UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA BIOMEDICA

Deep Learning for genomic sequences

Relatore
Prof. Sales Gabriele

Laureando
Guarnieri Luca

ANNO ACCADEMICO 2023-2024

Data di laurea 12/03/2024

Abstract

Nowadays, thanks to advanced techniques, such as the Next Generation Sequencing (NGS), the time and costs of DNA sequencing are constantly lowering. These kinds of techniques offer high throughput, speed and scalability, so that the amount of available data on DNA sequences is greater than ever before. Nevertheless, when it comes to decoding and understanding what is encoded within this great number of sequences, there is an urgent need for new technologies, which can keep up with the data production and be able to comprehend the contextual information of genes, scattered over long sequences of DNA.

Artificial Intelligence and Deep Learning, in the field of Genomics, promise great advances in the interpretation and classification of genomic sequences. These kinds of models can learn and recognize significant genomic sequences and patterns, without the need for expensive, time-consuming, complicated wet-lab experiments. Moreover, they have been proven to do that, even when trained with a shortage of data.

This study will describe the state-of-the-art deep-learning model architecture, namely the Transformer, and how it works. Afterward, two examples of its application to the biological problem will be presented: Nucleotide Transformers and Gena-LM, both implementing advanced foundational DNA language models, capable of high performances in numerous sequence prediction tasks. These works will be described and compared.

Lastly, the aforementioned models will be tested: the fine-tuning technique will be exploited, assessing the performances of each model on different datasets.

All the results and the fine-tuned models can be found on the HuggingFace page of the author: https://huggingface.co/LiukG

Contents

Chapter 1

Introduction

In this introductory chapter, a brief overview of Genomics is presented, to better understand its peculiarities and how deep learning is so impactful in this field.

Secondly, the core of this chapter is devoted to explaining the Transformer architecture: It will be compared to former architectures, to capture the innovative features that allow this architecture to stand out, and every component will be explained in detail.

Finally, a brief introduction to foundational models is provided.

1.1 Genomics

> Genomics is a field of biology focused on studying all the DNA of an organism — that is, its genome. Such work includes identifying and characterizing all the genes and functional elements in an organism's genome as well as how they interact[1].

This is a relatively recent discipline: Its origin dates back to the late '80s, when it was first named in the newborn journal ”Genomics”. It comes from the combination of molecular and cell biology with genetics and is supported by computational science[2]. In this field, the amount of data available on DNA sequences is constantly growing, thanks to advanced techniques, such as the Next Generation Sequencing (NGS), which have been lowering the times and costs of DNA sequencing. To exploit and analyse such a huge data throughput, nothing could be more useful than a computer and that is why Bioinformatics is, nowadays, more important than ever.

Bioinformatics is going through an important transformation too; It is led by the recent methodological developments in deep learning domains such as computer vision or natural language processing, paving the way to deep genomics: the application of deep learning to genomic sequences. These techniques can now provide a lot of information: the protein's 3D structure, nucleosome and other protein bindings sites, the local accessibility of the DNA sequence, the epigenetic, the activity of genes, and much more [3]

Natural language models applied to DNA sequences are the main interest for this thesis, some examples will be provided in the following chapters.

1.2 Transformer

Transformer models made their appearance in 2017, when they were first described in the very well-known article ”Attention is all you need”, by researchers working at Google [4]. As evidenced in this article, Transformer architecture was implemented to tackle sequence elaboration, which includes translation, speech recognition, and text classification to name but a few. Many other architectures were implemented before this article, achieving high results both in vision and in natural language processing, but this work marks a turning point in the history of Deep Learning and in the way of conceiving Artificial Intelligence architectures.

Before this, neural networks used an encoder-decoder architecture. In short, the input sequence, which can be a sentence, enters the encoder’s layer, where is transformed into a more compact mathematical representation. This will be the input of the decoder, which will provide the output sequence. Two main architectures established themselves in word processing before the rise of Transformer: Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs). To better appreciate the innovations that have allowed Transformer to emerge, we will go over these two architecture’s features.

1.2.1 Recurrent Neural Networks

A Recurrent Neural Network can be imagined as the same network **H**, replicated several times, in a row. Each of these networks receives its input data from the previous one and hands its output down to the following. This is what recurrence would be, if we unroll the loop. 1.1

Applying this kind of network to sentence translation, the network processes a word at a time, adding one to the hidden state at every cycle. This is an important aspect when working with natural language: as humans, we grasp the meaning of words from their position and, more in general, from the context they are in. As far as the hidden state is conserved in the loop and upgraded with the next word cycle after cycle, the word’s order is clear and the following word is evaluated over the previous ones. When the last word is analysed, the obtained hidden state is transferred to the decoder, which generates the output. Even if Recurrent Neural Networks works very well on short sentences, this kind of architecture has two disadvantages:

1. Training such models takes a lot of time and even more, taking into consideration that, for the same reason, the training process can not be done in parallel, exploiting several computational units.

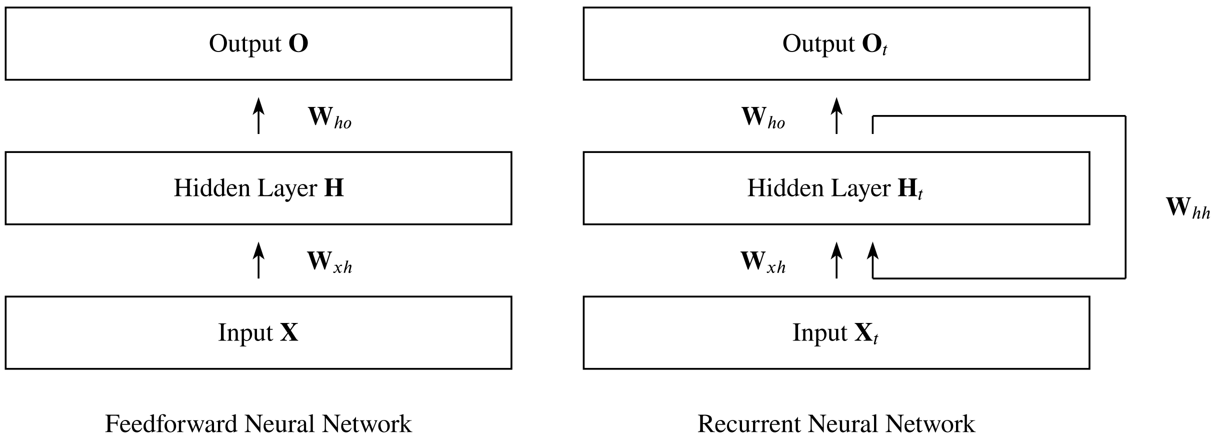

Figure 1.1: The different information's flow in feed-forward and Recurrent Neural Networks [5]

2. Recurrent Neural Networks lose efficiency and correctness when it comes to long pieces of text, as explained in the following example.

Given two sentences of the same text, one far from the other: "I worked in the Netherlands for a long period." and "Fortunately, I practiced a lot and now I speak a fluent ...". Looking at the last sentence, the model can easily predict that the next word should be a language, and yet, the information it needs, namely "Netherlands", was encoded too many words before, to be still relevant within the hidden state. This Recurrent Neural Network lacks the capability to distinguish the important information, the most important words, in this case, among all the others.

Long-short-term memory (LSTM)

Long-short-term memory network is a particular type of Recurrent Neural Network that aims to overcome the previous problem. They are equipped with the Cell State mechanism, which allows the network to selectively forget or remember information.

As you can see from the picture 1.2, each cell takes its input and the output of the previous cell, auditioned to the previous cell state, and generates a new cell state and a new output weighted on them.[5] This kind of process is more accurate than the one seen above, but it does not completely solve the problem: the probability of keeping the context given by a far-located word in a long piece of text decreases exponentially with the distance from the word that has to be analysed.

