short introduction as input [27].

Transformers are also used for other than strictly NLP models. DALL • E (named after the artist Salvador Dalí and movie character Wall-E) is a multimodal version of GPT-3 that produces images from natural language [29]. An example of the image generated with input words ”an armchair in the shape of an avocado” is shown in Figure 3.5.

Bidirectional Encoder Representations from Transformers, shortly BERT, is a state-of-the-art Transformer model, that comes with an open-source collection of pre-trained BERT models [10]. Good transfer learning capabilities are indeed one interesting feature of Transformers, as ”pre-training teaches the models about the structure of language” [8]. Therefore one pre-trained BERT model can be fine-tuned for several different tasks with just a few additional training layers and even a relatively small amount of data. Such an approach saves resources compared to training a different model for each task, and the savings in computational power lead, among other things, to lower CO_2 emissions [32]. Additionally, as explained below, pre-training is done

Figure 3.5: The image of "an armchair in the shape of an avocado" generated by DALL • E [26].

with unlabeled data and therefore there is no need for a great amount of task-specific, labeled data.

While GPT-3 has only Decoder layers, BERT only has the Encoder. For the training, it uses a Masked Language Model (MLM) approach, where part of the input tokens are masked and the model learns to predict their content by looking at the non-masked tokens around them [10]. The loss is computed between these predictions and the ground truth tokens behind the masked ones. The weights are then adjusted based on the loss.

More precisely, 15% of the input tokens are chosen randomly to be masked. From these tokens, 80% are replaced with a [MASK] -token, 10% with another random token and the rest 10% stay unchanged. This approach gives the model the ability to learn the desired outputs while avoiding overfitting to the masked tokens. It also allows for a bidirectional view of the input context, unlike in Decoder based systems where the Masked Self-Attention blocks the right side context of the input.

For fine-tuning, the weights of the model are initialized with pre-trained ones. The input used for fine-tuning is not masked, instead, depending on the task, the loss is computed between the outputs and ground truth values. Data used for the fine-tuning is task-specific, and usually labeled.

There are no labels in the pre-training, but the model gets the ground truth behind the masked tokens to adjust the weights. Therefore the pre-training part is called self-supervised learning, and fine-tuning is supervised learning. A large amount of unlabeled training data combined with a small amount of labeled fine-tuning data makes this approach in its entirety semi-supervised learning.

3.4 DNABERT

In our experiments, we use a BERT-based Transformer model that has been pre-trained with genome data. DNABERT [18] has a similar Encoder architecture and masking approach as BERT, with minor modifications due to the differences between natural language and genome inputs. DNABERT uses the human genome as training data. As explained in Section 2.1, a genome is a long sequence of nucleotides. For pre-training, the human genome is divided into windows with a maximum length of 510 nucleotides. When NLP approaches convert words into tokens, DNABERT uses a k-mer approach. Intuitively, the input window is divided into overlapping sequences of length k to form the "words" to tokenize.

More precisely, every window w is divided into k-mers, where $k = 3, 4, 5$ or 6. That is, for every index $i, i \in [1, 510 - k)$, there is a k-mer $t = n_i, n_{i+1}, \ldots, n_{i+k-1}$, where n is one nucleotide. These k-mers are further converted into tokens according to a vocabulary of all possible k-length nucleotide combinations. The k-mer approach and tokenization are demonstrated at the top of Figure 3.6.

This approach enriches the contextual information available for the layers of the network since every nucleotide is concatenated with the surrounding ones. However, this leads to issues with the random masking approach. The predicting task is too easy if the correct answer can be deduced from the neighboring tokens. Therefore, the masking approach is modified so that instead of random separate tokens, continuous spans of tokens are masked. Weights are then adjusted based on the predictions during the pre-training. For the first 100000 training iterations, 15% of the inputs are masked, and masking is increased to 20% for the last 20000 iterations.

The DNABERT model was pre-trained with cross-entropy loss shown in Equation (3.3). Other parameters for the training were set to $\beta_1 = 0.9, \beta_2 = 0.98, \varepsilon = 10^{-6}$ and weight decay $= 0.01$. The purpose of these variables is explained earlier in Section 3.2.5. In DNABERT, the Multi-Head Attention, explained in Section 3.2.2 consists of 12 heads and there are 12 Decoder layers in the model. Lastly, there is a Classification layer, that performs the final predictions for masked tokens. Figure 3.6 visualizes the layers of DNABERT architecture.

Unlike in pre-training, there are no masked tokens in the fine-tuning phase. Instead, the input has special [CLS] and [SEP] tokens referring to the beginning and the end of the window, respectively, as well as the corresponding label of the window. The final classification layer is trained by updating its weights according to the predicted and true labels.

The self-supervised pre-training gives DNABERT a general understanding of the so-called DNA language, which can then be fine-tuned for different tasks. The au-

thors give several examples, one of them called *prom-core*, where the input window is classified according to whether it contains a promoter[1] sequence or not. As a sequence-level classification task, it is justified to use the prom-core method as a basis for our corresponding experiments in which we classify gene sequences. The fine-tuning of DNABERT for our specific task is discussed in Section 4.2.

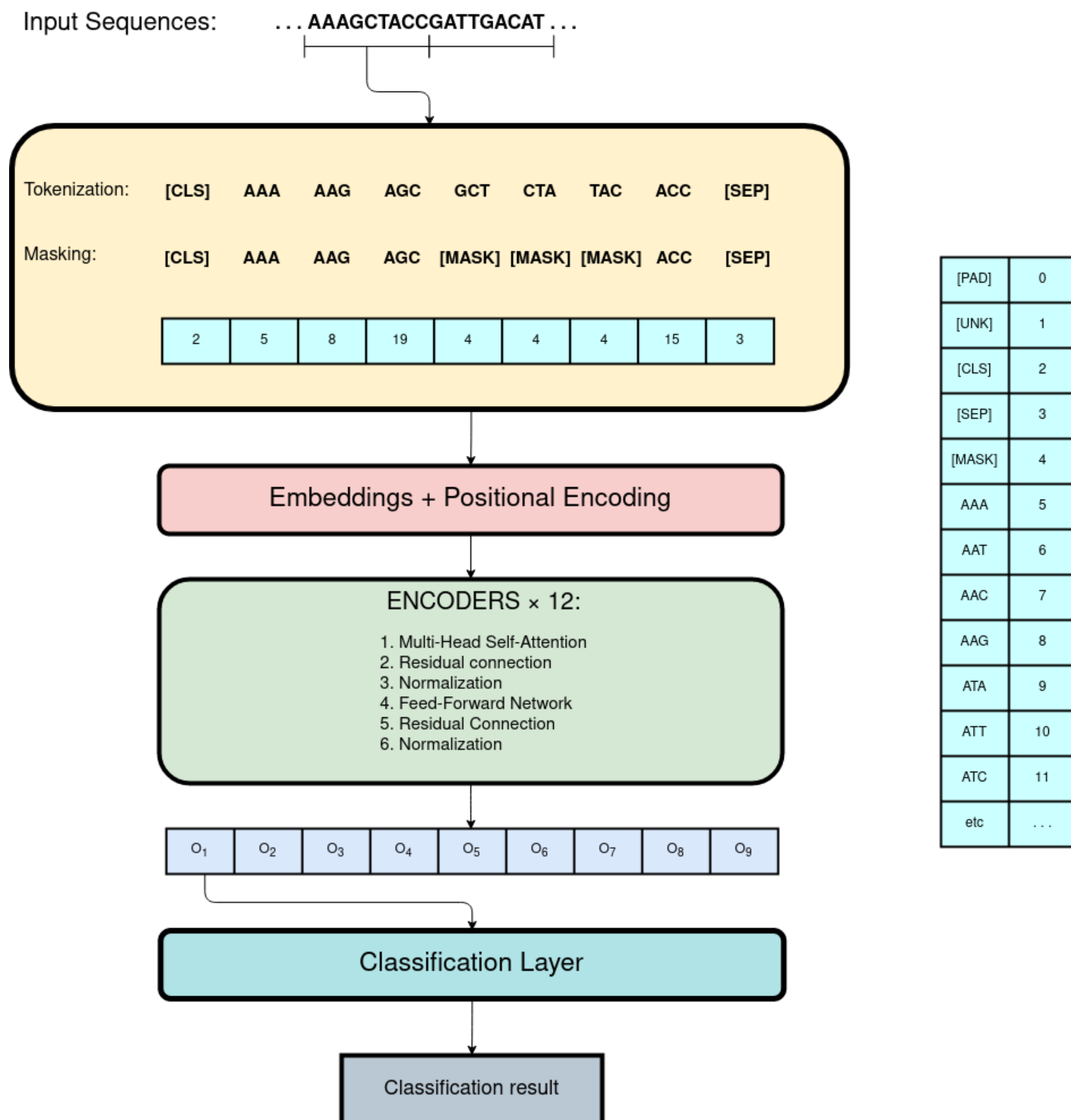


Figure 3.6: Demonstration of how one input sequence flows through the DNABERT model. After tokenized into 3-mers, some spans of tokens are replaced with [MASK]-tokens. The replacing takes place only in the pre-training. Every token has its corresponding index in the vocabulary, which is used for embedding. After the positional encoding is added, the pre-processed input vectors are sent through 12 Encoder layers. In the fine-tuning, the last hidden output goes to a final Classification layer, that outputs the results.

[1]Promoters are a certain type of DNA sequences that are related to genes and protein biosynthesis.

4. Experiments

The execution of experiments requires multiple steps, which are explained in this Chapter. The data is collected from the sources and preprocessed to be used for the training. Section 4.1 opens this process in detail. The tools and commands used for the fine-tuning are introduced in Section 4.2. Our evaluation setup and metrics are introduced in Section 4.3. After that, the actual training, including hyperparameter optimization and the results, is elaborated in Section 4.4.

4.1 Collecting data

For the fine-tuning process, the training data must be labeled according to the gene predicting task. Therefore we need the nucleotide representation of the genome, as well as the gene coordinates, to form the proper dataset for training. In this section, the collecting of the data is explained for two sources, the *E. coli* genome and the human chromosome 18.

4.1.1 *Escherichia coli* K-12

For the experiments with the genome of the *Escherichia coli* (*E. coli*) bacterium, two text files are loaded from the RegulonDB database [30, 31] The genome sequence of *E. coli* K-12 bacterium, genome version U00096.3, is loaded in fasta format[1]. In addition, the gene sequences of that same *E. coli* bacterium are also loaded in the same format. One line in the gene file represents one gene and includes the left and right end coordinates of each gene, DNA strand direction (forward or reverse) and the nucleotide sequence of the gene. There is also other information such as the name and identifier of the gene, but those are not used in the experiments.

The lines of the *E. coli* genome file are concatenated to one long string, containing a total of 4641652 nucleotide characters. Preparing the gene file requires several steps. First, 21 gene sequences with no coordinates and nucleotide sequences are left out. The lines are then ordered based on the left end coordinate of each gene, and only

[1]Fasta format is an established way of presenting bioinformatics sequences in text form.

coordinates, directions and nucleotide sequences are included. The ground truth data of *E. coli* genes thus includes 4665 gene sequences and coordinates, 2297 in the forward direction and 2368 in the reverse direction of the DNA strand.

