Gene Promoter Prediction with Deep Learning

u6650550
Nathan Hu

Abstract

We consider the task of using machine learning to learn DNA embeddings in order to make predictions on gene promoters. Promoter regions play a significant role in DNA regulation. DNABERT [1] provides pre-trained models based on existing DNA libraries using BERT, a state-of-art natural language processing model. Such pre-trained models may help in understanding the context of DNA regulation. We compare two ways of embedding DNA sequences into vectors: (1) one-hot encoding and (2) trained embeddings generated by DNABERT [1]. These vector representations can then be used in conjunction with a convolutional neural network and/or a simple linear layer for prediction tasks. Three pipelines are evaluated on their performance on yeast promoter strength prediction and human promoter classification. Our results show that the pipelines using DNABERT embeddings performed better than one-hot encoding. Our result also shows that a global understanding of DNA is transferable to different organisms and tasks.

1 Introduction

Biological promoters are crucial for gene and protein expression regulation [2]. They play an important role in engineered metabolic pathways and nearly all synthetic or natural gene circuits [2, 3, 4].

With the emergence of more and more deep-learning methods, models such as convolutional neural networks (CNNs) [2] and, more recently, transformer neural networks [5, 6] have been applied to bioinformatics tasks [1]. Kotopka and Smolke [2] outlined in their paper a CNN-guided approach to the generation of artificial promoters for the yeast *Saccharomyces cerevisiae*. They constructed a CNN model trained to predict protein expression levels of given constitutive or inducible promoters. A more recent approach by Ji et al. adapted the Bidirectional Encoder Representations from Transformers (BERT) model to DNA [1]. The original BERT model is based on the transformer model by Google [5] which achieves state-of-the-art performance on most natural language processing tasks. BERT adopts a pre-training and fine-tuning structure, where the model gains a general understanding of DNA (or language) during the pre-training stage. This model can then be fine-tuned with task-specific data to model different downstream tasks which could either be a classification or regression task. Although DNABERT was pre-trained on human genome data, it was shown to perform well on tasks relating to a different organism [1].

The paradigm proposed by DNABERT is that a general understanding of DNA is transferable to different tasks and organisms which may help elucidate certain features of DNA and its regulation.

DNABERT can *learn* representations of sequences of DNA, whereas, the alternative method, one-hot encoding, is a direct mapping of each base to a vector and therefore has no learnable parameters. In this project, we build three pipelines to assess how transferable DNABERT's knowledge of DNA is to different downstream tasks and organisms. The first pipeline one-hot encodes DNA sequences and passes these encoded sequences to a CNN, as done by Kotopka and Smolke [2]. The second pipeline uses a pre-trained DNABERT, courtesy of Ji et al. [1], to create vector representations of DNA sequences (DNABERT embeddings) which are then passed to a single dense layer, as done by Ji et al. [1]. The third pipeline combines the previous pipelines by using the same DNABERT embeddings as before but instead passes them to a CNN (the same CNN as pipeline 1) to evaluate whether a different neural network architecture performs better

than the single dense layer.

This project aims to evaluate DNABERT and one-hot encoding as two different methods of representing DNA which will help determine whether a general understanding of DNA is transferable to different tasks and organisms. This project will also evaluate the effectiveness of the neural network architectures used with either representation (CNN and single dense layer).

2 Background

2.1 Promoters

Promoters are key elements belonging to the non-coding regions of DNA which regulate the transcription of genes [7, 8]. They are located in the immediate vicinity and upstream of the transcription start site [7, 9] and work with further upstream elements called enhancers to regulate DNA transcription [9]. These DNA elements provide the specific recognition sites for DNA binding proteins; forming the basis of transcription and the 'context' of gene regulation [9]. This context, however, may extend many kilobases as regulatory DNA elements can affect transcription of very distant genes [9]. Furthermore, similar regulatory elements may carry out different functions within different contexts and multiple regulatory elements may work together providing an additional layer to the regulatory context [1, 10]. This is analogous to the way a word may have different meanings in different sentences.

Although promoters usually feature conserved motifs, such as the TATA box, their inclusion and organisation vary within genomes and across organisms [11], with some motifs still yet to be identified [12]. Promoters that are always initiating transcription are called constitutive promoters and ones that only function in certain environments are called inducible promoters [2]. Thus, the regulation of DNA is a complex and context-dependent process of many elements interacting and recruiting DNA binding proteins, such as transcription factors, via mechanisms that remain poorly understood [12, 13].

2.2 Supervised machine learning

Two broad categories of machine learning are supervised and unsupervised methods. Supervised machine learning methods use *labelled* data, meaning each training example includes a label that the model aims to predict. The training examples and their labels are then used to iteratively adjust the model's prediction so that they match the true label when given the corresponding input, in a way that allows the model to best predict on unseen data [14]. The types of prediction a supervised method can make are separated into two types of tasks: classification and regression. Classification involves sorting inputs into pre-defined classes such as [`Yes, No`] or [`Frog, Horse, Rat`] and regression involves assigning number values such as projected sales revenue for a company. Both tasks however require an input that contains information. Using the same examples as before, the inputs could be an image or historical sales data.

2.3 Convolutional neural networks

Convolutional neural networks (CNNs) are often used to work with data that comes in multiple arrays such as multi-channel 2D image data [14].

CNNs are composed of two main types of layers. These are a convolutional layer with a non-linear activation function such as ReLU and a sub-sampling layer known as a pooling layer [14, 15], normally max pooling. The four main properties of CNNs are local connectivity, shared weights, pooling and the use of many layers [14].