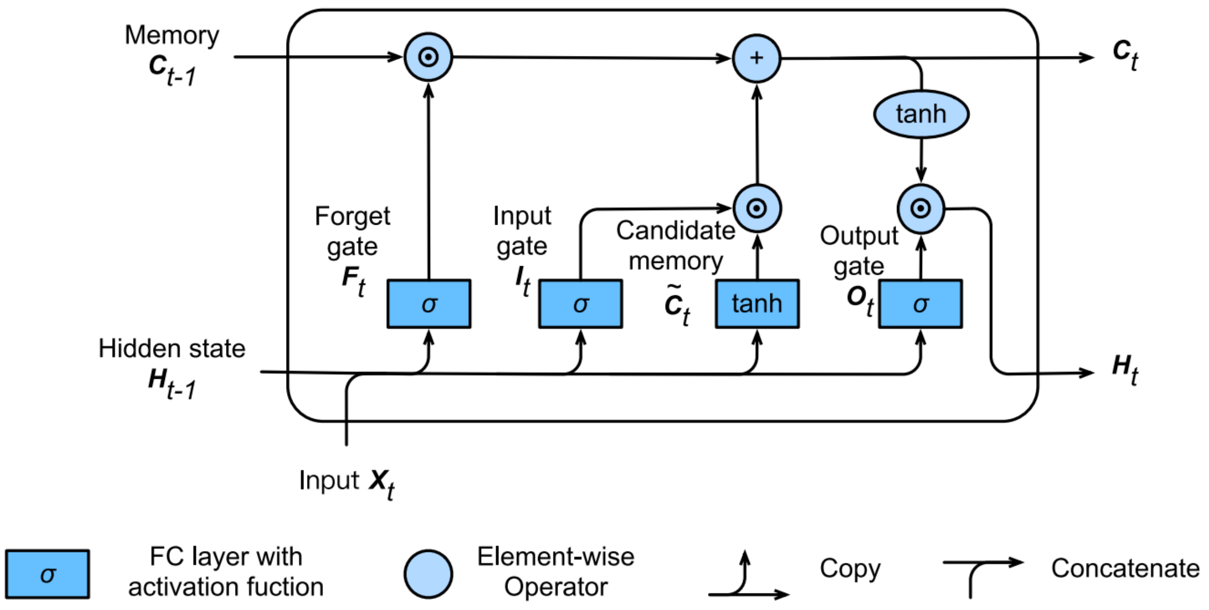

Figure 1.2: The cell state scheme [5]

Attention

To address these problems, researchers have developed 'Attention': thanks to this technique, Recurrent Neural Networks can focus on some specific keywords. In fact, every analysed word has an associated hidden state, which will passed to the decoder directly, and all of them will be used together to decode the sentence (sticking to our example).
You can look at the picture 1.3 as an example.

Now that the correctness problem has been solved, one last weakness should be enhanced, and that is time. Apart from attention, the network still analyses word after word only, which causes the network's training to last too long.

1.2.2 Convolutional Neural Networks

In order to reduce the time impact, Convolutional Neural Networks come in handy. This is a network architecture able to capture the dependencies between the inputs spatially and temporally. CNNs are made up of three different types of layers: convolutional layer, pooling layer and fully connected layer 1.4. The convolutional layer is the first layer the word embeddings must go through; it consists of a sliding window that applies a series of filters, designed to detect patterns or features. This sliding window has some parameters:

- Kernel size: that is the number of elements that the window can pay attention to, at the same time. In our example: the number of words the window can look at, in each step (typically between 2 and 5).

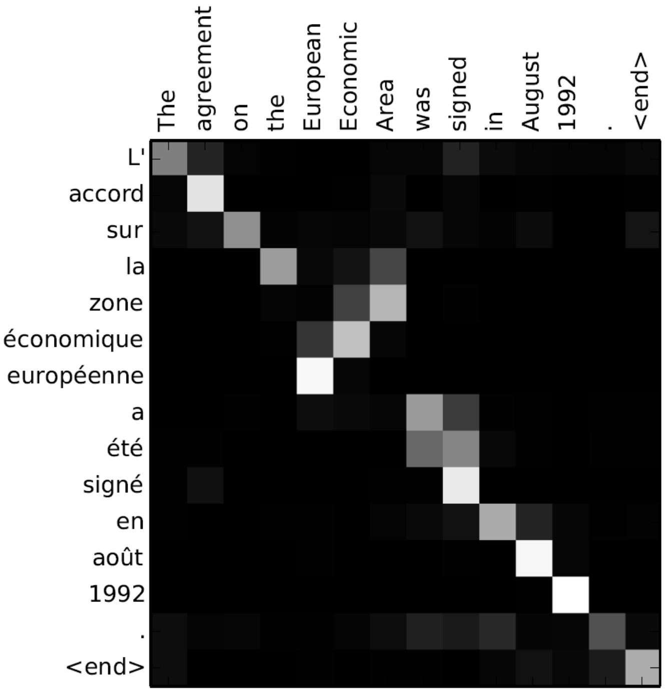

Figure 1.3: Example of an Alignment matrix of "L'accord sur la zone économique européen a été signé en août 1992" (French) and its English translation "The agreement on the European Economic Area was signed in August 1992" [5]

- Stride: this indicates how much the window moves at each step. In our example: the number of words the window proceeds

- Padding: adding zeros to both sides of the input. This is important when the Stride parameter is >1

- Bias: a bias vector simply added to the filter's matrix

The turning point we were looking for lies in this matrix, as all the filters and calculations can be computed in parallel, at each step.

Then, the pooling layer is applied, to reduce the number of parameters. Max-pooling or mean-pooling can be used: they extract the maximum or the mean of each feature's values, respectively.

Finally, a fully connected layer is implemented specifically for different tasks.[6] As far as the evaluation of the input sentence is made in parallel, words are processed at the same time, so that they do not depend on the words translated before.

To sum up, CNN can reduce running times and the impact of distance between words, but they can not still get the dependencies. Eventually, here is where the Transformer makes the difference.

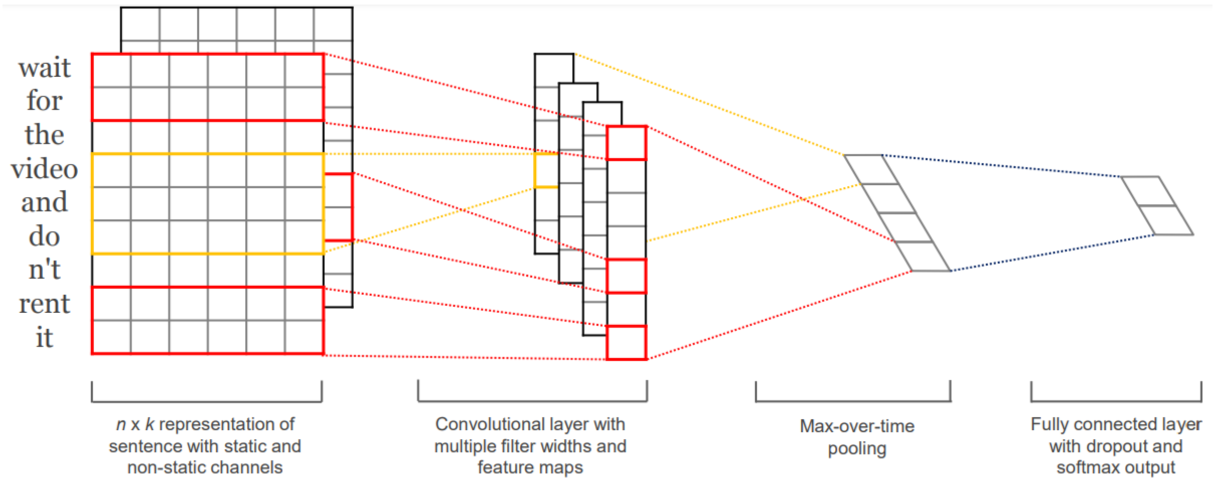
wait
for
the
video
and
do
n't
rent
it
n x k representation of sentence with static and non-static channels
Convolutional layer with multiple filter widths and feature maps
Max-over-time pooling
Fully connected layer with dropout and softmax output

Figure 1.4: Model architecture with two channels for an example sentence [7]

1.2.3 Transformer's architecture

In this section, the architecture of the Transformer will be described and an insight into the self-attention mechanism will follow. Every image shown in this section was taken from the aforementioned paper "Attention is all you need"[4].