4.1.2 Human chromosome 18

The size of the full human genome is over three billion nucleotides. This dataset is too large in to be utilized reasonably with the tools and computational resources available. To make the experiment more manageable, we chose to use only a part of one chromosome. This reduced dataset is of the same order of magnitude with *E. coli* dataset.

The human genome file `GRCh38_latest_genomic.fna` is available in fasta format from National Center for Biotechnology Information (NCBI) web page [12], and chromosome 18 is extracted from it using the `grep` command-line tool. Lines in the file represent the nucleotides of the chromosome. They are written to a text file and then concatenated into a sequence containing a total of 80373285 nucleotides.

As explained in Section 2.1 human genes are more complicated than, for instance, genes of a bacteria. Exons are the main parts of human genes, while introns are non-coding clips between them. To simplify the prediction task we use only exons instead of the full human genes. Coordinates of the exons are extracted with the following steps: From NCBI, we first load file `GRCh38_latest_genomic.gff` containing different sequences of the human genome. The file is in General Feature Format (GFF), meaning that every line contains nine fields, from which we are interested in four: 1. Sequence id, 3. Type of feature, 4. Start position and 5. End position of the feature. Lines with Sequence id = `NC_000018.10` and Type = `exon` are extracted to a file called `chr18exons.txt`. The file now contains a total of 35998 exon coordinates, and these are ordered in the same way as the *E. coli* gene coordinates above to form ground truth data of exons.

Chromosome 18 is about five times the size of *E. coli* bacterium in terms of nucleotide amount. To make the two datasets equal in size, chromosome 18 is reduced to about one-fifth of the original. The cutting of the chromosome and exon coordinates is done as follows:

1. Calculate the cutting point by using the length of the full chromosome: $80373285/5 = 16074657$

2. If the cutting point is inside an exon sequence, move the point to be the last index of non-exon

3. Only exon coordinates with an end point smaller than the cutting point are included

4. The first 10000 nucleotides are filler characters 'N' and are therefore excluded. Exon coordinates are relocated accordingly.

The reduced chromosome 18 dataset contains 15315920 nucleotides and 10285 exon sequences.

4.1.3 Preprocessing

Genome data input for the pre-trained DNABERT model has to be divided into context windows of length $w < 512$. Each window has to be in the k-mer form, k being $3, 4, 5$ or 6. For instance, if $k = 3$ and one of the windows is ACCGT with $w = 5$, the k-mer representation is ACC CCG CGT. The model is then able to transform each k-mer into a numerical token using the dictionary of all possible nucleotide combinations of length k and corresponding integers. For instance in case of $k = 3$ there are $4^3 = 64$ tokens, and when $k = 6$ there are total of $4^6 = 4096$ possible tokens. Eventually, every context window has its own representation consisting of a list of tokens.

For the model to be able to learn genes and non-gene parts of the genome, we also need a label for each window telling whether the nucleotides belong to the gene or not. For that purpose, we form a label sequence equal to the length of the genome, where every index corresponding to a nucleotide is either 1 or 0 depending on whether it belongs to a gene or not, based on the gene coordinates collected earlier. The final labeling of train data is not the same as the labeling of the test data. For the training data, the ground truth labels are presumably available. The testing data, instead, must follow the scenario where the goal is to predict labels for the novel genome data without knowing the coordinates. Ground truth coordinates are still used for the evaluation of the predictions, as we see in Section 4.3.

The training data is first sequenced into gene and non-gene sequences by separating them from the label exchange points. Every sequence is then divided into windows with a maximum length of w. In this way, also shorter windows than w are created in situations where the length of the sequence is not divisible by w. The window-level labels are the same as the nucleotide-level labels inside each window.

Instead, the test data is first sectioned evenly into w-long windows, and every window is then labeled proportion of nucleotide-level labels. We consider three different methods for labeling:

1. The window is labeled as 1 if *more than half* of its nucleotides belong to a gene, otherwise, the label is 0.

2. The window is labeled as 1 if *all* of its nucleotides belong to a gene, otherwise, the label is 0.

3. The window is labeled as 1 if *any* of its nucleotides belong to a gene, otherwise, the label is 0.

More formally:

$$L^w = 1, \text{ if } \sum_i L^{n_i} > \frac{w}{2}, \text{ else } L^w = 0 \tag{4.1}$$

$$L^w = 1, \text{ if } \sum_i L^{n_i} = w, \text{ else } L^w = 0 \tag{4.2}$$

$$L^w = 1, \text{ if } \sum_i L^{n_i} > 0, \text{ else } L^w = 0 \tag{4.3}$$

Here L^w is a label of the window, and L^{n_i} is a label corresponding to a nucleotide in the ith position of the window. The first method is the most justified because it emphasizes the contribution of non-genes and allows the model to identify sequence change sites. Therefore method 1 was adopted for the experiments in Section 4.

Before the genome is partitioned into windows, labeled and formed into k-mers, it is split into separate training and evaluation sets. We use a 70 : 30 split, the training dataset being 70% of the size of the whole dataset, and the testing dataset being 30%. Window-label pairs in the training dataset are shuffled whilst testing data remains in order. The training data is saved as `train.tsv`[1] and the test data as `dev.tsv` so that DNABERT can read them. A simplified example of forming the train and test sets, from genome and coordinates to labeled windows of k-mers, is illustrated in Figure 4.1.

Training datasets are then used to fine-tune the DNABERT model as described in Section 4.2, and the results of predicting labels for the test set are given in Section 4.4. The codebase for collecting and processing the data is available in Appendix B.

4.1.4 Dataset variations

By using different window lengths and k-mers we get multiple variations of the data. In our experiments with *E. coli*, we use lengths between $10, 25, 50, 75, 100, 200, 300$ and 500 for the context windows and transform them into 3-mers, 4-mers or 6-mers. We also use one dataset with window length 75 and $k = 4$. With the human chromosome 18 data, variations are somewhat similar: Window lengths $15, 25, 75, 100, 200$ and 300 are used with 3-mers and 6-mers when experimenting with one-fifth of the full data size. We also did an experiment with the full chromosome, using 3-mers and window size 75.

[1]A text file format in which values are separated by a tabulator.

Figure 4.1: The train and test labels are formed differently, as seen in this example. The test windows here have an absolute length of 5 while the train window lengths are in the range of 3 to 5 depending on the gene coordinates. Sections that are shorter than k have to be excluded from the input, as they would not have a corresponding numerical token in the dictionary. However, the share of these sections and the impact on the overall result is extremely small. For test window labeling, both methods 1 or 2 lead to the same label in this example.

In addition to varying window lengths and k-mers, we also did some attempts to make the data more balanced. Over 88% of *E. coli* genome consists of genes., and only 5% of human chromosome 18 is exons. These imbalances are far from optimal in labeling tasks such as ours, and therefore it is arguable to try to either reduce the dominating label or increase the diminutive one. [36] With *E. coli*, we created a dataset using only coordinates of forward genes and ignoring reversed ones. That led to 43 : 57 ratio between genes and non-genes. The same method was used to create a dataset with only reversed genes with a ratio of 46 : 54.

With human chromosome 18, balancing experiments were executed by removing sequences that were labeled as 0, in other words, non-coding sequences between exons. To get the ratio between labels close to 50 : 50 most of the non-coding sequences have to be removed. In our experiments using one-fifth of the chromosome, we removed 1300 randomly chosen sequences out of 1410 non-coding sequences. With this approach, the

ratio between labels 1 and 0 was 39 : 61.

Discussion about these balancing methods, as well as results of balanced dataset variations, are covered in Section 4.4.2. It is notable here, that the main experiments were done with the unbalanced *E. coli* dataset, the results of which can be found in Section 4.4.1.

4.2 Fine-tuning the DNABERT

The overall process of fine-tuning the pre-trained DNABERT-model is described in this section, as well as the chosen execution environment. The selection of training parameters is omitted from this section, although it is strongly related to fine-tuning. Instead, Sections 4.3 and 4.4 explain in detail how the evaluation method serves in the selection of optimal parameters.

Collecting and preprocessing of each dataset variation generates two text files containing training and testing data. In order to use the DNABERT repository as such, the input files must be named as `train.tsv` and `dev.tsv`. Files for different variations are stored in separate, descriptively named folders.

The DNABERT code-base is open source and the repository is available on Github [35]. We clone the repository into a Colab notebook and install all the requirements according to the instructions in the `README.md` file. Colab notebook is a browser-based environment for running python code using Google Cloud. It was chosen due to its capacity to use cloud-hosted GPU resources for the computations.

Depending on the chosen k for k-mer size, we load a corresponding pre-trained DNABERT model from the Github repository. We then specify the paths for the pre-trained model, data folder and output model, as well as all the training parameters. Fine-tuning is then performed by executing the `run_finetune.py` python file from the repository with the `do_train` option. The paths to the data and models are defined in the command, as well as the training parameters. Appendix A shows the details of the command, as well as a minor modification to the code base before executing it.

The fine-tuning function uses the data from `train.txt` to modify the weight matrices of the pre-trained model and saves the new, fine-tuned model to the specified path. It takes from a couple of minutes to about an hour to run one epoch in the Colab, depending on the dataset configurations: When the data is divided into the shortest windows ($w = 15, 25$), the amount of input rows is substantially larger than with the longest windows ($w = 300, 500$). This naturally increases the iterations of the neural network layers (introduced in Section 3.2.2), and thus the time spent on training.

After fine-tuning, the results are generated by executing `run_finetune.py` again, this time with the `do_predict` option. For the results, we use the test data created

in Section 4.1.3 and saved as the `dev.tsv`. DNABERT uses this file as an input and predicts labels for those sequences based on the fine-tuned weights of the model. DNABERT then prints out prediction results like accuracy, F1-score, precision and recall, but unfortunately, these numbers are not usable in our use case. As explained in Section 4.1.3, labels for test data are formed at window level, and DNABERT calculates the results for predictions based on these labels. Instead, we want to evaluate the predictions at the nucleotide level. Tailored evaluation calculations are explained in next Section 4.3.

The model was fine-tuned with different hyperparameters, values of k, window sizes and amounts of epochs. The hyperparameters, such as learning rate and weight decay, define the weight update process according to Section 3.2.5. Window size and k-mer affect to the training data structure, and the epochs determines the amount of training cycles through the data. Both datasets, *E. coli* and human chromosome 18 were utilized, but *E. coli* was studied in more depth, while only a few different configurations were tested for the human dataset. The process of how the final configurations were chosen is described in Section 4.4.

4.3 Evaluation setup

Once the DNABERT-model has been trained with the training data, we test it by predicting the results for the test data. The `run_finetune.py` function writes the results in a NumPy[1] file `pred_results.npy` where for every input window of k-mers, there is a corresponding value that is in the range of 0 to 1. The default threshold value is 0.5. Windows with smaller values get label 0 and therefore are predicted as non-gene sequences, and windows with greater value are predicted to belong to a gene with label 1.