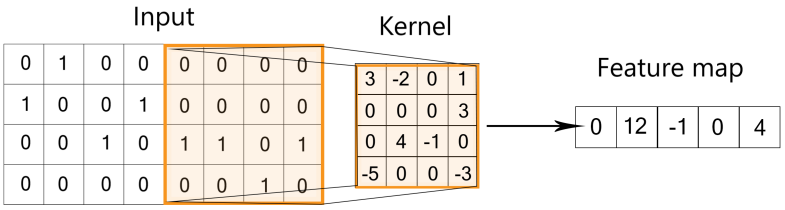

Figure 1: An example of a 1D convolution performed on an input array. The kernel focuses on a local patch of the input and performs a weighted summation to produce a single value in the feature map. This step maps the highlighted section in the input to the value 4 in the feature map (previous steps not shown). The feature map is then passed through an activation function.

Convolutional layer A convolutional layer is a set of feature maps that are mapped from the previous layer through a set of weights called a kernel or a filter [14]. These kernels only focus on local patches at a time in the previous layer (local connectivity) and a single feature map is created using the same kernel (shared weights) [14] as illustrated in figure 1. These operations work well with data where local groups of values are highly correlated which can be detected by kernels [14]. The detection of these motifs is also invariant to their location in the array because of shared weights [14]. In reality, many kernels are used to map the previous layer to multiple corresponding feature maps and many convolutional layers are used. In subsequent layers after the input layer (figure 1), each feature map is mapped to another feature map with a separate kernel [14]. After each convolution, the feature maps are passed through a non-linear activation function, such as ReLU [14, 16].

$$\mathrm{ReLU}(x) = \max(0, x)$$

The convolutional layers used in this project are only 1 dimensional (the kernel only moves in one axis direction [16], left to right as explained in figure 1).

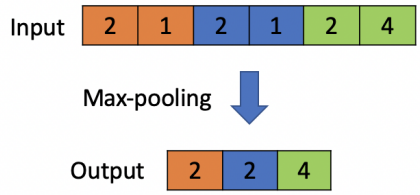

Figure 2: Max pooling operation. Image courtesy of Mavuduru [17].

Pooling layer The next operation, normally performed after a convolution, is the pooling operation. The role of pooling is to merge similar features or motifs together, reducing the dimensionality of each feature map which makes the model more invariant to small shifts and changes in the input [14]. Normally the pooling operation done is max pooling (figure 2).

2.4 BERT and Attention

Previous sequence transduction models include recurrent neural networks (RNNs), long short-term memory [18] and gated recurrent neural networks [19] which are inherently limited by sequential computation [5]. Transformers address this issue by relying solely on the attention mechanism which allows for more parallelisation [5]. Based on the original transformer neural network, Bidirectional Encoder Representations from Transformers (BERT) incorporates a truly bidirectional architecture and reduces the need for custom task-specific model architectures by adopting a *fine-*

tuning approach to language modelling, instead of a *feature-based* approach (ie. ELMo) [6]. BERT achieves state-of-the-art performance on numerous natural language processing tasks [6].

2.4.1 Architecture

The original transformer model architecture utilises an encoder-decoder structure [5], whereas BERT consists of stacked transformer encoder blocks only (figure 3), with a regression or classification head on top. Each encoder block takes embeddings from the previous block and outputs embeddings to the next block [6, 5]. The output of the final encoder block is then the final embedding, or representation, of the input.

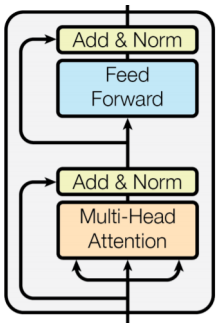

Figure 3: A transformer encoder block [5].

Key to BERT's task agnostic architecture is the *pre-training* and *fine-tuning* approach to model training (figure 4). In pre-training, a general understanding of language is learned by self-supervised training on a large corpus of text. This is done by (1) predicting masked regions of sentences (masked language modelling) and (2) predicting whether or not two given sequences follow from one another (next sentence prediction). Using the model parameters after pre-training, fine-tuning is done by further adjusting these parameters using task-specific data [6, 5]. Fine-tuning preserves the pre-trained architecture even for different tasks, only the output layers and parameters are changed [6]. Once fine-tuned, when passing input sequences to BERT the final embeddings represent the input sequence and can be passed to a classification or regression head to make a prediction.