Internally, the Transformer is made up of N encoders and N decoders (the value of N depends on the model). Each encoder shares the same internal architecture, which is a self-attention layer followed by a feed-forward neural network. The same thing can be said for the decoders: the first layer is a self-attention layer, which is followed by an attention layer and a feed-forward neural network. The attention layer in between helps the decoder to focus on important parts of the input.

Starting from the input sequence, the usual sentence to translate, for instance; Before it can enter the first encoder module, it must be tokenized, which means that every word of the sentence must be turned into tokens, numerical vectors that the network can make sense of. In our example, every word has a corresponding token. Vectors of tokens are called Embeddings, mathematical representations of the input that will be used by the network to comprehend human language.

The last step before entering the encoder module is positional encoding. This is a crucial step, as far as the Transformer does not analyse the tokens sequentially, but in parallel, the token's positions would be lost. Positional encoding adds a unique positional signal to the token's embedding; in this way, this information is preserved and the network can understand the context from the token's position.

The first layer of the encoder is the self-attention layer. This particular layer allows the encoder to focus on the relevance of each input token, especially on those that are useful when

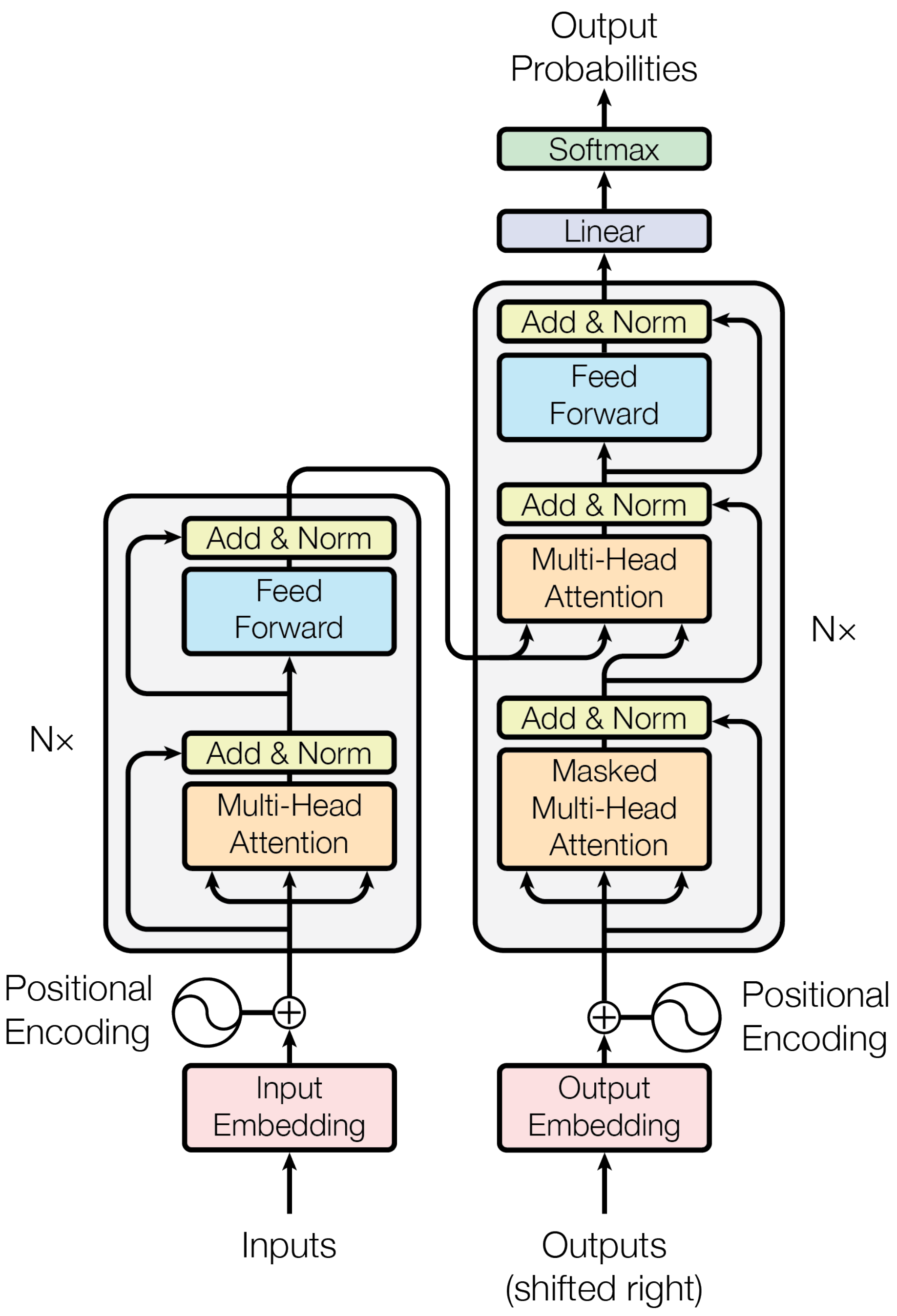

Figure 1.5: The Transformer - model architecture.[4]

it comes to predictions. We will narrow down to the self-attention layer, later in this section.

On top of that, a fully connected feed-forward network is employed: networks in which information can only flow forward. This layer is used to tune the parameters and perform the task it was trained for, at its best. Both the sub-layers are immediately followed by a normalisation layer. This layer limits the output values in a certain range, to avoid numerical instabilities while calculating derivatives, which would slow down the training process. Residual connections are added around each sub-layer: they connect the principal components of the network, skipping heavy operations and making training much faster.

Looking at the decoder module, it is similar to the encoder in many of its parts, except for the layers described below. The desired output enters the decoder, but this is right-shifted. This was done for better training results: the network is forced to predict, and translate, the following word, without knowing the correct translation; otherwise, the network would simply copy and paste the correct word. The first self-attention layer is modified to mask some tokens, i.e. words, in the output, training the network to guess them. On top of the decoder module, the linear and softmax blocks can be found. In the linear block, every token's score, named logit, is calculated, whereas the last block, the softmax function, normalizes all the logits, assigning a value to each possible token. This causes the sum of the latter values to be always equal to 1.

Self-Attention

In this section, the calculation of self-attention will be explained step by step.

The first step is to calculate three vectors out of the input token: query (Q), value (V) and key (K) vectors. To obtain these vectors, we simply multiply the input embeddings by three matrices whose values are the result of training.

Secondly, every input word has to be scored against all the other words in the sentence. This score indicates the amount of attention that must be paid to other parts of the input when encoding that word in a certain position. The score value is simply the dot product of the query vector by the key vector of the word we are scoring

Third, the resulting score has to be divided by the square root of the key vector's dimension, in order to gain more stable gradients. In the original paper, that dimension was 64, so the scores were divided by 8.

Then, the softmax function is applied to the obtained results, normalising them, so they are all positive, in a range from 0 to 1. What is obtained here is a value indicating the likelihood of a word to be expressed in a certain position; this is useful to see what are the other relevant words to the one analysed.