The k-mer structure of the input data is no longer needed in the evaluating process. Instead, we expand the predicted label 1 or 0 to apply to every individual nucleotide in the corresponding window. For instance, if the predictions for three consecutive windows are 0.65, 0.8 and 0.45, we get labels $1, 1$ and 0 for the windows. Next, every label is multiplied by the window size so that we have as many labels as nucleotides. With window size 5 we end up having a label sequence of $1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0$ in this example. The resulting sequence is then compared to the ground truth labels of the corresponding sequence of the original genome. From predicted and ground truth labels we count the amounts of true positives (TP), false positives (FP), true negatives (TN) and false negatives (TN). Figure 4.2 demon-

[1]Python's library for efficient array representations and functions.

strates more closely the process of addressing predicted labels and computing the TP, TN, FP and FN counts.

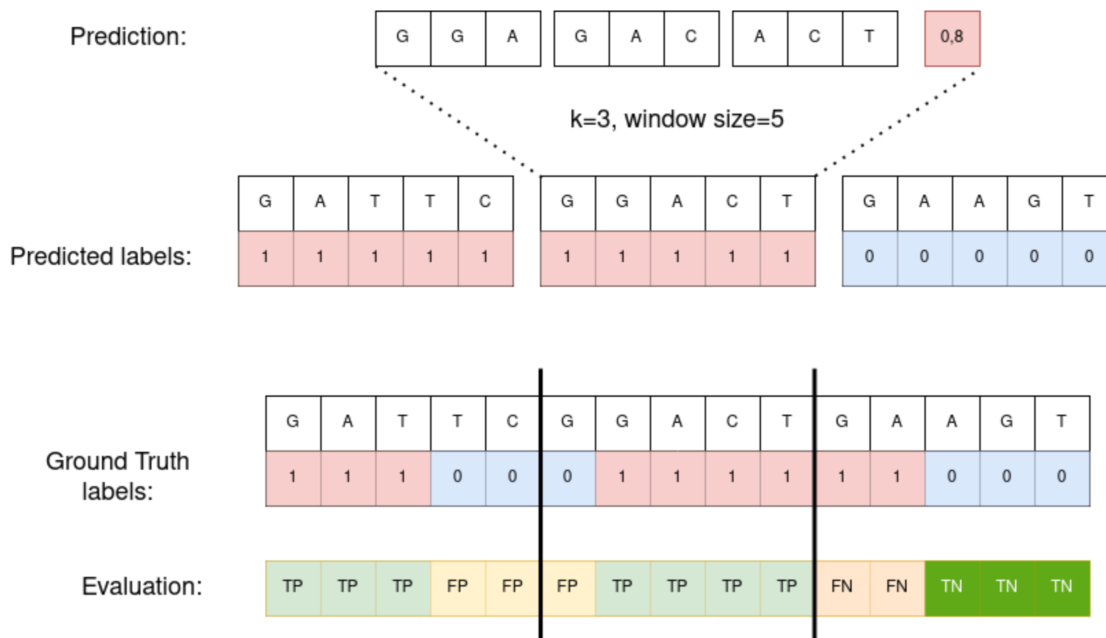


Figure 4.2: Evaluation process from predicted labels to comparison against ground truth labels. In this example, the window size is 5 and the train data is in the form of 3-mers. The bottom row demonstrates the formation of true positives (TP), false positives (FP), true negatives (TN) and false negatives (TN). Accuracy here is $\frac{\mathrm{TP+TN}}{15} = \frac{10}{15} = 0,66$

Based on these counts, we calculate accuracy, precision, recall, F1-score and Matthews Correlation Coefficient (MCC) using the following formulas:

$$\text{accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{n}$$

$$\text{precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

$$\text{recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

$$\mathrm{F1} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

$$\mathrm{MCC} = \frac{\mathrm{TP} \cdot \mathrm{TN} - \mathrm{FP} \cdot \mathrm{FN}}{\sqrt{(\mathrm{TP}+\mathrm{FP})(\mathrm{TP}+\mathrm{FN})(\mathrm{TN}+\mathrm{FP})(\mathrm{TN}+\mathrm{FN})}}$$

Here n is the count of nucleotides in the test data, and also $n = \mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}$. Accuracy is simply the proportion of correct predictions of all predictions. It

must be noted here that if the proportion of genes from the whole genome is high, the accuracy would also be high even with a naive model that predicts all the labels to be 1. Therefore, the aim is to get at least better accuracy than the proportion of genes, to make sure the model has actually learned the predictions using the input data.

Precision tells which fraction of gene predictions are correct, and recall is the share of correct gene predictions from ground truth genes. Precision and recall can also be described as positive predictive value and sensitivity, respectively. F1-score is the harmonic mean of precision and recall, and therefore indicates the accuracy of those two values combined. F1-score, as well as precision and recall, does not take true negatives into account, and therefore it might be misleading in situations where the balance between labels is biased, like in our datasets. Therefore these metrics must be interpreted with caution, as we wil see in the Section 4.4 in the context of the results. For instance, a recall value of a naive model is misleadingly 1 since the FN count is zero.

MCC is said to be a better metric for evaluating binary classification models like ours, especially when labels are unbalanced [5]. There might be situations where the denominator would be zero, like the naive model mentioned above. In this case, the value of MCC is set to zero based on the following reasoning: All the values are non-negative whole numbers. Therefore, if any of the sums in the denominator is zero, it means that both of its addends are zeros. Then, also the numerator has to be zero, as both of the multiplications have zero as a factor.

MCC gives values between -1 and 1. However, if the result is exactly $-1, 0$ or 1, the metric is not defined [5]:source. In those situations, other metrics should be considered instead. A random model gets MCC values close to zero, and a good model is recognized by the values approaching one.

In addition to the metrics described so far, we also plot four figures from each training to help in the evaluation of the fine-tuned models. An example of these plots is in Figure 4.3.

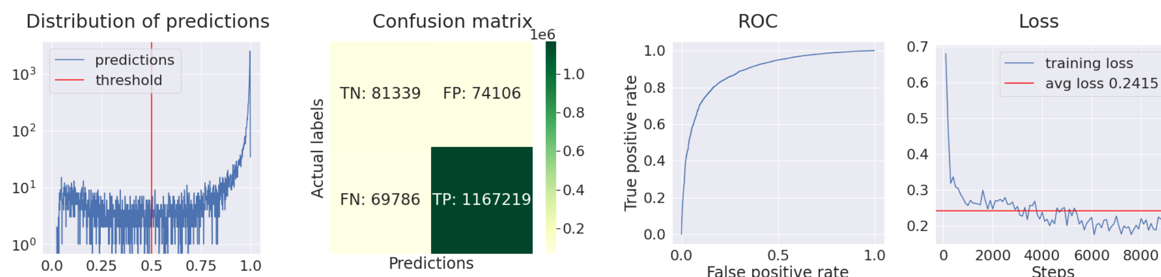


Figure 4.3: Example of figures plotted during the training.

The first plot shows the distribution of the predictions on both sides of the thresh-

old value of 0.5. The second plot, confusion matrix, simply shows the TP, TN, FP and FN values visualized in the colored grid. The third plot, Receiver Operating Characteristic curve (ROC curve), is explained below, and the fourth plot gives information about the loss function during the training. The concept of the loss was discussed earlier in Section 3.2.5.

For plotting the ROC curve we need the True Positive Rate (TPR) and False Positive Rate (FPR). TPR is simply the recall or sensitivity. FPR is also called the *probability of false alarm*, and it is calculated by $1-TPR$. TPR is plotted against FPR over different thresholds from 0 to 1. A diagonal curve means no better than a random model, and a well performing model has plenty of space above the diagonal and below the curve. Curves below the diagonal technically mean worse than a random model, but in that situation switching the labels between each other mirrors the curve by the diagonal, and therefore also makes the model better.

Our evaluation method tells us what windows are predicted to belong to a gene and which do not, and processes the results at the nucleotide level. This differs from some other approaches, such as GeneMarkS, which gives start and end coordinates of the predicted gene positions. Our approach may mistake in its predictions in two different ways::

1. There can be windows inside the true gene sequences labeled as non-genes or vice versa.

2. Beginnings and endings of the genes or non-genes can be labeled wrongly, as the window start and end points will most likely not hit exactly the ground truth coordinates.

The coordinate-based approach, in turn, attempts to predict the entire gene sequence at once by giving the start and end coordinates. We evaluate the GeneMarkS results also in the nucleotide level as a reference to our results in Section 4.4.1

4.4 Results

Based on the observations during the preprocessing phase and the first fine-tuning trials, the unbalanced *E. coli* dataset was chosen for a more detailed model optimization process. Subsection 4.4 focuses on the results from that process. Results from the balanced *E. coli* dataset variations, as well as human chromosome 18 results are briefly mentioned in Subsection 4.4.2.

4.4.1 Primary results

This section describes the results obtained with different fine-tuned models using E.coli data, as well as the hyperparameters and other settings used for fine-tuning those models. Therefore, this section will also explain how the optimization process proceeded from the first trials towards the final and best models. The main evaluation results to be examined are accuracy, MCC, $F1$-score and precision, all previously explained in Section 4.3. Loss graphs and ROC curves were also utilized to examine and explain the performance of the models.

Baseline parameters for the initial fine-tuning were chosen according to the ones used in the original DNABERT experiments [18], and more specifically in fine-tuning of their Prom-core model. Table 4.1 shows the parameter settings found in an appendix of the DNABERT article. The first line shows the initial parameters used. The most relevant parameters here are $k = 3$, learning rate $= 10^{-4}$ and sequence length $= 75$. Throughout this thesis, the corresponding term for "sequence length" has been "window size" and instead of steps, we use the term epoch. Hidden dropout probability, warmup percent and weight decay were set to $0.1, 0.06$ and 0.01, respectively, as shown in Table 4.1.

Promoter-core							
DNABERT-Prom-core							
k-mer	learning rate	batch size	seq length	dropout	warmup	max step	weight decay
3	1.00E-04	64	75	0.1	0.06	2	0.01
4	3.00E-04	64	75	0.1	0.06	2	0.01
5	2.00E-04	64	75	0.1	0.06	3	0.01
6	2.00E-04	64	75	0.1	0.06	3	0.01

Table 4.1: Hyperparameter settings used for fine-tuning the original DNABERT Prom-core model. [18] The hyperparameters in the top row were used as a baseline setting in the beginning of the parameter optimization process.

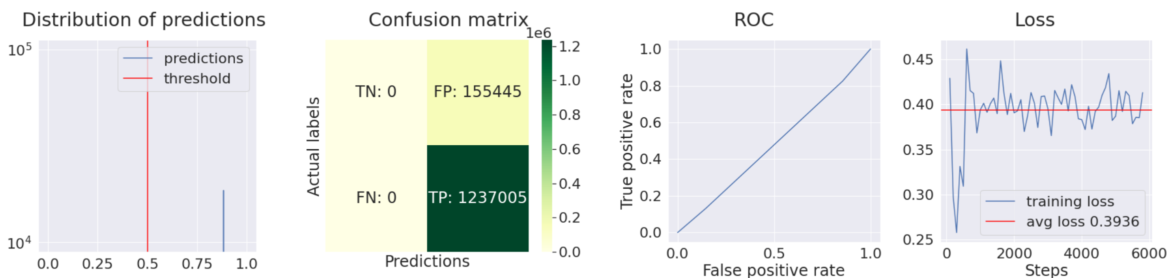


Figure 4.4: The initial training parameter setting: $k = 3$, epochs $= 2$, $w = 75$, $lr = 10^{-4}$, dropout $= 0.1$, weight decay $= 0.01$, warmup $= 0.06$.