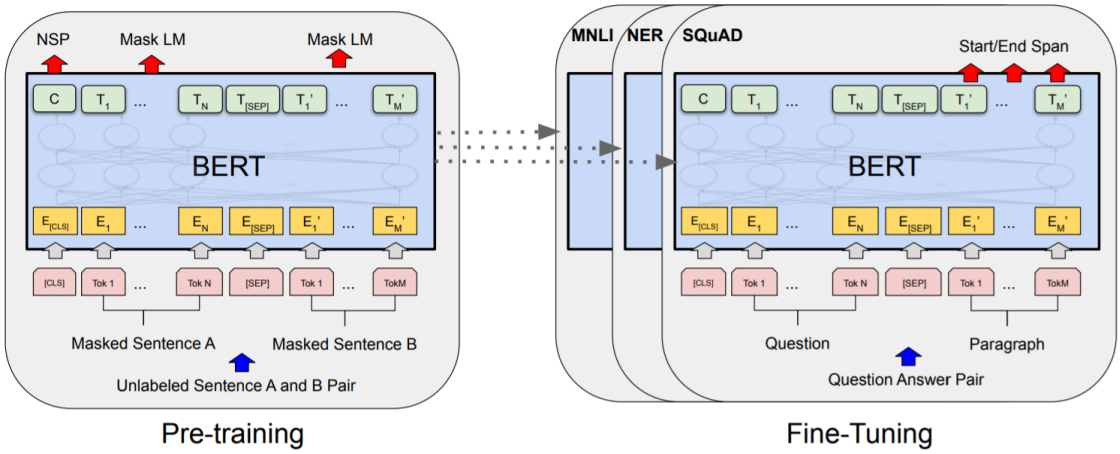

Figure 4: Figure acquired from [6]. The idea of first pre-training BERT using next sentence prediction (NSP) and masked language modelling (Mask LM) then using the same architecture, changing only the output layer, and fine-tuning for different downstream tasks. Fine-tuning is done by training on task-specific data but first initialising the model with pre-trained parameters [6]. Each square represents a fixed-length vector, so each word (or token) is mapped to a token embedding and then made into an embedding vector E_i which is then passed through BERT until it reaches a final embedding (hidden state) T_i. Special tokens include: `[CLS]` which is prepended to every token sequence and its final embedding represents an aggregate embedding of the entire token sequence and `[SEP]` which marks the end of one sequence.

Tokenization of input Sentences that are passed to BERT need to be split into words and then each word can be mapped to a token embedding (figure 5).

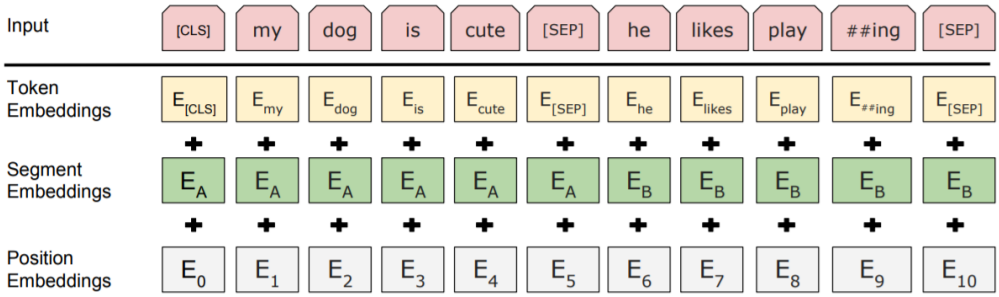

Figure 5: Figure acquired from [6]. Each word in the sentence is mapped to a unique token embedding which is then summed with a segment embedding and a position embedding. Each embedding is a learned embedding.

Each word in a sentence is mapped to a 768-dimensional token embedding via an index in a $n \times 768$ matrix, where n is the vocabulary size. This is the initial embedding for that particular word. In addition to the unique tokens which make up the vocabulary, important special tokens added are [`CLS`], [`SEP`] and [`MASK`], among a couple others. The [`CLS`] token is prepended to every token sequence and after pre-training its final embedding can be used to represent the entire sequence for sequence-level tasks [6]. The [`SEP`] token is appended to every token sequence to mark the end of one sentence and can also be used to separate two sentences.

Masked language modelling is the second pre-training objective and requires the [`MASK`] token. During pre-training, random input tokens are replaced with a [`MASK`] token and the model is tasked with predicting these masked tokens using only the context provided by the unmasked regions [6].

To retain the positional information of each token (as this information is lost when tokens are passed simultaneously to the model) a positional embedding is added. A segment embedding is also added which marks whether the token belongs to sentence A or sentence B. These are then the initial embeddings which are passed to the first of 12 encoder blocks.

Attention mechanism BERT understands the global context of a given sentence by utilising self-attention. Self-attention helps add context to words in a sentence and is used in every encoder block of BERT (figure 3).

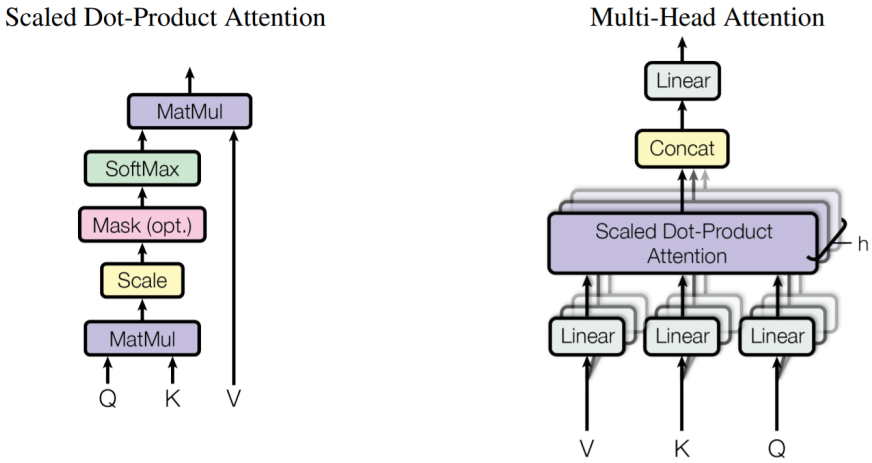

Figure 6: Figure acquired from [5]. (left) Attention mechanism involving query, key and value matrices which are scaled and softmaxed. (right) Attention but applied to query, key and value matrices which have been linearly projected, forming a 'head'. Separate heads are then concatenated and linearly projected.

Taking two token embeddings from a sequence of length n we designate the first token embedding as the query vector, q, and the second as the key vector, k. Calculating the dot-product of these two vectors returns a scalar, which is ultimately used to understand the relationship between the two tokens (words). Repeating this process for every pair of tokens in the token sequence (sentence) we can construct an $n \times n$ matrix which is then scaled before applying the softmax function to every row. This can be done by packing each token embedding in the token sequence into a matrix, M. Then we define the query, key and values as $Q = K = V = M$, and then perform the matrix multiplication QK^T, setting aside V. As the same token embeddings are used to construct a query, key and value matrix, this is called self-attention.