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{1.1}$$

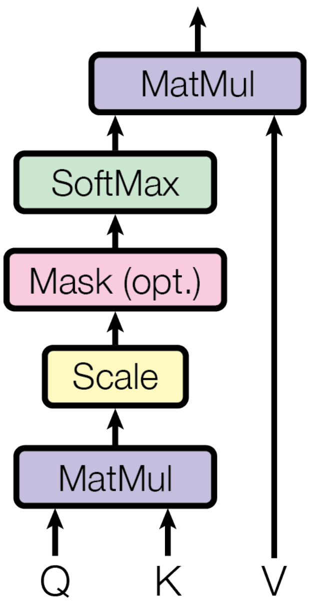

Figure 1.6: Scaled Dot-Product Attention[4]

In the last step, the softmax results obtained above are multiplied by the value vector and then summed all up: We obtained the self-attention score for one input word 1.6. All the calculation is done in a matrix, for obvious reasons of time 1.7. The idea behind this last step is to filter less relevant words from the value vectors, multiplying them by a tiny number

This is a revolutionary mechanism when it comes to training this model. First of all, it enables the model to train on multiple examples at the same time, which is an advantage both in efficiency and accuracy. Secondly, and maybe most importantly, these models can be trained on huge datasets of unlabeled examples, as they are able to find patterns between them automatically. That is why, nowadays, Transformer seems to be the natural choice to implement DNA foundational models.

1.3 Foundational Models

Foundational models are deep-learning neural networks that have been trained on a huge amount of unlabeled data. Moreover, thanks to this kind of training, foundational models internalize general representations of the data they were trained with, enabling them to perform different general tasks, such as text classification, text and image generation and so on. Foundational models, thanks to their adjustability, have reformed the way deep learning models are developed: instead of implementing and training a new model from scratch, researchers can take the foundational model that best suits their needs and adjust it with a fine-tuning protocol and their own data. Fine-tuning is a form of transfer learning, where all the weights of the foundational

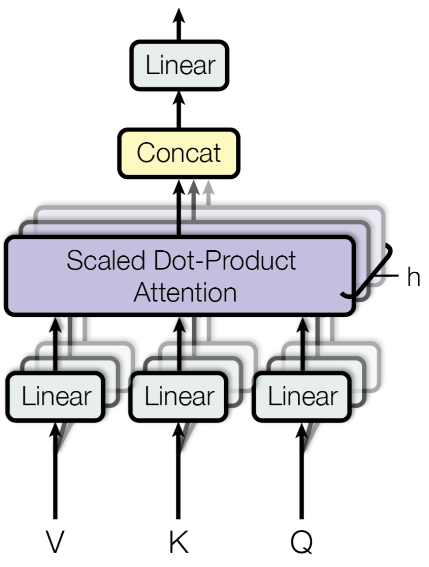

Figure 1.7: Multi-Head Attention consists of several attention layers running in parallel[4]

model are conserved, while a new Multilayer Perceptron (MLP) network is built and trained on top of the latter model. The added MLP head network will change its weights according to the input data and the 'frozen' weights of the foundational model. Further information about the fine-tuning technique exploited in this thesis, can be found in the Methods chapter. This will take less time to implement and train, and it will need way less data to achieve high performance in a particular task. Numerous foundational models had been implemented: BERT and GPT family to list but a few.

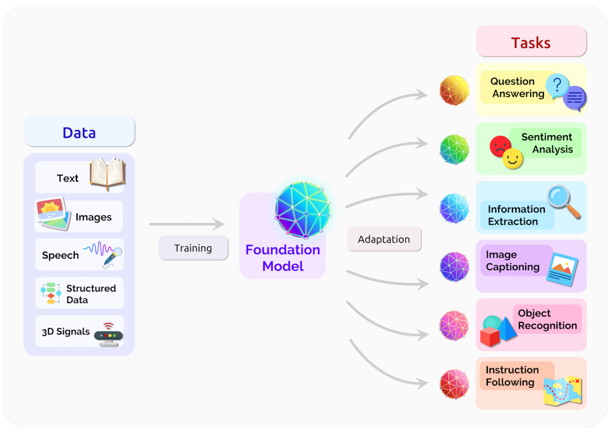

Figure 1.8: A foundation model can centralize the information from all the data from various modalities. This one model can then be adapted to a wide range of downstream tasks[8]

Chapter 2

Foundational Models for DNA sequences

Transformer-based foundational language model's potential is not limited to text analysis and classification, speech recognition and so on. As mentioned above, their adaptability, and fast improvements over time, make them applicable to tackle genomics problems too. Analysis of the DNA sequence and the protein structure, are the ones of major interest.

This thesis focuses on the first task: DNA sequence analysis. It takes into consideration two papers, ”The Nucleotide Transformer: Building and Evaluating Robust Foundation Models for Human Genomics” by Instadeep[9] and ”GENA-LM: A Family of Open-Source Foundational DNA Language Models for Long Sequences” by AIRI (Artificial Intelligence Research Institute)[10], both presenting new state-of-the-art DNA language models available for fine-tuning. These papers are described, highlighting the major differences in methods and results.

Finally, some of the models are fine-tuned on two different datasets. The objective is to confirm what is stated, i.e. that applying the fine-tuning method is fast-running, efficient despite the scarcity of data and widely accessible to everyone.

2.1 The Nucleotide Transformer

In this paper, a family of 8 different models is presented: they can vary in size and training dataset. Out of convenience, a practical scheme is provided in the table below.

model's name on HuggingFace	number of parameters	training datasets
nucleotide-transformer-2.5b-multi-species	2.5 billion	850 species from diverse phyla
nucleotide-transformer-2.5b-1000g	2.5 billion	3,202 genetically diverse human genomes
nucleotide-transformer-500m-human-ref	500 million	human reference genome
nucleotide-transformer-500m-1000g	500 million	3,202 genetically diverse human genomes
nucleotide-transformer-v2-50m-multi-species	50 million	850 species from diverse phyla
nucleotide-transformer-v2-100m-multi-species	100 million	850 species from diverse phyla
nucleotide-transformer-v2-500m-multi-species	500 million	850 species from diverse phyla
nucleotide-transformer-v2-250m-multi-species	250 million	850 species from diverse phyla

As far as the multi-species dataset is concerned, they consider multiple classes, namely Bacteria, Fungi, Invertebrate, Protozoa, Mammalian Vertebrate, and other Vertebrate. Bacterial species outnumber all other classes, but vertebrates are most present in the dataset by number of nucleotides. The following table shows the model organisms included in the multi-species dataset:

Class	Model Organisms
Bacteria	*Escherichia coli*
Fungi	*Saccharomyces cerevisiae*
Invertebrate	*Caenorhabditis elegans, Drosophila melanogaster*
Protozoa	*Plasmodium vivax, Plasmodium falciparum*
Mammalian Vertebrate	*Homo sapiens, Mus musculus, Rattus norvegicus*
Other Vertebrate	*Danio rerio, Xenopus tropicalis*

They assembled 18 different datasets, for 18 different genomic tasks, to evaluate the transformer models after self-supervised training. Two evaluation methods were applied: probing and fine-tuning.

Probing consists of using the model's embeddings of DNA sequences, learned during training, and inputting them into a smaller neural network to perform genomic label prediction.

Fine-tuning will be described in detail in the 'Method' chapter; in short, the DNA language model's head is substituted with a classification head, or a regression head, and then retrained on a task's specific dataset. Comparing the results obtained with these methods, fine-tuning seems to be the most effective: fine-tuned models matched or surpassed 15 of the 18 baseline models, while probed models matched or exceeded those of 12 out of the 18 baseline models[9].