The evaluation of the first fine-tuned model produced the following results: accuracy $= 0.8884$, MCC $= 0$ and F1-score $= 0.9409$. MCC being zero means that there is something wrong in the model, as explained in Section 4.3, and the distribution of

predictions in Figure 4.4 indeed shows that the first model is naively predicting every nucleotide belonging to a gene with the same predicted value. The loss is not decreasing as it should, which is also a characteristic of unsuccessful training. This is the starting point of the parameter optimization process and every evaluation affected the direction of the next modifications of the parameters. The process can be divided into four iterations:

1. Learning rate, epochs and k-mers
2. Warmup percent, weight decay and dropout probability
3. Window sizes
4. Window size validation with learning rates

After four iterations, a summary of the results is discussed. Also the results of the control system, GeneMarkS [3], are briefly presented.

Iteration 1

The learning rate affects the process of updating the weights, as we recall from Section 3.2.5, and the first step was to optimize it. From learning rates $10^{-4}, 10^{-6}$ and 10^{-8} the middle one gave the best results and encouraged us to try more values near it. Lower learning rate of 10^{-7} again led to a "one label model", but with a higher learning rate of $5 \cdot 10^{-6}$, accuracy 0.9027, MCC 0.4316, F1-score 0.9464 and precision 0.9262 were achieved.

Figure 4.5 shows the plots from the learning rate 10^{-8}, where the distribution of predictions seems somewhat promising, but the ROC curve revevals that the model is not working properly. The loss curve indicates that the learning was still in progress. Therefore, the next step of training was to have three epochs instead of two. From the learning rates 10^{-7} and $5 \cdot 10^{-6}$ the former did not improve the results, but the latter one proved to give the best MCC and precision so far, MCC being 0.4728 and precision 0.9403. Accuracy and F1-score were slightly lower than with two epochs, the Accuracy being 0.8967 and the F1-score 0.9420.

In addition to $k = 3$, other k-mer values were also explored using three epochs and a learning rate of $5 \cdot 10^{-6}$ but the results were slightly worse than with the 3-mer. The exact results from fine-tuning with different learning rates, epochs and k-mers are found in Table 4.2.

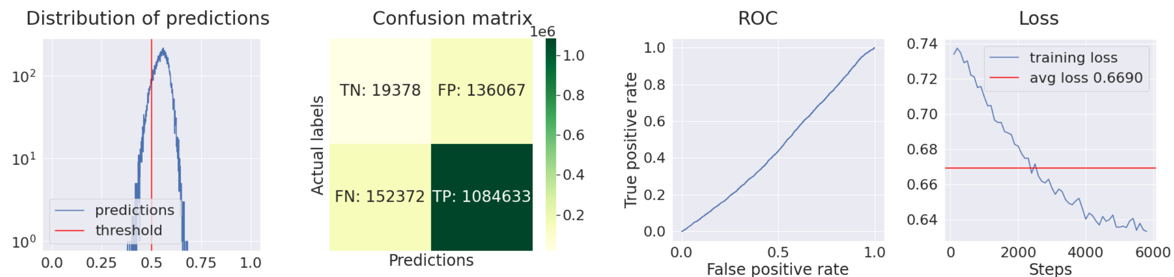


Figure 4.5: $k = 3$, epochs $= 2$, $w = 75$, $lr = 10^{-8}$, dropout $= 0.1$, weight decay $= 0.01$, warmup $= 0.06$

k	Epochs	w	Learning rate	Accuracy	MCC	F1-score	Precision	Recall
3	2	75	1.00E-04	0.8884	0	0.9409	0.8884	1
			1.00E-06	0.8935	0.2692	0.9426	0.9037	0.9851
			1.00E-08	0.7928	0.0014	0.8826	0.8885	0.8768
			5.00E-06	0.9027	0.4316	0.9464	0.9262	0.9675
			1.00E-07	0.8884	0	0.9409	0.8884	1
	3	75	5.00E-06	0.8967	0.4728	0.942	0.9403	0.9436
			1.00E-07	0.8884	0	0.9409	0.8884	1
6	3	75	1.00E-04	0.8884	0	0.9409	0.8884	1
			5.00E-06	0.8981	0.3898	0.9441	0.9209	0.9685
			1.00E-06	0.8915	0.2662	0.9414	0.9047	0.9812
4	3	75	5.00E-06	0.901	0.4	0.9458	0.9207	0.9722

Table 4.2: Learning rate results from the first iteration. Blue cells indicate the two highest results of the metrics of each column. Red cells highlight the models where the whole genome is labeled as a gene.

Iteration 2

MCC is a better indicator for evaluation than accuracy or $F1$-score in our setup, as explained in Section 4.3, and therefore the learning rate was fixed as $5 \cdot 10^{-6}$, which gave the highest MCC and precision values. Along with that, three epochs, 3-mers and $w = 75$ were used in this iteration, where the purpose was to optimize the other hyperparameters. Modifying the warmup percent or the weight decay had very little effect on any of the metrics, as Table 4.3 shows. Instead, increasing the dropout probability from the initial 0.1 to 0.5 reduced the values. Lower dropout values 0.01 and 0.06 showed minor improvements in the accuracy and the $F1$-score, but slightly lower MCC values as well. The differences being insignificant, the warmup percent and the weight decay values were not modified, but the dropout probability value was fixed

to be 0.06 for the further experiments with different window sizes.

Warmup	Dropout	Weight decay	Accuracy	MCC	F1-score	Precision	Recall
0.06	0.1	0.01	0.8967	0.4728	0.942	0.9403	0.9436
0.1	0.1	0.01	0.8964	0.4719	0.9418	0.9403	0.9432
0.06	0.1	0.1	0.8966	0.4713	0.9419	0.94	0.9438
0.06	0.1	0.5	0.897	0.4717	0.9422	0.9398	0.9446
0.06	0.5	0.01	0.8858	0.3344	0.937	0.9183	0.9565
0.06	0.01	0.01	0.9036	0.457	0.9466	0.9314	0.9623
0.06	0.06	0.01	0.9015	0.4683	0.9451	0.9359	0.9545

Table 4.3: Hyperparameter rate results from the second iteration. The two highest values of each column are colored blue.

Iteration 3

The model was then fine-tuned with different window sizes. In addition to the three hyperparameter values mentioned in the previous iteration, other parameters used here were $k = 3$, learning rate $= 5 \cdot 10^{-6}$ and epochs $= 3$. Figure 4.6 shows the ROC curves or windows with lengths of $10, 25, 50, 75, 100, 125, 200$ and 500 nucleotides, including the initial window size 75. The outermost window sizes 10 and 500 clearly have the lowest curves, while curves for the middle sizes $75, 100$ and 125 rise to the highest. Result values in Table 4.4 confirm this observation. While other window sizes got clearly lower results than the initial size of 75, the results with window size 100 were worth closer examination: The accuracy and the $F1$-score were slightly higher than with $w = 75$ but the MCC was 0.4339 and therefore lower compared to 0.4728 with the window size 75. The same observation applies to the other k-mers, 4 and 6, examined in this iteration. When looking at the confusion matrix in Figure 4.7, window size 75 with a learning rate of $5 \cdot 10^{-6}$, $k = 3$ and three epochs seems promising: True negatives count is greater than both false negatives or false positives, unlike in the other settings.

Iteration 4

As a summary from previous steps, the models fine-tuned with learning rate $5 \cdot 10^{-6}$ and window sizes 75 and 100 achieved the best results. However, fine-tuning with different window sizes was executed with only one learning rate, $5 \cdot 10^{-6}$, which was fixed in the first iteration. As a validation of the best window size, the final iteration was to perform a grid search using the following sets of values for window sizes and learning rates:

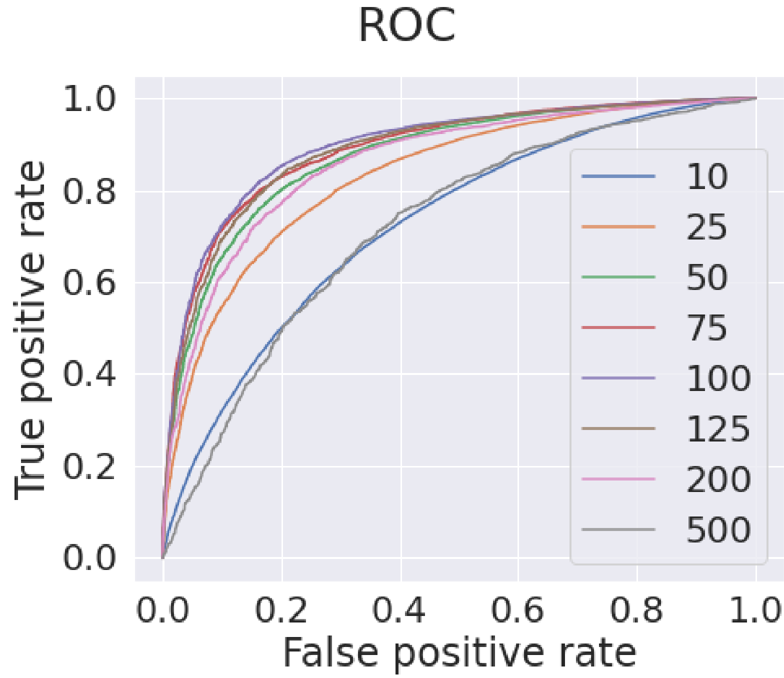


Figure 4.6: ROC curves for different window sizes explored in the third iteration. Other parameters: $k = 3$, learning rate $= 5 \cdot 10^{-6}$ and epochs $= 3$

.