Every row in QK^T after scaling and softmax represents a token and its relationship to every other token in the sequence in the form of weights. This matrix is then used to reweigh each token embedding by multiplying the matrix with V, producing an attention-weighted output of the token embeddings.

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

Here d_k is the dimensionality of the query and key vectors and is used to scale the dot-products, resulting in scaled dot-product attention. To allow the model to attend to different contexts involving the same tokens within the sequence, multi-head self-attention is used. Instead of performing attention directly on Q, K and V created from token embeddings, they are first linearly projected using weight matrices W_i^Q, W_i^K and W_i^V, which also provides learnable parameters. Each set of projected query, key and value matrices forms a 'head' and attention is calculated with these projected matrices. The resulting attention-weighted output for each head is concatenated and passed to a linear layer, producing the final output of the multi-head self-attention block.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^o$$
$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where W^0 is the weight matrix used to perform the final linear projection, producing the final output of the multi-head self-attention sub-layer. This is what is then passed to the next sub-layer in the encoder block which is a fully connected feedforward network consisting of two linear transformations with a ReLU activation in between [5].

In summary, each word is converted to a token embedding and this token embedding is passed, alongside the other token embeddings in the sentence, through each of the 12 encoder blocks such that each time it does the embedding changes (it changes depending on the context of the sentence). The encoder does not change the dimensionality of the token embedding, so at the beginning, in between encoder blocks and at the end all token embeddings are 768-dimensional.

2.5 Literature review

This project is largely based on the work by Kotopka and Smolke [2] and Ji et al. [1]. Both use different representations of DNA. Kotopka and Smolke [2] use one-hot encoding while Ji et al. [1] use a learned DNA embedding by adapting BERT [6] to the language of DNA.

2.5.1 CNN applied to yeast promoters

Kotopka and Smolke [2] build a CNN to predict the strengths of PGD and ZEV yeast promoter sequences (regression). This project focuses on the ZEV promoters which are yeast promoters with a binding site for the artificial transcription factor ZEV and can be induced in the presence of beta-estradiol [2]. Therefore, in this case, the examples are the promoter sequences and the labels are their induced strengths. Promoter strength is measured as the expression ratio of green fluorescent protein (GFP) regulated by the promoter in question and another fluorescent protein (mCherry) regulated by a constitutive promoter. These sequences are represented in one-hot format and passed to the CNN to output a prediction. One-hot encoding refers to mapping each base A, C, G and T to a corresponding one-hot vector.

$$A = \begin{pmatrix}1\\0\\0\\0\end{pmatrix},\ C = \begin{pmatrix}0\\1\\0\\0\end{pmatrix},\ G = \begin{pmatrix}0\\0\\1\\0\end{pmatrix},\ T = \begin{pmatrix}0\\0\\0\\1\end{pmatrix}$$

These vectors can then be combined to represent a sequence of DNA bases as a matrix. As the task tackled by Kotopka and Smolke [2] was a regression task, the metric they used to evaluate the performance of their model was R^2. R^2 score represents the proportion of variance in the dependent variable explained by the model [20]. If $\hat{y}_i$ is the prediction and y_i is the true label:

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

where $\bar{y}$ is the average true label.

2.5.2 BERT applied to human promoters

Ji et al. [1] extend the analogy made about linguistic language and DNA by applying BERT to bioinformatics tasks such as promoter classification, transcription factor binding site and splice site prediction, naming it DNABERT. DNABERT is first pre-trained with the masked language modelling objective described in section 2.4.1 but without the next sentence prediction objective. Single sequences of DNA are $k$merised such that a $k$mer is analogous to a word and the span of all $k$mers from a DNA sequence is analogous to a sentence. Thus, masked language modelling occurs by masking contiguous spans of $k$mers and asking the model to predict these masked $k$mers (contiguous because bases can be trivially inferred from adjacent $k$mers).

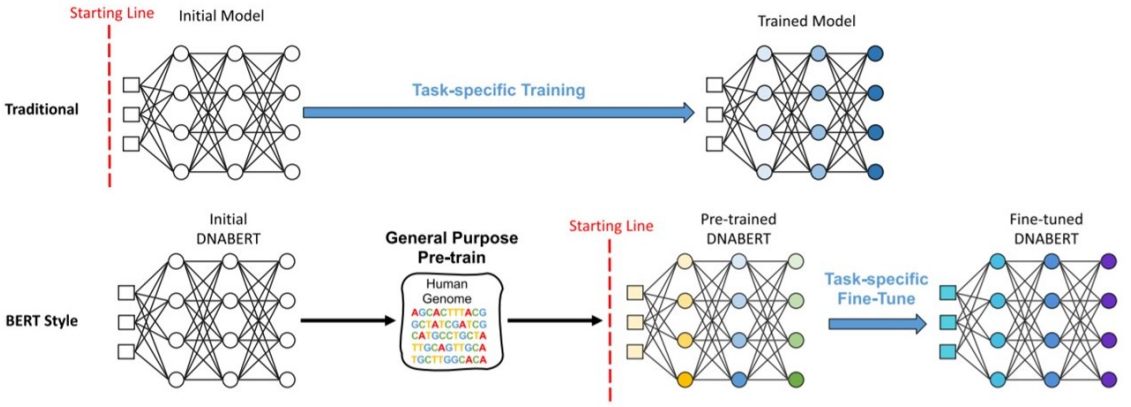

Figure 7: Figure acquired from [1]. (top) Traditional method of training a model from initial weights to final weights. (bottom) The fine-tuning approach used by DNABERT (and BERT). Initial weights are *pre-trained* using the human genome to create a pre-trained model which can be *fine-tuned* with task-specific data by further adjusting the weights.