Moreover, using a parameter-efficient technique, they could perform fine-tuning using 0.1% of the total model's parameter and a single GPU, only. Furthermore, smaller variances in performance were obtained with this technique, which demonstrates its robustness.

2.1.1 Multispecies models could make the difference

They verified that "the best performance is both model and layer-dependent"[9], but most importantly and interesting is the following result: not only the larger but also the more diverse models outperformed their smaller counterparts, which were trained on human genome only. This subsection collects the paper's findings dealing with this topic.

Firstly, the multi-species 2.5B model outperforms or matches the 1000G 2.5B model on tasks based on human-based assays, showing a mean Matthew correlation coefficient (MCC) across categories of 0.81 and 0.78, respectively.

Moreover, they compared the Nucleotide Transformer Multispecies 2.5B to HyenaDNA, a model pre-trained on the human reference genome only: the Multispecies 2.5B model outperformed HyenaDNA in 6 out of 8 human-related tasks, namely splicing classification (both acceptor and donor sites), promoters classification (both TATA and non-TATA), enhancer classification (it is an enhancer or not) and enhancer type classification.

Finally, in the paper's section ”The Nucleotide Transformer model learned to reconstruct human genetic variants”, the authors verified the capability of the models to reconstruct masked tokens. They found that the Multispecies 2.5B model achieved the highest accuracy performances overall, but when it comes to specific human sequences reconstruction, the highest level of accuracy is obtained with the Human 1000G 2.5B model. This could mean that only a human-specialized model could capture certain characteristics.

From these conclusions, increasing data diversity may be a strategy to improve prediction performances when dealing with limited computational resources and a shortage of data. Due to that, it is in the interests of this thesis to test whether the opposite phenomenon can also occur: measuring the performance of human-only trained models, on tasks derived from genomes of other model species.

2.1.2 Nucleotide Transformer v2

As you can see, the second half of the first table in this chapter hosts four models labeled ”v2”. Those are the second versions of the Nucleotide Transformer family's models, implemented by making some changes to the architecture and lengthening the training time. These changes are the following:

- incorporation of rotary embeddings (RoPE): This allows the model to encode absolute position information using a rotating matrix and to incorporate relative position information in the self-attention formulation.
- implementation of swiGLU activations: an activation function, variation of GLU function
- elimination of MLP biases and dropout mechanisms:
- augmentation of input's length: from 6 kilobase to 12 kilobase

All the ”v2” models were pre-trained on the same multispecies dataset and then fine-tuned, to perform the same set of 18 downstream tasks. Their findings are the following:

- The NT-v2-50 million-parameter model outperformed both the 500 million-parameter and the 2.5 billion-parameter model trained on the 1000G dataset
- None of the ”v2” models surpassed the original 2.5B multispecies model in performance.
- Only the v2-250 million-parameter managed to reach an MCC mean value close to the 2.5B multispecies model.

In conclusion, efficient training techniques and architecture, combined with a diverse dataset, can lead to top-level performances.

2.2 GENA-LM

In this work, a family of transformer-based language models is presented. These models are obtained by improving already implemented models for DNA sequences: DNABERT and Big-Bird.

First of all, AIRI Institute's researchers integrated Byte-Pair Encoding (BPE) for sequence tokenization. This encoding method creates a sequence dictionary, spotting the most frequent sequences within the genome. The resulting tokens range from 1 to 64 base pairs, with a mean length of 9 base pairs. The obtained dictionary reveals some biologically relevant tokens: the longest tokens, for instance, often indicate some well-known repetitive elements. Using the BPE method, it is possible to analyze even longer DNA sequences without changing the input length of the model, since ”512 overlapping 6-mers represent 512 base pairs, but 512 BPE tokens can represent approximately 4.5 kb”[10]. Nevertheless, the resolution's problem must be pointed out, since the granularity of the model, that is its abstraction of the data, is limited by the token's resolution; The application of BPE should be a case-specific choice.

The second enhancement concerns the attention mechanism: foundational GENA models are equipped with a conventional attention mechanism, while the ”sparse” GENA models utilize, in fact, a sparse attention mechanism. The sparse attention mechanism allows the analysis of longer sequences by limiting the number of connections and, still, it can capture long-distance dependencies. Using sparse attention and BPE together, the authors could train the sparse GENA models with 36 kb long sequences.

All the models were trained using the most recent Human T2T genome, together with the common variants from the 1000-genome project database and genomes from different species, including model organisms and others among the eukaryote taxa. Out of convenience, the table below provides a summary[10]

Model name	Number of parameters	Training dataset
gena-lm-bert-base	110M	T2T
gena-lm-bigbird-base-sparse	110M	T2T
gena-lm-bert-base-t2t	110M	T2T, 1000G
gena-lm-bert-base-t2t-multi	110M	T2T, 1000G , Multispieces
gena-lm-bigbird-base-sparse-t2t	110M	T2T, 1000G
gena-lm-bigbird-base-t2t	110M	T2T, 1000G
gena-lm-bert-large-t2t	336M	T2T, 1000G
gena-lm-bert-base-lastln-t2t	110M	T2T, 1000G

Among them, the authors stressed the attention on these three models

1. gena-lm-bert-base-t2t model: this model is the benchmark for the other models
2. gena-lm-bert-large-t2t model: this model has the largest number of parameters (336M) and 4.5kb input capacity.
3. The gena-lm-bigbird-base-sparse-t2t models: It has the greatest input capacity input (36kb)

To evaluate the model's performance, the created task-specific dataset, to perform the tasks listed below. They are all human-specific tasks, except for the fifth, of course.

1. polyadenylation site prediction
2. promoter classification
3. splicing site identification
4. chromatin profile prediction
5. determining the activity of housekeeping and developmental enhancers in a STARR-seq assay using Drosophila sequences[10]

Of particular interest for this thesis, is the promoter prediction task. The findings on this task are listed below:

- Models with a greater number of parameters outperform their smaller counterparts

- Models exploiting sparse attention, able to handle longer sequences, perform better than models using traditional full-attention. Nevertheless, It is measured that, models with shorter inputs and more parameters prevail over the ones using sparse attention.
- Adding multispecies data to the training dataset does not improve the results
- Using a longer input sequence yields better results. The gena-lm-bigbird-base-t2t model achieved an f1 score of 94.64 ± 0.3 on the 16 kb dataset[10], highlighting the importance of context in this kind of task.

2.3 Comparison

It is useful to highlight some differences of interest, between these two studies. The first one relates to the question about multispecies training datasets: can it improve a model's performance? Talking about the same task, namely promoter prediction, the two studies have different answers: Nucleotide Transformer models improve their performances using a diverse dataset, while GENA-LM cannot see any difference. One possible explanation to that discrepancy may lie in the quantity of diverse genomes exploited. Instadeep's researchers exploited a great number of genomes other than human's one, namely 850, whereas AIRI's researchers used 313 only, and the species ratios are not provided. Maybe, that quantity of genomes was not enough to train a model and see an appreciable difference.

Secondly, referring to the same task, the performance should be compared. The Nucleotide Transformer models were trained with a shorter input sequence length: the latter was 12kb, while GENA-LM exploited a 16kb long sequence. Nevertheless, comparing the F1 score, nearly all the Nucleotide Transformer's models outperform the best-performing model of the GENA family, gena-lm-bigbird-base-sparse-t2t. In fact, the best-performing Nucleotide Transformer in this task achieved an F1 score of 0.973. Discussing the latter model's results on promoter prediction, GENA's authors write about the importance of context analysis; However, this comparison shows that the input's length may not be sufficient, and the number of parameters becomes binding for a comprehensive analysis.