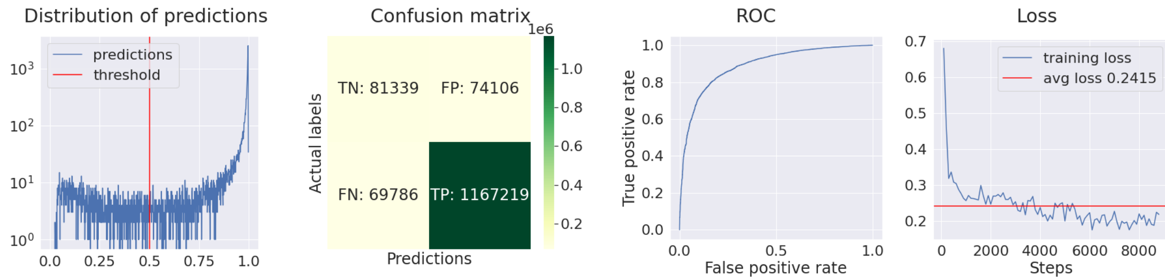


Figure 4.7: $k = 3$, epochs $= 3$, $w = 75$, $lr = 5^{-6}$, dropout $= 0.06$, weight decay $= 0.01$, warmup $=$ 0.06.

$$\text{window sizes} = \{25, 50, 75, 100, 125, 200\}$$
$$\text{learning rates} = \{10^{-4}, 5 \cdot 10^{-6}, 10^{-6}, 10^{-7}, 10^{-8}\}$$

Other parameters stayed as they were in the previous iteration: epochs $= 3$, dropout $= 0.06$, weight decay $= 0.01$ and warmup percent $= 0.06$. Figure 4.9 shows the MCC values of the window sizes from 25 to 500 with different learning rates, and the exact values are shown in Table 4.5. Window size 75 and learning rate $5 \cdot 10^{-6}$ have the highest MCC of 0.4683. All window sizes except $w = 500$ perform relatively well with the learning rate of $5 \cdot 10^{-6}$ and the results decrease as the learning rate becomes lower.

K-mer	Window	Accuracy	MCC	F1-score	Precision	Recall
3	10	0.8889	0.1243	0.9409	0.892	0.9954
	25	0.8953	0.3105	0.9433	0.9086	0.9809
	50	0.8958	0.4219	0.9421	0.9293	0.9553
	75	0.9015	0.4683	0.9451	0.9359	0.9545
	100	0.9023	0.4339	0.9461	0.9271	0.9659
	125	0.9005	0.4048	0.9454	0.9222	0.9698
	200	0.8917	0.3258	0.9408	0.9136	0.9698
	500	0.8881	-0.0029	0.9407	0.8883	0.9996
4	75	0.901	0.4	0.9458	0.9207	0.9722
	100	0.9004	0.4181	0.9452	0.9251	0.9661
6	75	0.8981	0.3898	0.9441	0.9209	0.9685
	100	0.9001	0.4172	0.9449	0.9252	0.9656

Table 4.4: Window size results from the third iteration.

The training loss in Figure 4.8 showing learning rate $5 \cdot 10^{-6}$ and $w = 75$ indicates that the network ends up in some unfortunate state in the middle of the training, resulting in the "one label model" and MCC $= 0$. Considering the trend with windows $100, 125$ and 200 having their best MCC with learning rate $5 \cdot 10^{-6}$, we can argue that it is possible for $w = 75$ to also reach better results with that learning rate if that unfortunate state can be avoided. Changing the random seed defined in the DNABERT repository is worth trying. Nevertheless, the learning rate $5 \cdot 10^{-6}$ seems to be more robust and therefore justifiable.

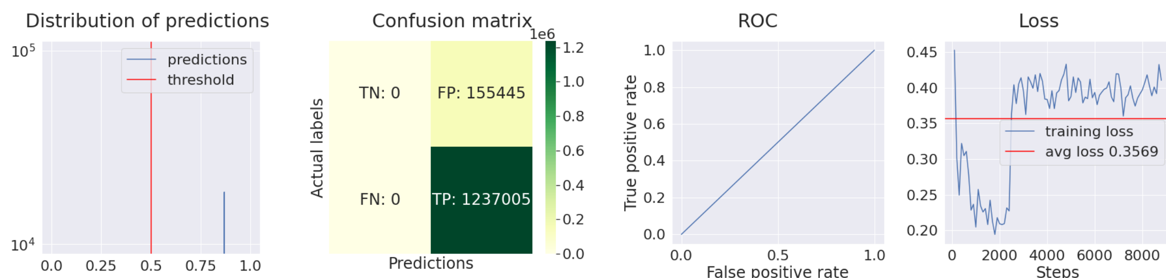


Figure 4.8: $k = 3, \text{epochs} = 3, w = 75, lr = 10^{-4}, \text{dropout} = 0.06, \text{weight decay} = 0.01, \text{warmup} = 0.06$.

Summary

The model fine-tuned with window size 75 and learning rate $= 5 \cdot 10^{-6}$ achieved the most promising results. Other parameters were $k = 3$, epochs $= 3$, dropout $= 0.06$, weight decay $= 0.01$ and warmup percent $= 0.06$. The parameters are nearly the same

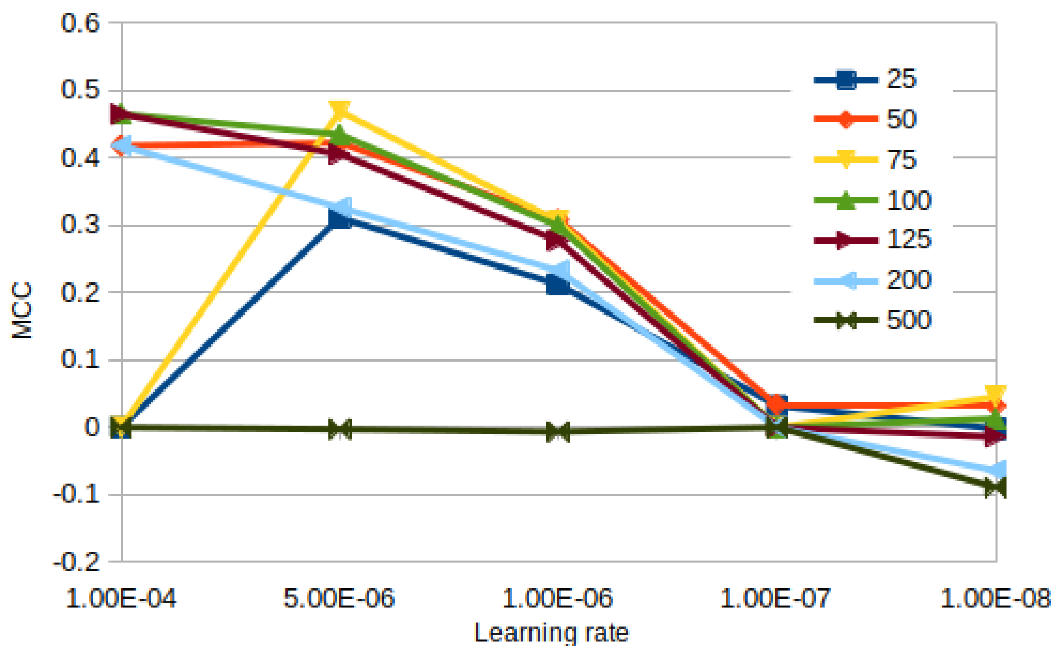


Figure 4.9: MCC values from the fourth iteration, showing different learning rates with a range of window sizes.

as recommended by the developers of DNABERT, even when the fine-tuning tasks are rather different when comparing their dna-prom detection and our gene finding task. Furthermore, the *E. coli* data is different from human DNA used in dna-prom detection.

Despite these differences, the pre-trained DNABERT model seems to adapt to our task relatively well, achieving an accuracy of 90.15%, MCC 0.4683, $F1$-score 0.9451 and precision 0.9359 when fine-tuned with the setting of training parameters mentioned above. The second highest MCC, 0.4646, was achieved by training with window size 100 and learning rate 10^{-4}, other parameters staying the same. Training times are also worth mentioning in this context: The best performing models are fine-tuned in about 20 minutes in the Colab environment mentioned in 4.2.

GeneMarkS results

As a reference system, we have the HMM based system, GeneMarkS [3]. It gives the predicted start and end coordinates for the genes. Usually its performance is measured in a gene-level, meaning that we examine the rate between correctly predicted coordinate pairs and the number of genes in the *E. coli*, 4665. With this approach,

Window	Learning rate				
	1E-04	5E-06	1E-06	1E-07	1E-08
25	0	0.3105	0.2118	0.0319	-0.0019
50	0.4176	0.4219	0.308	0.0336	0.0317
75	0	0.4683	0.3056	0	0.045
100	0.4646	0.4339	0.2985	0	0.013
125	0.4642	0.4048	0.2764	0	-0.0141
200	0.4176	0.3258	0.2324	0	-0.0642
500	0	-0.0029	-0.0067	0	-0.0891

Table 4.5: MCC results from the fourth iterations' grid search showing ranges of learning rates and window sizes.

GeneMarkS gives[1] total of 4269 predictions, from which 3824 match a ground truth genes. Therefore, 10.42% of the predictions are false, but 81.97% of the ground truth genes were correctly predicted.

However, these numbers are not comparable to the metrics in our method, since it does not give coordinates for the genes. Instead, we have to examine the nucleotide level labels, and therefore the *E. coli* genome was labeled according to the predicted coordinates. When compared to the ground truth labels, we get the following values:

- TP: 4024733
- TN: 494751
- FP: 25226
- FN: 96943
- Accuracy: 0.9737
- MCC: 0.8776
- F1-score: 0.9850
- Precision: 0.9938
- Recall: 0.9765

These values are calculated from the entire *E. coli* genome, whereas for our system, the data was divided into training and evaluation data. However, the GeneMarkS

[1]Experiments were executed in http://exon.gatech.edu/GeneMark/genemarks.cgi website 23th of March, 2021.

evaluation values were almost the same regardless of whether all or part of the genome was used. We can see that the GeneMarkS results are superior with this evaluation method. The explanation lies in the high proportion of genes compared to non-genes, combined with a large number of correctly predicted gene coordinates.

4.4.2 Secondary results

As we recall from Section 2.1, genes can be read either forward or backward from the genome. Attempts were made to modify the *E. coli* data towards a better balance between genes and non-gene sequences by labeling genes from only one direction and ignoring the coordinates of the other direction. This balancing method is explained in detail in Section 4.1.4. However, the models showed no signs of converging during three epochs, and accordingly the results were not promising. We can speculate that the characteristics of forward and reverse genes might be too similar, causing confusion: The model can not learn the difference between gene and non-gene sections if both labels contain similar patterns.

We used $k = 6$ and window size 75 for these experiments. ROC curves from both forward and reverse datasets are shown in Figure 4.10(a). Almost diagonal lines for forward and reverse datasets implicate that the models are comparable to a random or worse model. As a comparison, a higher ROC curve for unbalanced data with the same k and w implicates a better model performance. To mention some metrics, accuracy and MCC for the forward model were 0.5108 and 0.03244, respectively.