As pre-training is *self-supervised* (requires data with no labels) the sequences used for masked language modelling are sourced from the entire human genome where 5-510 base pair lengths are generated via direct non-overlap splitting and random sampling [1]. Once pre-trained, DNABERT can be fine-tuned by initialising the model with the pre-trained parameters, changing the output layer to suit the task and training on task-specific data. One downstream task used to fine-tune DNABERT was the classification of human genome sequences as either a promoter or not a promoter [1].

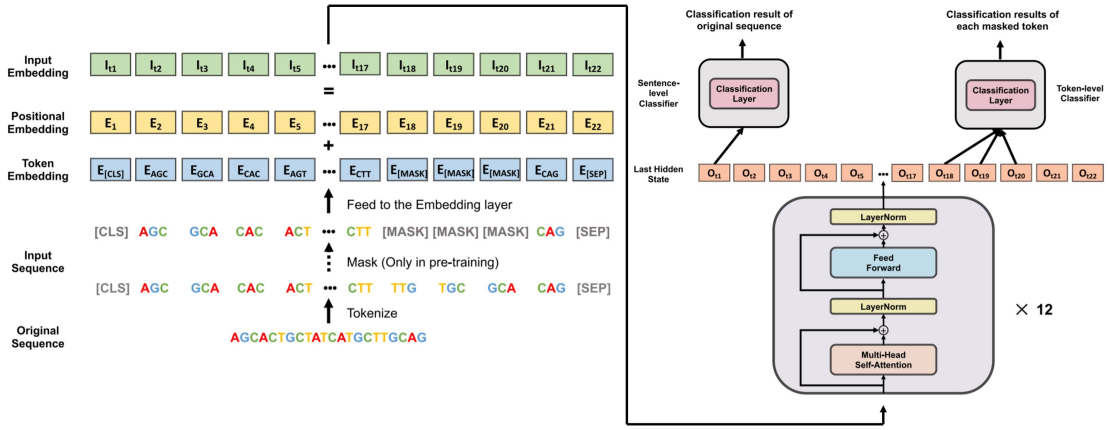

Figure 8: Figure acquired from [1]. Summary of DNABERT's architecture. DNA sequences are passed to the embedding layer which outputs a input embedding. The input embedding is passed to the 12 transformer encoder blocks (bent arrow) which produces a final embedded representation of each token to be used in different tasks.

The architecture of DNABERT is the same as BERT, with some minor alterations. An input sequence is converted to $k$mer tokens which are fed to an embedding layer. The embedding layer adds the positional embedding but not the segment embedding as DNABERT does not do NSP. This is the embedding which is then passed to the 12 transformer encoder blocks, each containing the same two sub-layers (multi-head self-attention and feedforward). The orange O_{ti} squares represent the final embeddings of each token. The final embedding of the `[CLS]` token is used as an aggregate of the DNA sequence for sequence level classification tasks. These final embeddings will be henceforth referred to as DNABERT embeddings and the final `[CLS]` token embedding represents the entire input sequence of DNA. Finally, these embeddings are passed to a classifier or a regressor, which in the case of DNABERT and BERT is a dense layer, which outputs a prediction. For the human promoter classification task performed, only the final `[CLS]` embedding is passed to the output layer.

Ji et al. [1] use the widely adopted classification metrics accuracy, F1 score and Matthew's correlation coefficient (MCC) [7]. If we define the number of true positives, true negatives, false positive and false negatives as tp, tn, fp, fn, respectively, we can define [20, 7]:

$$\text{Accuracy} = \frac{tp}{tp + tn}$$

$$\text{F1 score} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$MCC = \frac{tp \times tn - fp \times fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}}.$$

where $\text{Precision} = tp/(tp + fp)$ and $\text{Recall} = tp/(tp + fn)$.

3 Methods & Results

All code was written in Google Colaboratory using PyTorch `1.8.1+cu101`. The GPU used was normally the Tesla V100.

There are 2 promoter datasets (human and yeast) and 2 tasks (regression and classification) in the aforementioned literature review. Kotopka and Smolke [2] perform regression on yeast promoters with one-hot encoding and a CNN and Ji et al. [1] perform classification on human promoters using DNABERT embeddings with a dense layer. This project will investigate yeast promoter regression and human promoter classification.

3.1 Pipelines

Three pipelines will be used to investigate these two tasks: (1) using the same one-hot encoding and CNN approach as Kotopka and Smolke [2] (2) the same DNABERT embedding with a dense layer approach as Ji et al. [1] and (3) a new combined pipeline using the DNABERT embedding with a CNN. The pipelines only need to be slightly adjusted to perform regression or classification. In regression, the final output of the last layer is a single number and for classification, the final output is 2 numbers (representing the probabilities of a yes or a no). In all pipelines, Adam was used as the optimiser, Huber loss as the loss function for regression and cross-entropy loss for classification.