Chapter 3

Results

To test the presented models on downstream tasks and verify the accessibility of fine-tuning using HuggingFace, two datasets are assembled:

1. The first dataset was assembled to test the fine-tuning of models using an already-seen genomes. This dataset is composed of *mus musculus*'s promoter sequences and the human gut's metagenomic sequences.

2. The second dataset is made up of human gut phages genomic sequences and the human gut's metagenomic sequences. It was assembled to assess the capability of the models to recognize genome sequences that are completely different from the ones they were trained on.

All further information about the datasets can be found in the Methods chapter.

To fine-tune the models, Google colaboratory service was used, and the used code can be found in the Code chapter. Some of the foundational models were not fine-tuned: the BigBird-based model using sparse attention was not easily tunable using the notebooks provided by the authors. The latter comment should not be misunderstood, the author of this thesis adjusted, a little bit, all the notebooks provided by HuggingFace and by the two papers, both to make the code clearer and to fix errors. In this way, it was possible to use the pre-trained models with a new dataset and add some metric calculations. But, in the case mentioned above, the Python code raises some error messages that go far beyond the author's knowledge.

The following tables report the resulting metric, calculated during the finetuning process.

3.1 *mus musculus* promoter classification

In the HuggingFace repository of the author, fien-tuned models for this task are all named with the prefix ”mus_promoter-finetuned-lora”, apart from the last one, for which a different approach was used.

Model name	Validation Loss	F1	Mcc	Accuracy
NT-500m-human-ref	0.4766	0.8732	0.7192	0.8594
NT-500m-1000g	0.3065	0.9351	0.8414	0.9219
NT-2.5b-1000g	0.1687	0.9863	0.9686	-
NT-2.5b-ms	0.0117	1.0	1.0	-
NT-v2-50m-ms	0.1172	0.9863	0.9686	0.9844
NT-v2-100m-ms	0.0387	0.9867	0.9683	0.9844
NT-v2-500m-ms	0.1042	0.9863	0.9686	0.9844
NT-v2-250m-ms	0.0011	1.0	1.0	1.0
bert-large-t2t	0.1108	0.9867	0.9683	0.9844
bert-base-t2t	0.1792	0.9577	0.9094	0.9531
bert-base-lastln-t2t	0.0934	0.9867	0.9683	0.9844
bert-base-t2t-multi	0.0021	1.0	1.0	1.0
gena-bigbird-base-t2t	0.0475	0.9938	0.9866	-

The 2.5B-1000G and 2.5B-Multispecies models of the Nucleotide Transformer don’t show the Accuracy value because the CUDA device rapidly ran out of memory, and some changes were applied to the code, to carry out the fine-tuning.

Looking at the NT models, the poorest results were obtained by the 500M NT model, pre-trained on the reference human genome only. This was easily surpassed by its counterpart, pre-trained on the 1000G dataset, and overtaken by the ”v2” models. It seems like, using the same number of parameters, fine-tuning dataset and fine-tuning routine, the model pre-trained on a diverse dataset of human genomes yielded better results, even if it had to recognize non-human DNA sequences. Obviously, the best performances were achieved by the models pre-trained on multi-species datasets, where mice genome was present. The same thing can be said for the GENA-LM models since the top-performing model was the Bert-based model, pre-trained on both human and multi-species genomes.

Overall, the obtained values are really high, and that can be explained since humans and mice are both mammalian vertebrates, and their genomes have tons of similarities.
It is important to point out that the best-performing GENA’s model could reach those results

with a lower number of parameters (only 110M).

3.2 Phage sequences classification

This is a particular task, and possibly a though one for the models, since they are ask to distinguish phage DNA sequencing, which were absolutely absent from every pre-training dataset.

In the HuggingFace repository of the author, fien-tuned models for this task are all named with the prefix ”gut_1024-finetuned-lora”, apart from the last one, for which a different approach was used.

Model name	Validation Loss	F1	Mcc
NT-500m-human-ref	0.5875	0.7769	0.3628
NT-500m-1000g	0.4974	0.8222	0.5121
NT-2.5b-1000g	0.6900	0.8402	0.5492
NT-2.5b-ms	0.6652	0.8473	0.5631
NT-v2-250m-ms	0.4815	0.8414	0.5610
NT-v2-100m-ms	0.5193	0.8251	0.5308
NT-v2-50m-ms	0.4237	0.8660	0.6384
NT-v2-500m-ms	0.4480	0.8532	0.6018
bert-large-t2t	0.4378	0.4378	0.6476
bert-base-t2t	0.4700	0.8448	0.5728
bert-base-t2t-multi	0.4764	0.8478	0.5903
bert-base-lastln-t2t	0.4049	0.8657	0.6421
bigbird-base-t2t	1.2397	0.8195	0.5808

At first sight, all the metrics have values way lower than the ones seen in the previous task. Nevertheless, those are good results, meaning that the models were able to apply their acquired knowledge to distinguish the phage sequences. The pattern shown in the task above can be found in these results too. Comparing the two non-multi-species models, the best-performing is, again, the one pre-trained on the 1000G dataset, enhancing the hypothesis of the great potential of a diverse training dataset. Moreover, this is an even more outstanding fact, since the Nucleotide Transformer’s author has voluntarily excluded viruses and plant genomes from the training dataset, and still, the models capture useful features to work with phage genomes. Thanks to the architectural advancements, the ”v2” models outperformed the others, in a way that doesn’t seem bound to the number of parameters.

Less data were used to fine-tune the 2.5B models, for the same problem mentioned above; still, both the training and test ratio as the phage and metagenomic ratio was conserved. As you can see, comparing the results from these models, the one trained on the multipescies model reached higher performances.

Chapter 4

Conclusions

Transformer architecture promises great advancement in the near future. Its simplicity and adaptability allow these kinds of models to perform on different downstream tasks. Thanks to architectural improvements and the self-attention mechanism, they are suitable models to analyse long DNA sequences too. The latter is of key importance since contextual information is crucial to achieving great performances, as shown by the presented papers.

Secondly, another important finding is highlighted. From Instadeep's paper, it emerged that training a model on a diverse dataset, gathering different genomes between prokaryotes and eukaryotes, yields better results. On the other hand, from the GENA-LM's one, it is stated that no significant improvements were seen, by using a multi-species training dataset. To reach a deeper understanding of the matter, this thesis was reported the fine-tuning results of some models. It emerged that there may be a correlation between the diversity of the training dataset and the result's improvement. In addition. it may be possible that no improvements could be seen in the GENA's results since they used a small quantity of multi-species data; for the Nucleotide Transformer models family, a three times larger multi-species dataset has been used.

Moreover, it was assessed the modest ability of those foundational models to recognize phage DNA sequences, even if there were no virus DNA samples in the training datasets of both the foundational model families. This is a remarkable finding, showing the efficacy of the Transformer architecture, and its usefulness in the genomic field.

Finally, fine-tuning seems within everyone's reach. Thanks to services like the Hugging-Face website and community, it is possible to fine-tune the most famous and most performing foundational models existing, at will. Tutorials and example notebooks are provided, covering a wide range of applications. It must be pointed out that, not all the notebooks work properly when it comes to non-popular models; still, those can be fixed with a little time and effort.

In conclusion, Bioinformatics and dry-lab work will get a leading role in the genomic research field, and this new technology, DNA foundational models in particular, will have a prominent role in future discoveries.