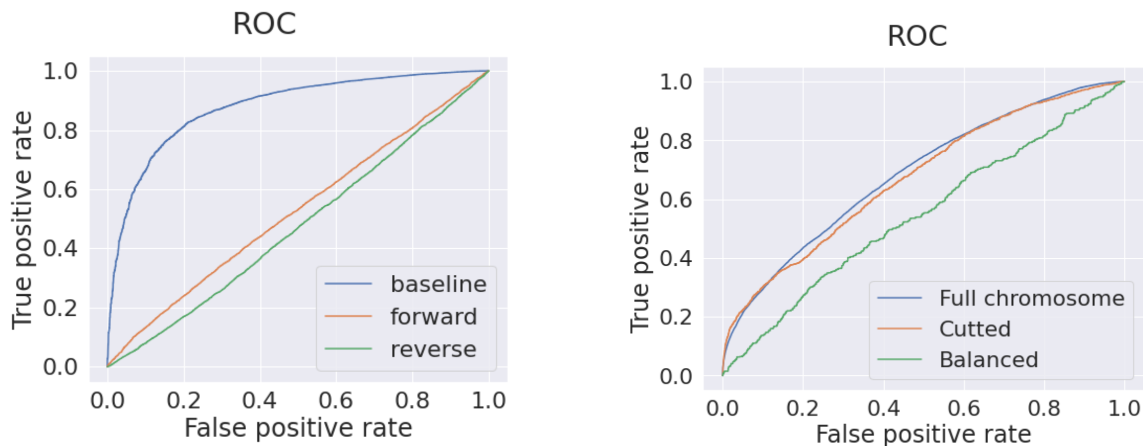
The experiments with human chromosome 18 used window size 75 and both 6-mers and 3-mers. A single model was trained with the full chromosome, but for other experiments, the data was cut according to the explanation in Section 4.1.4. The training with full chromosome lasted several orders of magnitude longer than with the cut dataset, but the results did not differ much. The ROC curves in Figure 4.10(b) demonstrate how the performance of the model did not reach the average level of *E. coli* performance.

The same figure also shows how the results declined when the data was balanced. The data was modified by removing the non-coding parts between exons, as explained in Section 4.1.4. However, this approach seems to degrade the quality of the data to the extent that the model is unable to take advantage of it.

In human chromosome 18, the imbalance between exons and other sequences is even higher than with *E. coli*, but the ratio is vice versa: Only 5% of the nucleotides belong to an exon. This ratio might be causing the poor performance of the preliminary experiments with the human chromosome. Moreover, the approach to labeling only exons may be too naive because the structure of human genes is more complex than

this[1].

While the unbalanced *E. coli* data seemed the most promising in terms of the results and the model performance, the other dataset variations were not investigated further.



(a) ROC curves of results from balanced *E. coli* datasets compared to a better performing, unbalanced model.

(b) Human dataset variations: Full chromosome with no balancing efforts, and the cut dataset as unbalanced and balanced versions.

Figure 4.10: The ROC-curves of the experiments with other dataset variations illustrated.

[1]Returning to Section 2.1, eukaryotic genes also have intron sequences between the encoding exons.

5. Conclusions and discussion

This chapter starts with a summary of the results and findings, suggesting that using the NLP-based Transformer network is a promising tool for the gene prediction task. However, there are limitations in the methods and datasets used in this study. These are discussed in detail in the second section. The last section presents the thoughts for the follow-up research approaches.

5.1 Summary

The motivation for this thesis was to study the potential of Transformer neural networks, commonly used in the NLP field, in the analysis of genome data. Detecting genes from genomes generally requires deep knowledge about genetics, but our approach was not based on human expertise. Instead, the intention was to examine whether a neural network can learn the underlying semantics of a genome and typical gene locations from labeled input examples. Our approach was to use a pre-trained model to save time and other resources and fine-tune it with a genome of the *E. coli* bacterium and the human chromosome 18.

Indeed, given 20-minute training times, the fine-tuning approach seems practical. In addition to that, the result values are promising. Compared to the naive baseline models' 88.8% accuracy, our best models achieved accuracies over 90% with a greatly unbalanced dataset. Precision increased even more than MCC, from 0.88 by the baseline model to 0.9359 using our best performing model.

Correct classification of minority labels, non-genes in the case of *E. coli*, is difficult given the unbalanced ratio of the labels. The model has to be particularly confident to make predictions against a strong prior probability of the majority label. Yet our best model predicts more TN labels than FP, being confident in this sense. Thus we can argue that the model makes predictions based on what it has learned rather than just random guesses.

In addition to evaluating the accuracy, the MCC was used as the main metric due to its ability to evaluate unbalanced data. The baseline model has zero as False positive and False negative counts. Therefore, the denominator of the MCC formula

would be zero and MCC is set to 0, as explained in Section 4.3. For this reason, it is not directly comparable with the MCC = 0.4683 achieved by our best model. However, MCC values lower than 0 are worse than a random model, and values higher than 0 are better than random guesses. From that perspective, our results are somewhat promising and indicate that Transformer networks have potential in gene detecting tasks and could therefore be a beneficial tool in the annotation of new genomes.

5.2 Discussion

The methods and data used for this research have limitations and concerns, and those are discussed in this section. The challenges include problems with the chosen dataset as well as the window and *k*-mer settings used for the training data. Another challenge is the chosen method in respect to the task.

The *E. coli* genome has a significant imbalance between genes and non-genes, as mentioned earlier in Section 4.1.4. Therefore interpreting the results is challenging due to the fact that over 88% accuracy is already achieved by assigning the same label to all inputs. Separating the relevant and insignificant differences between the models cannot be done by just comparing the accuracy, but requires other techniques like the MCC and the ROC curve. Additionally, careful examination of the confusion matrix is essential. It reveals that often in the predictions, a great proportion of non-gene nucleotides are marked as genes (FP count), and many nucleotides predicted as non-gene actually belong to a gene (FN count). Even in the best *E. coli* models, the proportion of True negatives is only 10% greater than False positives, and 17% greater than the False negatives count. By comparison, there are over 14 times as many True positives as False positives.

In addition to challenges with relevant results, the problem of the "one label model", where the model learns to predict only one value to every sequence, might be due to data imbalances. With some other genome with better balance, the model might be more robust and unlikely to end up in such an unproductive state that causes failures in the learning process. The example data in DNABERT prom-core[1] has an even ratio between the labels, and it was not reported to have such problems. However, using the *E. coli* genome is justified with its easy availability and widespread usage in research, and the simplicity of the bacterial gene structure.

Experiments with human chromosome 18 have similar challenges regarding the balance of the data, only the proportions are opposite and label 0 is dominant. In addition to the balance between labels, overlapping gene sections are a potential limitation

[1] Available in the DNABERT repository [35].

when trying to predict gene sections. There are 4665 genes in the *E. coli* genome, and 902 of them overlap with other genes, being either partly or entirely inside another gene sequence. Our model only gives labels at the nucleotide level, predicting whether they belong to any gene. In situations where gene sequences overlap with each other, one nucleotide can belong to two or even more genes, and our model does not have the capability of determining that.

Our experiments with the *E. coli* genome were mainly using the 3-mer representation, although other k-mers were also tried. This choice can be justified by the fact, that codons in genes consist of three nucleotides[1]. However, drawing precise conclusions about the best-performing k-mer size would require further experiments, such as grid search using various sets of parameters with each k-mers.

The chosen length of sequences, the window size, affects how accurate the predictions can be, in theory. The shorter the window, the closer to the nucleotide level the predictions can be. On the other hand, it can be argued that the Attention mechanism in Transformer networks benefits from a reasonable amount of tokens in one input sequence, to be able to calculate the connections between them. The details of the Attention mechanism are explained earlier in Subsection 3.2.2. Predicting genes with DNABERT appears to be a balancing act between short enough windows to avoid long mislabeled sequences, and long enough windows to give the network the best opportunity to learn the characteristics of typical gene and non-gene sequences.

The DNABERT Prom-core version used as a pre-trained model requires the input as a sequence and predicts the labels at the sequence level. When trying to predict gene sequences of a genome without knowing the ground truth, the genome has to be divided into sequences blindly, and some of them inevitably contain both gene and non-gene parts. Therefore the result are not the exact coordinates of genes, but rather the estimated locations for genes. To determine the exact start and end nucleotides, a specialist in genomics has to review the predictions and trim the labels, but the predictions can potentially serve as suggestions or starting points for the specialist.

Another limitation regarding the use of DNABERT in this thesis was the limited visibility on how the network actually works. Neural networks are somewhat black boxes overall, as we do not know how exactly the network forms its knowledge about the given data. Additionally, when using a pre-trained model like DNABERT Prom Core, the data and parameters used for pre-training are not necessarily the most optimal for our task. But again, advantages of pre-trained models include savings in time, computing power and emissions, like already mentioned in Section 3.3, and therefore we did not pre-train our own model.

Though their open source contribution is highly respected, there are some chal-

[1]Revisit Section 2.1 for the explanation of codons.

lenges with using the code-base of DNABERT. There is no clear documentation of which of the parameters can be modified and how the evaluation metrics are calculated. Studying the source code to find those answers can be time-consuming. On the other hand, implementing one's own Transformer-based neural network for genome data would also require a lot of time and effort.

5.3 Future work

The imbalance of the gene and non-gene sections leads to many challenges, as discussed in the previous section. One solution is to oversample the minority sections or undersample the majority sections using approaches other than those already tried. Adding copies of non-gene sections or removing gene sequences, or parts of them, might affect the semantics of the genome and therefore the networks' ability to learn the characteristics of gene locations. Another way to affect the imbalance would be to optimize the weights of the classes and thus raise the visibility of the minority class, for example the non-gene sequences in the *E. coli* genome.

DNABERTs visualization feature can help to better evaluate the model's level of understanding over the input genome sequences. It would be interesting to see whether the model emphasizes some k-mers over others, and how the attention values are distributed within the windows. However, interpreting those visualizations would potentially require deeper knowledge about genomics. Another idea for determining how well the model is learning would be to visualize how the predicted values are distributed around the gene start and end locations. Are they closer to the threshold 0.5 in those areas, compared to the center of the gene and non-gene sequences?

One of the limitations of our method is its inaccuracy in predicting the start and end coordinates of genes. Therefore, other approaches than sequence-based methods should be considered for future experiments with gene detecting tasks. DNABERT and Transformer networks seem to be able to understand the characteristics of genomes to some extent, based on both this thesis and the original DNABERT results [18], but the challenge is to find the best way to benefit from that understanding in this specific task of finding genes. An interesting basis for further experiments with DNABERT would be to increase the number of labels based on whether the window is located in the beginning, middle or end of the gene. Using overlapping windows as input sequences can also be considered, in either learning or when using the model for predicting.

Using genomes from multiple bacteria for fine-tuning would show the generalizing abilities of DNABERT and Transformer networks. The genetic structure of eukaryotes is more complex than that of prokaryotes, but experiments with human chromosomes and genomes is one interesting avenue for follow-up research.

Bibliography

[1] Jay Alammar. The illustrated transformer, Jun 2018. http://jalammar.github.io/illustrated-transformer/, Accessed on 20th March 2021.

[2] Babak Alipanahi, Andrew Delong, Matthew T Weirauch, and Brendan J Frey. Predicting the sequence specificities of DNA-and RNA-binding proteins by deep learning. *Nature biotechnology*, 33(8):831–838, 2015.

[3] John Besemer, Alexandre Lomsadze, and Mark Borodovsky. GeneMarkS: a self-training method for prediction of gene starts in microbial genomes. Implications for finding sequence motifs in regulatory regions. *Nucleic Acids Research*, 29(12):2607–2618, Jun 2001.

[4] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.", 2009.

[5] Sabri Boughorbel, Fethi Jarray, and Mohammed El-Anbari. Optimal classifier for imbalanced data using Matthews Correlation Coefficient metric. *PloS one*, 12(6):e0177678, 2017.