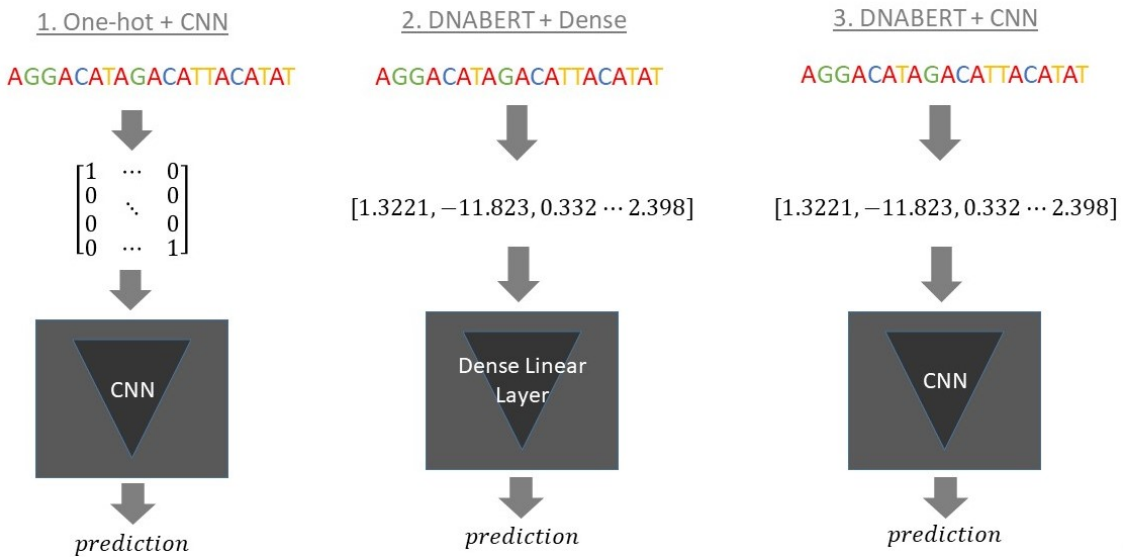

Figure 9: Pipeline 1 (left): DNA sequences are one-hot encoded (matrix) and passed to the CNN producing a prediction. Pipeline 2 (centre): DNA sequences are passed to DNABERT producing an embedding (vector) which is then passed to a single dense layer producing a prediction. Pipeline 3 (right): DNA sequence is encoded in the same way as pipeline 2 except they are instead passed to a CNN. The prediction is either a single value or a 2-valued vector depending on if the task is regression or classification, respectively.

Pipeline 1 (One-hot+CNN) The first pipeline evaluated was using one-hot encoding to represent sequences of DNA which were then passed through a CNN. The CNN specified by Kotopka and Smolke [2] was rewritten using the PyTorch [16] machine learning framework as it was originally written in TensorFlow1.

First, the input sequences were one-hot encoded by mapping each base to a one-hot vector. So a sequence of DNA was represented with a $4 \times n$ matrix, where n is the length of the sequence and 4 corresponds to the number of bases in DNA. The one-hot encoded input is then passed to 6 1-D convolution layers with max-pooling and finally 2 linear layers to output a prediction. The kernel size was 8, the number of filters was 128, non-linear activation was ReLU and pool size was 2. To aid model training an augmented dataset was generated by sub-setting each promoter sequence using a sliding window.

Pipeline 2 (DNABERT+Dense) The second pipeline evaluated was using a pre-trained DNABERT model and fine-tuning it with either the yeast or human promoter data. As a reminder, fine-tuning refers to training the model after initialising it with pre-trained parameters. The pre-trained DNABERT model, as mentioned earlier, was pre-trained using human genome data. DNABERT produces learned embeddings of input sequences which can be passed to a classifier or a regressor. In pipeline 2 (and DNABERT and BERT) this classifier/regressor is a single dense layer and the final `[CLS]` token embedding is what is passed.

As the original DNABERT architecture was only used for classification tasks, additional classes/functions in DNABERT's source code had to be created for regression. This involved creating a new `Transformer.DataProcessor` to supply features and the appropriate label to the model during *fine-tuning* and as evaluation on a validation set is done during *fine-tuning* a metric function also had to be made which calculated the R^2 score.

Pipeline 3 (DNABERT+CNN) BERT, and therefore DNABERT, consists of a 'base' model which outputs final embeddings of the tokens which it then passes to a classifier or regressor. Therefore, we can use the base model and its final embeddings (DNABERT embeddings) directly with another neural network architecture, instead of a single dense layer. Combining the DNABERT embeddings with the CNN from pipeline 1 we can create a new pipeline that passes final embeddings (specifically the `[CLS]` token embedding) of DNA sequences to a CNN. With this, we can evaluate whether a different neural network architecture would be better than the single dense layer that is normally used with BERT and DNABERT.

3.2 Task 1: Yeast promoter regression

The first task requires predicting the strength of a given yeast promoter sequence. Each example is labelled with a promoter strength represented by the GFP:mCherry expression ratio when induced by beta-estradiol. Although Kotopka and Smolke [2] use two different libraries of yeast promoters: constitutive PGD promoters and inducible ZEV promoters, this project focuses on ZEV promoters.

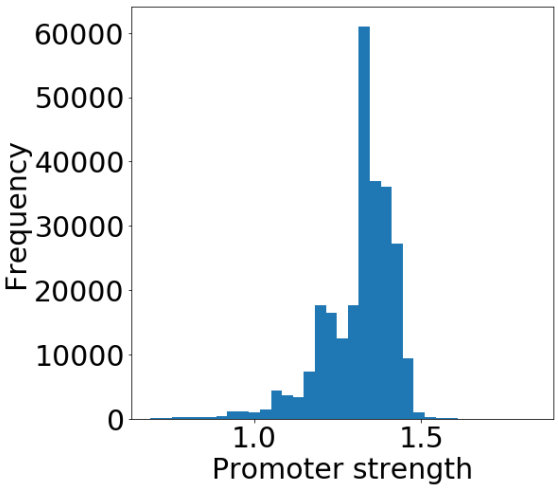

Figure 10: Histogram of the 327,000 ZEV promoter strengths. Promoter strength is measured as the GFP:mCherry expression ratio for each promoter.