Chapter 5

Methods

5.1 HuggingFace

To have access to the foundational models implemented by Instadeep and AIRI, I used the HuggingFace website. HuggingFace is a community and platform for data science and machine learning development. This platform offers many features:

- Model Hub: a library of more than 200,000 publicly available pre-trained models.
- Data sets: this library hosts more than 30,000 datasets provided by the community.
- Transformer library: It allows users to easily implement natural language processing models from scratch or by fine-tuning a pre-trained model from the model hub

All the results collected in this work were obtained by downloading the pre-trained models from the hub and fine-tuning them, with the commands provided by HuggingFace, on different datasets available on the Dataset library

5.2 Google Colaboratory

The utilized code, reported in the dedicated chapter, is fully written in Python. This code was run using Google Colaboratory notebooks. Colaboratory is a cloud-based Jupyter notebook, that runs exclusively on browsers and no setups are needed. The code was computed with a T4 GPU hosted by the same web service.

5.3 Datasets

Below are the details of the datasets used in this work and available on the HuggingFace page of the author: https://huggingface.co/LiukG

5.3.1 gut_pahge_and_metagenomic

This dataset gathers genomic sequences of the human gut's phages and metagenomic sequences produced by studies on human gut metagenome. This choice of data was considered to be valid since the fine-tuned foundational models were all pre-trained on the Human genome. Phage data were downloaded from the PhageScope database, which gathers and systematically elaborates phage sequences from other databases: the used sequences came from GVD (gut virome database).

The metagenomic sequences were downloaded from the NCBI Assembly database, searching for the latest uploaded assemblies available on Genebanck and downloading the first 20 items. These sequences were collected in a single FASTA file, divided into shorter segments of 1024 nucleotides and overlapping the previous and following segments for 512 nucleotides. Every segment was labeled: label '1' indicates a phage segment, whereas '0' indicates a metagenomic sequence. The resulting dataset is made up of two splits, 'train' and 'test', for a total number of 60,000 sequences and a phage sequence percentage of 60

5.3.2 mus_promoter

This dataset gathers promotorial sequences of *mus musculus* and the metagenomic sequences used in the dataset above. Mouse's promoter sequences were downloaded from EPD (Eukaryotic promoter database), selecting the species '*mus musculus*' and downloading the corresponding FASTA file. Additional data were found on the NCBI database, looking for *mus musculus* genes and filtering the results searching 'promoter'. Sequences were elaborated in the same way as the dataset above, but the segments were reduced to 300 nucleotides the resulting dataset is made up of two splits, 'train' and 'test', for a total number of 2,570 sequences and a mouse promoter sequence's percentage of 53

5.4 Fine-Tuning

In order to adapt the presented models to perform specific downstream tasks, in this thesis, the fine-tuning technique is exploited. It is a form of transfer learning, a way to leverage the internal representations and knowledge of the foundational model, acquired during the training process, and adapt them to perform on a different related task.

This protocol is way more efficient than training a new model from scratch since it needs less time and fewer data to achieve high performance, in a few epochs of training. Tools from the HuggingFace's Transformer library were used, together with the LoRA method.

5.4.1 LoRa

Low-Rank Adaptation (LoRA) method, is a type of Parameter-efficient Fine-tuning (PEFT) for large language models, introduced by Microsoft researchers in 2021. Unlike the full-parameter fine-tuning, the LoRA method tracks the parameters changes in a matrix, instead of upgrading them directly, and splits that matrix into two smaller matrices. In a nutshell, the so-called matrix of changes is the product of two sub-matrices: one column vector and one-row vector.

Step by step, when fine-tuning a model on the first batch of data, the model's parameters are ”frozen” and the changes are stored. It is trivial, at any time, to multiply the two vectors and add the matrix of changes to the ”frozen” one. According to the original paper[11], this method saves 3 times the memory usage and the results are competitive with the full-parameter fine-tuning method.

Chapter 6

Code

6.1 Fine-Tuning with Lora method

The code listed below was used to fine-tune the presented models with the LoRA method; This code is based on the Google Colaboratory's notebook provided by the 'Nucleotide Transformer' paper's authors, with some little adjustments. The original notebook is available at this link: https://huggingface.co/docs/transformers/notebooks#pytorch-bio

Installing the required modules, libraries and defining the working device:

```
#install
! pip install -q biopython transformers datasets huggingface_hub
accelerate peft
! pip install -U accelerate
! apt install git-lfs

# Imports
from transformers import AutoTokenizer, AutoModelForMaskedLM,
TrainingArguments, Trainer, AutoModelForSequenceClassification
from sklearn.metrics import matthews_corrcoef, f1_score, accuracy_score
from sklearn.model_selection import train_test_split
from datasets import load_metric, load_dataset, Dataset
from peft import LoraConfig, TaskType, get_peft_model
import torch
import matplotlib.pyplot as plt
import numpy as np


#working device
device = torch.device("cuda")
```

Inserting the links of the model to fine-tune, the chosen dataset (and its number of labels) and the name of the new model:

```

    model_to_finet = "____________________"
    original_model_name = "____________________"
    new_model_name = "____________________"
    dataset_link = "____________________"
    dataset_name = "____________________"
    num_labels = "______"
```

Loading the model for the sequence classification task: If you are loading the Nucleotide Transformer models, you can run the code below:

```

model = AutoModelForSequenceClassification.from_pretrained(model_to_finet,
    num_labels=num_labels, trust_remote_code=True)
model = model.to(device)
```

Else, to load the Bert based models, run the following:

```

    ! pip install deepspeed

    model = AutoModel.from_pretrained(model_to_finet, num_labels=num_labels
    , trust_remote_code=True )
    gena_module_name = model.__class__.__module__
    cls = getattr(importlib.import_module(gena_module_name), '
    BertForSequenceClassification')
    model = cls.from_pretrained(model_to_finet, num_labels=num_labels,
    trust_remote_code=True )
    model = model.to(device)
```

Adding the LoRA parameters and the parameters to fine-tune:

```

    peft_config = LoraConfig(   task_type=TaskType.SEQ_CLS,
                                inference_mode=False,
                                r=1,
                                lora_alpha= 32,
                                lora_dropout=0.1,
                                target_modules= ["query", "value"],
                                modules_to_save=["intermediate"]        #
    modules that are not frozen and updated during the training
                            )
    lora_classifier = get_peft_model(model, peft_config)               # transform our classifier into a peft model
    lora_classifier.print_trainable_parameters()
    lora_classifier.to(device)                                         # Put the model on the GPU
```

Dataset preparation:

```

    train_dataset = load_dataset(
            dataset_link,
            dataset_name,
            split="train",
            streaming= False,
            trust_remote_code=True,
        )
    test_dataset = load_dataset(
            dataset_link,
            dataset_name,
            split="test",
            streaming= False,
            trust_remote_code=True,
        )

    # Get training data
    train_sequences = train_dataset['sequence']
    train_labels = train_dataset['label']

    # Split the dataset into a training and a validation dataset
    train_sequences, validation_sequences, train_labels, validation_labels = train_test_split(
                                                                                train_sequences,
                                                                                train_labels,
                                                                                test_size=0.05,
                                                                                random_state=42
                                                                            )

    # Get test data
    test_sequences = test_dataset['sequence']
```

```
test_labels = test_dataset['label']
```

Tokenizing the dataset:

```

# Load the tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_to_finet)

# Dataset
ds_train = Dataset.from_dict({"data": train_sequences,'labels':train_labels})
ds_validation = Dataset.from_dict({"data": validation_sequences,'labels':validation_labels})
ds_test = Dataset.from_dict({"data": test_sequences,'labels':test_labels})

def tokenize_function(examples):
    outputs = tokenizer(examples["data"], )
    return outputs

# Creating tokenized dataset
tokenized_datasets_train = ds_train.map(
                                         tokenize_function,
                                         batched=True,
                                         remove_columns=["data"],
                                     )
tokenized_datasets_validation = ds_validation.map(
                                                 tokenize_function,
                                                 batched=True,
                                                 remove_columns=["data"],
                                               )
tokenized_datasets_test = ds_test.map(
                                     tokenize_function,
                                     batched=True,
                                     remove_columns=["data"],
                                  )


from transformers import DataCollatorWithPadding
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Loading the metrics:

```
def compute_metrics(eval_pred):

    metric1 = load_metric("f1", trust_remote_code=True)
    metric2 = load_metric("matthews_correlation", trust_remote_code=True)
    metric3 = load_metric("accuracy", trust_remote_code=True)

    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)

    f1 = metric1.compute(predictions=predictions, references=labels)["f1"]
    MCC = metric2.compute(predictions=predictions, references=labels)["matthews_correlation"]
    ACC = metric3.compute(predictions=predictions, references=labels)["accuracy"]
    return {"f1": f1, "mcc_score": MCC , "accuracy": ACC,  }
```

Defining the training arguments:

```

batch_size = 8
model_name= new_model_name
args = TrainingArguments(
    f"{model_name}-finetuned-lora-{original_model_name}",
    remove_unused_columns=False,
    evaluation_strategy="steps",
    save_strategy="steps",
    learning_rate=5e-4,
    per_device_train_batch_size=batch_size,
    gradient_accumulation_steps= 1,
    per_device_eval_batch_size= 64,
    num_train_epochs= 2,
    logging_steps= 100,
    load_best_model_at_end=True,  # Keep the best model according to the evaluation
    metric_for_best_model="f1",
    label_names=["labels"],
    dataloader_drop_last=True,
    max_steps= 1000
    )
```

Defining the Trainer and fine-tuning the model:

```

trainer = Trainer(
    model.to(device),
    args,
    train_dataset= tokenized_datasets_train,
    eval_dataset= tokenized_datasets_validation,
    tokenizer=tokenizer,
    data_collator= data_collator,
    compute_metrics=compute_metrics
    )
train_results = trainer.train()
```

6.2 Fine-tuning the Big Bird based models

To fine-tune the GENA's BigBird based models, the example notebook provided by AIRI's researchers, was used. You can find the original notebook following this link: https://github.com/AIRI-Institute/GENA$_LM/blob/main/notebooks/GENA_sequence_classification_example.ipynb$

Installing modules:

```
! pip install torch --index-url https://download.pytorch.org/whl/cu118
! pip install transformers[torch] datasets
import torch
import importlib
from sklearn.metrics import matthews_corrcoef, f1_score, accuracy_score
from datasets import load_dataset, Dataset, load_metric
from transformers import AutoTokenizer, AutoModel
from transformers import TrainingArguments, Trainer
import numpy as np
```

Loading the model:

```

model = AutoModel.from_pretrained('AIRI-Institute/gena-lm-bert-base-t2t
', trust_remote_code=True)
gena_module_name = model.__class__.__module__
cls = getattr(importlib.import_module(gena_module_name), '
BigBirdForSequenceClassification')
model = cls.from_pretrained('AIRI-Institute/gena-lm-bert-base-t2t',
num_labels=2)
```

Loading the dataset:

```

  train_dataset = load_dataset(
      dataset_link,
      dataset_name,
      split="train",
      streaming= False,
      trust_remote_code=True,
  )
  train_dataset = train_dataset.map()

  test_dataset = load_dataset(
          dataset_link,
          dataset_name,
          split="test",
          streaming= False,
          trust_remote_code=True,
      )
  test_dataset =test_dataset.map()
```

Tokenizing the dataset:

```

  tokenizer = AutoTokenizer.from_pretrained('_______________')
  def preprocess_function(examples):
      return tokenizer(examples["sequence"], truncation=True, max_length=128)
  train_dataset_tokenized = train_dataset.map(preprocess_function, batched=True)
  test_dataset_tokenized = test_dataset.map(preprocess_function, batched=True)

from transformers import DataCollatorWithPadding
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Loading the metrics:

```

  def compute_metrics(eval_pred):

      metric1 = load_metric("f1", trust_remote_code=True)
      metric2 = load_metric("matthews_correlation", trust_remote_code=True)
      metric3 = load_metric("accuracy", trust_remote_code=True)
```

```


        logits, labels = eval_pred
        predictions = np.argmax(logits, axis=-1)

        f1 = metric1.compute(predictions=predictions, references=labels)["
    f1"]
        MCC = metric2.compute(predictions=predictions, references=labels)["
    matthews_correlation"]
        ACC = metric3.compute(predictions=predictions, references=labels)["
    accuracy"]
        return {"f1": f1, "mcc_score": MCC , "accuracy": ACC,  }
```

Defining the trainin arguments, the Trainer and fine-tuning the model:

```

training_args = TrainingArguments(
    output_dir="test_run",
    learning_rate=2e-05,
    lr_scheduler_type="constant_with_warmup",
    warmup_ratio=0.1,
    optim='adamw_torch',
    weight_decay=0.0,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=32,
    num_train_epochs=10,
    evaluation_strategy="epoch",
    save_strategy="epoch",
    logging_strategy="epoch",
    load_best_model_at_end=True
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset_tokenized,
    eval_dataset=test_dataset_tokenized,
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics,
)

trainer.train()
```

Bibliography

[1] N. H. G. R. Institute. “Talking glossary of genomic and genetic terms.” (February 21, 2024), [Online]. Available: https://www.genome.gov/genetics-glossary/genomics.

[2] F. R. V.A. McKusick, “A new discipline, a new name, a new journal,” *Genomics*, 1987.

[3] E. Routhier and J. Mozziconacci, “Genomics enters the deep learning era,” *PeerJ*, vol. 10, e13613, Jun. 2022. doi: 10.7717/peerj.13613. [Online]. Available: https://hal.science/hal-03930029.

[4] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, *et al.*, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[5] R. M. Schmidt, “Recurrent neural networks (rnns): A gentle introduction and overview,” *CoRR*, vol. abs/1912.05911, 2019. arXiv: 1912.05911. [Online]. Available: http://arxiv.org/abs/1912.05911.

[6] K. O’Shea and R. Nash, “An introduction to convolutional neural networks,” *CoRR*, vol. abs/1511.08458, 2015. arXiv: 1511.08458. [Online]. Available: http://arxiv.org/abs/1511.08458.

[7] Y. Kim, “Convolutional neural networks for sentence classification,” *CoRR*, vol. abs/1408.5882, 2014. arXiv: 1408.5882. [Online]. Available: http://arxiv.org/abs/1408.5882.

[8] R. Bommasani, D. A. Hudson, E. Adeli, *et al.*, “On the opportunities and risks of foundation models,” *CoRR*, vol. abs/2108.07258, 2021. arXiv: 2108.07258. [Online]. Available: https://arxiv.org/abs/2108.07258.

[9] H. Dalla-Torre, L. Gonzalez, J. Mendoza-Revilla, *et al.*, "The nucleotide transformer: Building and evaluating robust foundation models for human genomics," *bioRxiv*, 2023. doi: 10.1101/2023.01.11.523679. eprint: https://www.biorxiv.org/content/early/2023/09/19/2023.01.11.523679.full.pdf. [Online]. Available: https://www.biorxiv.org/content/early/2023/09/19/2023.01.11.523679.

[10] V. Fishman, Y. Kuratov, M. Petrov, *et al.*, "Gena-lm: A family of open-source foundational dna language models for long sequences," *bioRxiv*, 2023. doi: 10.1101/2023.06.12.544594. eprint: https://www.biorxiv.org/content/early/2023/11/01/2023.06.12.544594.full.pdf. [Online]. Available: https://www.biorxiv.org/content/early/2023/11/01/2023.06.12.544594.

[11] E. J. Hu, Y. Shen, P. Wallis, *et al.*, "Lora: Low-rank adaptation of large language models," *CoRR*, vol. abs/2106.09685, 2021. arXiv: 2106.09685. [Online]. Available: https://arxiv.org/abs/2106.09685.