[6] Terence A Brown. *Genomes, 2nd edition.* Oxford: Wiley-Liss, https://www.ncbi.nlm.nih.gov/books/NBK21134/, 2002.

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[8] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. What does BERT look at? An analysis of BERT's Attention. *arXiv preprint arXiv:1906.04341*, 2019.

[9] Jim Clauwaert and Willem Waegeman. Novel transformer networks for improved sequence labeling in genomics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 19(1):97–106, 2022.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[11] Robert D Fleischmann, Mark D Adams, Owen White, Rebecca A Clayton, Ewen F Kirkness, Anthony R Kerlavage, Carol J Bult, Jean-Francois Tomb, Brian A Dougherty, Joseph M Merrick, et al. Whole-genome random sequencing and assembly of Haemophilus influenzae Rd. *Science*, 269(5223):496–512, 1995.

[12] National Center for Biotechnology Information. Homo sapiens (human) genome assembly GRCh38, 2019. https://www.ncbi.nlm.nih.gov/assembly/GCF_000001405.26/, Accessed on 29th April 2021.

[13] John Fuerst. Beyond prokaryotes and eukaryotes: Planctomycetes and cell organization. *Nature Education*, 3(9):44, Jan 2010.

[14] Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. Transformer feed-forward layers are key-value memories. *arXiv preprint arXiv:2012.14913*, 2020.

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

[16] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[17] Sungchul Ji. The linguistics of DNA: words, sentences, grammar, phonetics, and semantics. *Linguistics*, 1999.

[18] Yanrong Ji, Zhihan Zhou, Han Liu, and Ramana V Davuluri. DNABERT: pre-trained Bidirectional Encoder Representations from Transformers model for DNA-language in genome. *Bioinformatics*, 37(15):2112–2120, 2021.

[19] Daniel Jurafsky and James H Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, 2nd edition*. Pearson Education, 2009.

[20] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *ArXiv e-prints*, pages arXiv–1607, 2016.

[21] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2019.

[22] Catherine Mathé, Marie-France Sagot, Thomas Schiex, and Pierre Rouzé. Current methods of gene prediction, their strengths and weaknesses. *Nucleic Acids Research*, 30(19):4103–4117, Oct 2002.

[23] Brian W Matthews. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442–451, 1975.

[24] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[25] Staffan Normark, Sven Bergström, Thomas Edlund, Thomas Grundström, Bengtake Jaurin, Frederik P Lindberg, and Olof Olsson. Overlapping genes. *Annual review of genetics*, 17(1):499–525, 1983.

[26] OpenAI. Dall•E: Creating images from text, Jan 2021. https://openai.com/blog/dall-e/, Accessed on 9th February 2022.

[27] Liam Porr. A robot wrote this entire article. are you scared yet, human?, 2020. https://www.theguardian.com/commentisfree/2020/sep/08/robot-wrote-this-article-gpt-3, Accessed 4th May 2021.

[28] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[29] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 8821–8831. PMLR, Jul 2021.

[30] RegulonDB. E. coli k-12 genome sequence, 2019. https://regulondb.ccg.unam.mx/menu/download/datasets/index.jsp, Accessed on 3rd February 2021.

[31] Alberto Santos-Zavaleta, Heladia Salgado, Socorro Gama-Castro, Mishael Sánchez-Pérez, Laura Gómez-Romero, Daniela Ledezma-Tejeida, Jair Santiago

García-Sotelo, Kevin Alquicira-Hernández, Luis José Muñiz-Rascado, Pablo Peña-Loredo, et al. RegulonDB v 10.5: tackling challenges to unify classic and high throughput knowledge of gene regulation in E. coli K-12. *Nucleic Acids Research*, 47(D1):D212–D220, 11 2018.

[32] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green AI. *Communications of the ACM*, 63(12):54–63, Nov 2020.

[33] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. Big data: astronomical or genomical? *PLoS biology*, 13(7):e1002195, 2015.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[35] Han Liu Ramana V Davuluri Yanrong Ji, Zhihan Zhou. DNABERT GitHub repository. https://github.com/jerryji1993/DNABERT, Commit a62e217, Accessed on 20th May 2021.

[36] Yongqing Zhang, Shaojie Qiao, Rongzhao Lu, Nan Han, Dingxiang Liu, and Jiliu Zhou. How to balance the bioinformatics data: pseudo-negative sampling. *BMC bioinformatics*, 20(25):1–13, 2019.

Appendix A. Fine-tuning

The fine-tuning was executed in Colab environment using the DNABERT code from the GitHub repository. The only modification required to the code-base was to change one import from line 35 in `run_finetune.py` as follows:

```
# Before:
from tqdm import tqdm
# After:
from tqdm.notebook import tqdm
```

Without the modification, unnecessary log lines were printed during the training. Adding `notebook` to the import allowed us to follow the loss and learning rate changes more easily.

Below is an example script for executing `run_finetune.py` command. Option `do_train` is selected in the fine-tuning step, and the prediction is executed with the `do-eval` option.

```
!python run_finetune.py \
    --model_type dna \
    --tokenizer_name=dna3 \
    --model_name_or_path "3-new-12w-0" \
    --task_name dnaprom \
    --do_train \
    --data_dir "../../drive/MyDrive/dnabert/data_ecoli" \
    --max_seq_length 75 \
    --per_gpu_eval_batch_size=16 \
    --per_gpu_train_batch_size=16 \
    --learning_rate 1e-6 \
    --num_train_epochs 3.0 \
    --output_dir "../../drive/MyDrive/dnabert/model_ecoli" \
    --logging_steps 100 \
    --save_steps 60000 \
    --warmup_percent 0.06 \
    --hidden_dropout_prob 0.1 \
    --overwrite_output \
    --weight_decay 0.01 \
    --n_process 8
```

Appendix B. Preprocessing and evaluation scripts

The data was loaded and preprocessed with Python functions using Jupyter Notebook. The attached file shows the functions for processing the *E. coli* source files, the genome and the gene coordinates, before the fine tuning. Preprocessing of human chromosome 18 followed the similar process with minor adjustments due to the differences in the data format.

After the preprocessing functions, the notebook shows the functions developed for evaluating the predicted results.

ecoli_preprocess_and_evaluate

February 9, 2022

1 Collecting and processing data

[1]:
```
import numpy as np
from collections import Counter
import random
import matplotlib.pyplot as plt
import seaborn as sn
import pandas as pd
import sklearn.metrics as metrics
from PIL import Image
```

[2]:
```
# Gene coordinates (ground truth)
gt_dir = "ecoli_data/regulonDB/txt/gene.txt"
# Genome sequence
ecoli_dir = "ecoli_data/fasta/E_coli_K12_MG1655_U00096.3.txt"
```

[4]:
```
# Ground truth data with gene coordinates:
# 1. Read the file
gt_data = []
with open(gt_dir, 'r') as f:
    gt_lines = f.readlines()
    for line in gt_lines:
        if line[0] != '#':
            gt_data.append(line.split("\t"))

print(f'Total amount genes in the original: {len(gt_data)}') #4686

# 2. Put in order
def order(elem):
    r = elem[0]
    if len(r) > 0:
        return int(r)
    else:
        return 0

# Choose the needed values
```

```
gt_data_ordered = [d[2:6] for d in gt_data if len(d[2])>0] # Genes with no␣
↪coordinates are left out
gt_data_ordered.sort(key=order)
print(f'Genes with no coordinates removed, total amount now:␣
↪{len(gt_data_ordered)}')

# 3. Separate sets for forward and reverse genes
gt_data_forward = [d for d in gt_data_ordered if d[2]=="forward"]
gt_data_reverse = [d for d in gt_data_ordered if d[2]=="reverse"]
print(f'Forward genes: {len(gt_data_forward)}')
print(f'Reverse genes: {len(gt_data_reverse)}')
```

```
Total amount genes in the original: 4686
Genes with no coordinates removed, total amount now: 4665
Forward genes: 2297
Reverse genes: 2368
```

[5]:
```
# Read the genome to one string
with open(ecoli_dir, 'r') as f:
    ecoli_lines = f.readlines()[1:]
    line_length = len(ecoli_lines[50])
    ecoli_lines = [l[:line_length-1] for l in ecoli_lines]

print(f'{len(ecoli_lines)} lines')

ecoli = ''
for line in ecoli_lines:
    ecoli += line
print(f'{len(ecoli)} nucleotides in ecoli genome')
```

```
61889 lines
4641653 nucleotides in ecoli genome
```

[8]:
```
# Assigns labels according to the gene coordinates
def label_genome(genome, gt_data):
    coordinates = [(int(c[0]), int(c[1])) for c in gt_data]
    labels = np.zeros(len(genome))
    for i, c in enumerate(coordinates):
        # Set all gene labels to 1
        labels[c[0]-1:c[1]] = 1
    c = Counter(labels)
    print(f'Zeros account of the total labels: {c[0]/(c[0]+c[1]):.4f}')
    return labels

test_genome = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
test_gt = [(3,8), (11,15), (14,19)]
label_genome(test_genome, test_gt)
```

```
Zeros account of the total labels: 0.4231
```

[8]: array([0., 0., 1., 1., 1., 1., 1., 1., 0., 0., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 0., 0., 0., 0., 0., 0., 0.])

[9]:
```
# Make train and test sets from the genome and ground truth coordinates
def get_train_and_test(genome, gt_data, train_size=.7):
    labels = label_genome(genome, gt_data)
    idx = int(len(labels)*train_size)
    return genome[:idx], labels[:idx], genome[idx:], labels[idx:]
```

[11]:
```
# Genomes and labels for test and train sets
train_genome, train_labels, test_genome, test_labels = get_train_and_test(ecoli,␣
 ↪gt_data_ordered)
print(f'Length of train sequence: {len(train_genome)}')
print(f'Length of test sequence: {len(test_genome)}')
```

```
Zeros account of the total labels: 0.1120
Length of train sequence: 3249157
Length of test sequence: 1392496
```

[12]:
```
# Forming the k-mer representations for a nucleotide sequence
def make_kmers(data, k, sliding_window=True):
    if len(data)%k != 0 and not sliding_window:
        print('Check that seq length is 0 mod k')
    output = ''
    if sliding_window:
        for i in range(0,len(data)-k+1):
            output += data[i:i+k]
            output += ' '
    else:
        for i in range(0, len(data)-k+1, k):
            output += data[i:i+k]
            output += ' '
    return output[:-1]

d = 'ATGAAACGCATTAGCACCACCATTACCACCACCATCACCATTACCACAGGTAACGGTGCGGGCTGACC'
make_kmers(d, 6)
```