The ZEV dataset provided by Kotopka and Smolke [2] includes 327,000 ZEV promoter sequences with their induced strengths. Each sequence is 242 bases long, include ZEV binding sites, the TATA box motif and a transcription start site. Again, pipeline 1 is the same as the architecture outlined by Kotopka and Smolke [2] so we are also attempting to reproduce their results and also compare this pipeline to pipeline 2 and pipeline 3 for the task of yeast promoter regression. Each pipeline used the same training, validation and test data. The split was 80:10:10.

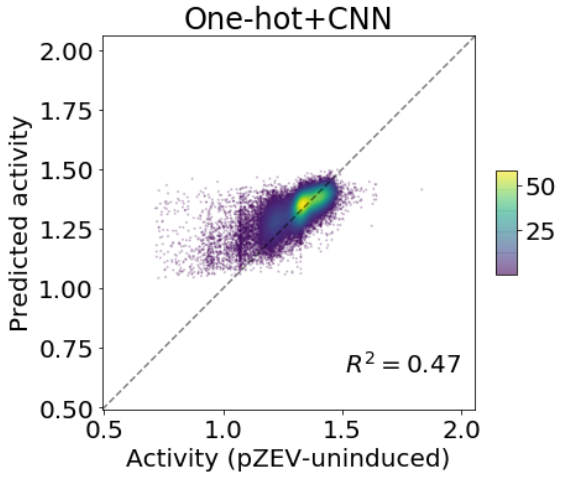

Figure 11: A true vs predicted label plot for pipeline 1 predicting induced ZEV promoter strengths. R^2 score is 0.47. Activity is the measured GFP:mCherry expression ratio.

Figure 11 compares the predicted value and the measured (true) value of each promoter sequence in the test set. This is the same procedure as done by Kotopka and Smolke [2] but they quote in their supplementary section an R^2 score of 0.73 which is much higher than the highest score achieved after several attempts and some hyperparameter tuning (R^2 of 0.47). Most of the mispredictions are in general over-predictions, as seen by the large proportion of points above the best-fit line. Figures 10 and 11 indicate that the label distribution is skewed, with a higher density of labels at high activities. As Kotopka and Smolke [2] used data augmentation, this pipeline was trained for 20+ epochs until the validation scores started to decrease. The data augmentation involved generating 4 subsequences of each DNA sequence using a sliding window. Training of this pipeline took about 2 hours.

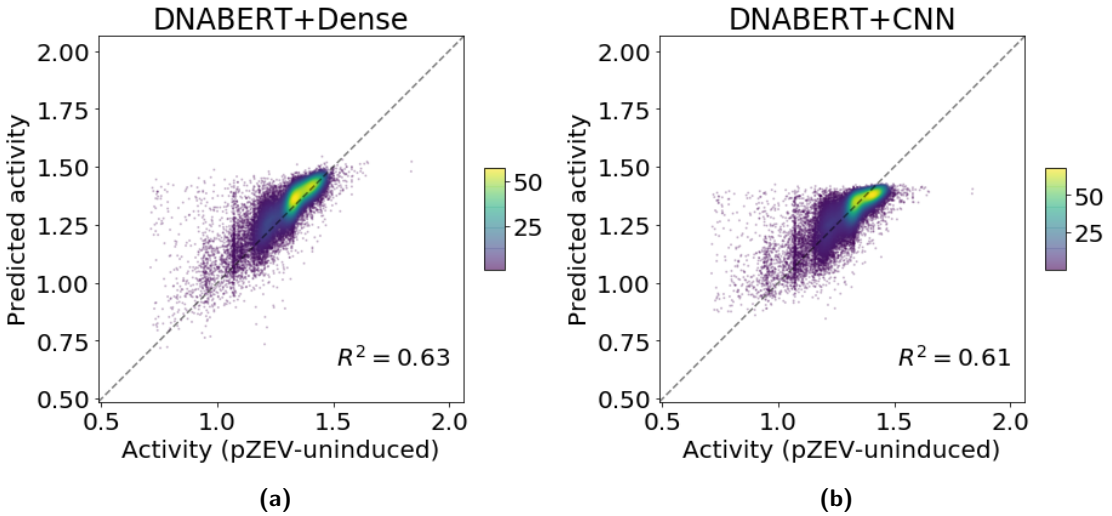

Figure 12: True vs predicted plots (a) pipeline 2: DNABERT embeddings passed to a single dense layer. R^2 score of 0.63 (b) pipeline 3: DNABERT embeddings passed to a CNN. R^2 score of 0.61

Pipeline 2 and pipeline 3 used DNABERT embeddings but passed them to a single dense layer or a CNN, respectively. Both pipelines used the same data as pipeline 1. Both pipelines are also affected by the skewed label distribution. Pipeline 2 has the highest R^2 score out of the 3 pipelines (0.63) which is much improved compared to pipeline 1. As no data augmentation was performed, these models were trained for 3 epochs only and took 3-4 hours.

3.3 Task 2: Human promoter classification

The next task was classifying whether a sequence was a promoter or not. The same dataset as Ji et al. [1] was used which contained 59,000 TATA and non-TATA containing promoter sequences. Some negative examples (non-promoter sequences) contain the TATA box motif so the model is forced to discriminate more subtle details. This task was a core promoter prediction task so each sequence was only 70 bases. The positive to negative label ratio was 50 : 50. Ji et al. [1] use pipeline 2 (DNABERT+Dense) outlined in section 3.1, just with a classifier as the output layer. Thus, we are attempting to reproduce their reported results and also compare this pipeline with pipeline 1 and pipeline 3. The metrics used are simple accuracy, F1 score and Matthew's correlation coefficient (MCC).