[12]:
```
'ATGAAA TGAAAC GAAACG AAACGC AACGCA ACGCAT CGCATT GCATTA CATTAG ATTAGC TTAGCA
TAGCAC AGCACC GCACCA CACCAC ACCACC CCACCA CACCAT ACCATT CCATTA CATTAC ATTACC
TTACCA TACCAC ACCACC CCACCA CACCAC ACCACC CCACCA CACCAT ACCATC CCATCA CATCAC
ATCACC TCACCA CACCAT ACCATT CCATTA CATTAC ATTACC TTACCA TACCAC ACCACA CCACAG
CACAGG ACAGGT CAGGTA AGGTAA GGTAAC GTAACG TAACGG AACGGT ACGGTG CGGTGC GGTGCG
GTGCGG TGCGGG GCGGGC CGGGCT GGGCTG GGCTGA GCTGAC CTGACC'
```

[13]:
```
# Train sequences and test sequences are made differently
# Train data is first splitted by labels, and then every sequence is transformed␣
 ↪into k-mers
# and then into sequences of desired window size (seq_len)
```

```
def make_train_sequences(genome, labels, seq_len, k=6):
    seq_len = seq_len-k+1
    labels = np.split(labels, np.where(np.diff(labels[:]))[0]+1)
    seqs = []
    zeros = 0
    idx = 0

    for l in labels:
        l_len = len(l)
        idx+=l_len
        # Sequences shorter than k are left out (tokenizer does not have tokens␣
↪for them)
        if l_len >= k:
            g = genome[idx-l_len:idx]
            kmers = make_kmers(g, k, sliding_window=True).split(' ')

            for i in range(0, l_len-1, seq_len):
                seq = ' '.join(kmers[i:i+seq_len])
                label = int(l[0])
                if label == 0:
                    zeros+=1
                line = ('{}{}{}{}'.format(seq, '\t', label, '\n'))
                seqs.append(line)

    print(f'Zeros account of the total labels in the train set:  {zeros/
↪len(seqs)}')
    return seqs
```

[14]:
```
# Test data is first splitted according to desired window size (seq_len).
def make_test_sequences(genome, labels, seq_len, method=1, k=6):
    seqs = []
    zeros = 0
    for i in range(0,len(genome)-seq_len+1, seq_len):
        seq = make_kmers(genome[i:i+seq_len], k, sliding_window=True)

        # Only sequences fully inside the gene are labeled as 1
        if method==2:
            if sum(labels[i:i+seq_len])==seq_len:
                label=1
            else:
                label=0
                zeros+=1
        # If one or more nucleotides in sequence are inside the gene, label as 1
        elif method==3:
            if sum(labels[i:i+seq_len])>0:
                label=1
            else:
```

```
                label=0
                zeros+=1
        # If more than half of nucleotides are inside the gene, label as 1
        else:
            if sum(labels[i:i+seq_len])>seq_len/2:
                label=1
            else:
                label=0
                zeros+=1
        line = ('{}{}{}{}'.format(seq, '\t', label, '\n'))
        seqs.append(line)
    print(f'Zeros account of the total labels in test set:  {zeros/len(seqs)}')
    return seqs
```

[15]:
```
# Write the .tsv files in a desired path
def write_test_and_dev_files(train_genome,
                             train_labels,
                             test_genome,
                             test_labels,
                             seq_len,
                             path,
                             k=6,
                             train_size=0.7,
                             method=1,
                             shuffle=True,
                             sliding_w=False):
    train_data = make_train_sequences(train_genome, train_labels, seq_len, k)
    test_data = make_test_sequences(test_genome, test_labels, seq_len, method, k)
    print(f'Train sequences: {len(train_data)}')
    print(f'test sequences: {len(test_data)}')
    header = ['sequence  label\n']
    if shuffle:
        np.random.seed(123)
        np.random.shuffle(train_data)
    train = header + train_data
    test = header + test_data
    train_dir = path + 'train.tsv'
    test_dir = path + 'dev.tsv'

    with open(train_dir, 'w') as f_output:
        for line in train:
            f_output.write(line)

    with open(test_dir, 'w') as f_output:
        for line in test:
            f_output.write(line)
```

```
    return train_data, test_data
```

[16]:
```
# Function for making multiple datasets
def make_datasets(windows, k, method=1):
    for w in windows:
        print(f'window={w}, k={k}')
        trains, tests = write_test_and_dev_files(
            train_genome,
            train_labels,
            test_genome,
            test_labels,
            w, f'ecoli_data/{k}/method{method}/{w}/', k=k
        )
        globals()[f'train_{k}_labels_{w}'] = trains
        globals()[f'test_{k}_labels_{w}'] = tests
```

Next steps: Copy train.tsv and dev.tsv files to drive, and run the run_finetune.py with Colab using –do_train option. To get predictions as the pred_results.npu -file, run the script with –do_predict option. The file is used for the evaluation.

2 Evaluation

[17]:
```
# Change the predictions from floats in range [0,1] into labels 0 or 1
def make_predicted_labels(data, seq_len, th=0.5):
    labels = []
    probabs = []
    for d in data:
        if d>th:
            for i in range(seq_len):
                labels.append(1)
                probabs.append(d)
        else:
            for i in range(seq_len):
                labels.append(0)
                probabs.append(d)
    return labels, probabs
```

[18]:
```
# Visualize the distribution of predicted values.
def plot_distribution(data, th):
    vals = [int(v*1000) for v in data]
    counts = Counter(vals)
    keys = counts.keys()
    values = counts.values()

    c = np.zeros(1000)
    for key, value in counts.items():
```

```
        c[key]=value
    fig = plt.figure(figsize = (6,6))
    plt.plot(c, label='predictions')
    plt.yscale('log')
    fig.suptitle('Distribution of predictions')
    plt.axvline(x=th*1000, color='red', label='threshold')
    plt.legend(['predictions', 'threshold'])
    locs, labels = plt.xticks()  # Get the current locations and labels.
    stp=1/(len(locs)-2)
    lbls = [x/1000 for x in locs]
    plt.xticks(ticks=locs[1:len(locs)-1], labels=lbls[1:len(lbls)-1])
    plt.savefig('figures/plots/distribution.png')
```

[19]:
```
# Visualize the TP, TN, FP and FN counts
def plot_confusion(tp,tn,fp,fn):
    results = np.array([[tn,fp],
            [fn,tp]])
    df_cm = pd.DataFrame(results)
    strings = np.asarray([['TN', 'FP'],
                         ['FN', 'TP']])
    labels = (np.asarray(["{0}: {1}".format(string, value)
                      for string, value in zip(strings.flatten(),
                                               results.flatten())])
         ).reshape(2, 2)

    fig = plt.figure(figsize = (7,6))
    sn.set(font_scale=2)
    sn.heatmap(df_cm, annot=labels, fmt="", cmap="YlGn", xticklabels=False,␣
 ↪yticklabels=False)
    fig.suptitle('Confusion matrix')
    plt.xlabel("Predictions")
    plt.ylabel("Actual labels")
    plt.savefig('figures/plots/confusion.png')
```

[20]:
```
# Show Matthews correlation coefficient (MCC curve
def plot_roc(gt_labels, probas):
    fpr, tpr, threshold = metrics.roc_curve(gt_labels, probas)
    fig=plt.figure(figsize = (6,6))

    plt.plot(fpr, tpr)
    fig.suptitle('ROC')
    plt.xlabel('False positive rate')
    plt.ylabel('True positive rate')
    plt.tight_layout(pad=0.5)
    plt.savefig('figures/plots/roc.png')
```

[21]:
```
# Save loss values from the logs and read them from the file
def get_loss(path):
    data = []
    avg = 0
    with open(path, 'r') as f:
        lines = f.readlines()
        #print(lines)
        for line in lines:
            if line[0]=='{':
                line = line[:-1]
                #print(line)
                data.append(eval(line))
            elif len(line)>5:
                #print(line)
                line = line.split(' ')
                avg = float(line[-1])

    rates = [x.get('learning_rate') for x in data]
    losses = [x.get('loss') for x in data]
    steps = [x.get('step') for x in data]
    return steps, losses, avg
```

[22]:
```
# Visualize the model performation bu showing the loss values during the training
def plot_loss(path):
    steps, losses, avg = get_loss(path)

    fig=plt.figure(figsize=(6, 6))
    plt.plot(steps, losses, label='training loss')
    plt.axhline(y=avg, color='red', label='avg loss')
    plt.legend(['training loss', f'avg loss {avg:.4f}'])
    plt.xlabel("Steps")
    fig.suptitle(f'Loss')
    plt.tight_layout(pad=0.5)
    plt.savefig('figures/plots/loss.png')
    #plt.plot(steps, rates)
```

[23]:
```
# Combine and save the four images into one.
def make_image(path):
    images = [Image.open(x) for x in ['figures/plots/distribution.png',
                                      'figures/plots/confusion.png',
                                      'figures/plots/roc.png',
                                      'figures/plots/loss.png']]
    widths, heights = zip(*(i.size for i in images))
    total_width = sum(widths)
    max_height = max(heights)

    new_im = Image.new('RGB', (total_width, max_height))
```

```
    x_offset = 0
    for i, im in enumerate(images):
        new_im.paste(im, (x_offset,0))
        x_offset += im.size[0]

    new_im.save(path)
```

[24]:
```
# Calculate the evaluation metrics and show the plots
def evaluate(datapath, losspath, seq_len, test_labels, img_name, th=0.5):
    data = np.load(datapath)
    print(f'Predicted values are between {np.min(data):.4f} and {np.max(data):.
 ↪4f}.')

    pr_labels, probabs = make_predicted_labels(data, seq_len, th)
    print(f'Count of predicted labels: {len(pr_labels)}')
    print(f'Count of gt labels: {len(test_labels)}')

    # Chacking that the label lengts match
    if len(test_labels)-len(pr_labels)>seq_len:
        print('Wrong test labels!')

    # TP, TN, FP and FN:
    mismatches, tp, tn, fp, fn = 0, 0, 0, 0, 0
    for i in range(len(pr_labels)):
        if pr_labels[i] != test_labels[i]:
            mismatches+=1
        if pr_labels[i] == 1 and test_labels[i] == 1:
            tp+=1
        elif pr_labels[i] == 0 and test_labels[i] == 0:
            tn+=1
        elif pr_labels[i] == 1 and test_labels[i] == 0:
            fp+=1
        elif pr_labels[i] == 0 and test_labels[i] == 1:
            fn+=1

    # Precision and recall:
    if tp+fp == 0:
        prec = 0.5
    else:
        prec = tp/(tp+fp)
    if tp+fn==0:
        rec=0.5
    else:
        rec = tp/(tp+fn)

    # MCC:
```

```
    mcc_denom = (tp+fp)*(tp+fn)*(tn+fp)*(tn+fn)
    if mcc_denom == 0:
        mcc_denom = 1
    mcc = (tp*tn-fp*fn)/mcc_denom**.5

    print(f'Accuracy: {1-mismatches/len(test_labels):.4f}')
    print(f'MCC: {mcc:.4f}')
    print(f'F1-score: {2*prec*rec/(prec+rec):.4f}')
    print(f'Precision: {prec:.4f}')
    print(f'Recall: {rec:.4f}')

    plot_distribution(data, th)
    plot_confusion(tp,tn,fp,fn)
    plot_roc(test_labels[:len(probabs)], probabs)
    plot_loss(losspath)
    make_image('figures/plots/ecoli/'+img_name)
```