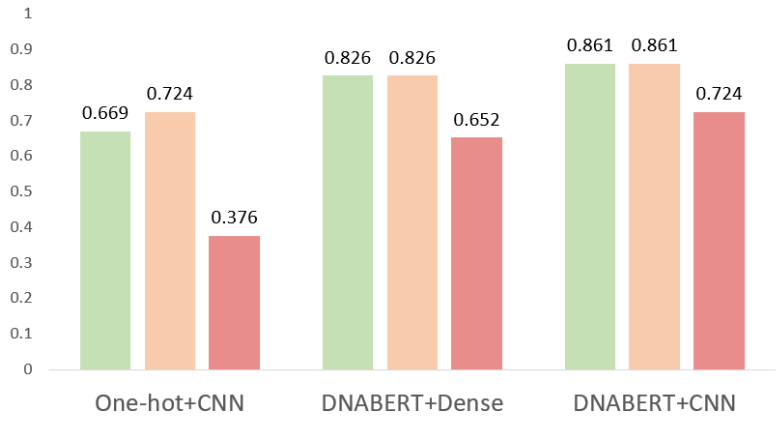

Figure 13: Scores evaluated on the validation set (green) simple accuracy. (orange) F1 score. (red) MCC. (left) One-hot encoding passed to a CNN. (centre) DNABERT embedding passed to a single dense layer. (right) DNABERT embedding passed to a CNN.

The dataset was the same for all pipelines with a 90:10 training and validation split. All pipelines took similar amounts of time to train (approximately 30 minutes). This dataset was provided by Ji et al. [1] but no test set was included, so only validation scores are shown. The pipelines which use DNABERT perform better than one-hot encoding with a CNN on all metrics. Although DNABERT with a CNN performs better than DNABERT with a single dense layer, they are similarly well-performing pipelines where they score 0.826 and 0.861 in simple accuracy, respectively. These results are in agreement with reported results by Ji et al. [1] of simple accuracy around 0.86. Each result was achieved after only a single run.

4 Discussion

It is important to note that R^2 scores are not directly comparable between different datasets as R^2 score represents the proportion of explained variance, and variance may differ across datasets [20]. As it is not known exactly what data Kotopka and Smolke [2] used to train and test their models, it may not be meaningful to compare R^2 scores or pipeline performance in general. However, there is quite a large discrepancy between reported scores and our results and it can be assumed that the distributions of training and test examples are likely similar to the original dataset distribution. As it is unlikely that the translation of code from TensorFlow1 to PyTorch could cause such a large difference in performance, the issue could lie within hyperparameter selection as Kotopka and Smolke [2] did not specify exactly what hyperparameters were used for all their models, so it may not be possible to compare with their results. However, all three pipelines evaluated in this project were trained using the same code-base and all used the same dataset.

Reproducing the results of Ji et al. [1] saw more success as both pipelines using DNABERT embeddings produced similar scores to reported scores. Although pipeline 3 performed slightly

worse than pipeline 2 in the yeast promoter regression task, the opposite was true for the human promoter classification task. Swapping the single dense layer out for a CNN improved the performance of the model. This result may motivate exploration into applying different neural network architectures to DNABERT embeddings. However, it would be important to first investigate and validate each pipeline by more careful hyperparameter tuning and averaging runs as each score was from a single training run.

The results from both tasks suggest that using DNABERT embeddings produces more accurate predictive models. This could be attributed to DNABERT's general understanding of DNA that is learned during the pre-training stage. Furthermore, this also supports the findings by Ji et al. [1] that this understanding of DNA is not only transferable to different downstream tasks but also to different organisms. As the pre-trained model provided by Ji et al. [1] was pre-trained on human genome data, this project's work could be extended by pre-training DNABERT on yeast genome data instead to determine whether there is a marked difference in performance. A limitation of CNNs is that they are restricted by their 'receptive field' so are often unable to take in the global context of a given sequence [1, 14]. This could explain the poor performance of models relying on one-hot encoding and a CNN. As DNABERT embeddings already encode the global contextual information needed, when applying a CNN to the embeddings it is not restricted by the same problem.

The promoter sequences used in this project only span the region in the immediate vicinity of the transcription start site which is not enough to include the entire context of gene regulation [9]. DNA elements that take part in the transcription of a gene may be many kilobases away from the gene in question as the process of DNA transcription often requires the three-dimensional folding of DNA. Ji et al. [1] point this out and use a 'sliding window' approach to scan much larger spans of DNA (10,000 bases), which could be a natural extension for the pipelines evaluated in this project.

References

[1] Y. Ji, Z. Zhou, H. Liu, and R. V. Davuluri. "DNABERT: pre-trained Bidirectional Encoder Representations from Transformers model for DNA-language in genome". In: *Bioinformatics* (Feb. 2021). btab083. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btab083. eprint: https://academic.oup.com/bioinformatics/advance-article-pdf/doi/10.1093/bioinformatics/btab083/36253031/btab083.pdf. URL: https://doi.org/10.1093/bioinformatics/btab083.

[2] B. J. Kotopka and C. D. Smolke. "Model-driven generation of artificial yeast promoters". In: *Nature Communications* 11.1 (Apr. 2020), p. 2113. ISSN: 2041-1723. DOI: 10.1038/s41467-020-15977-4. URL: https://doi.org/10.1038/s41467-020-15977-4.

[3] H. Redden and H. S. Alper. "The development and characterization of synthetic minimal yeast promoters". In: *Nature Communications* 6.1 (July 2015), p. 7810. ISSN: 2041-1723. DOI: 10.1038/ncomms8810. URL: https://doi.org/10.1038/ncomms8810.

[4] J. Blazeck, R. Garg, B. Reed, and H. S. Alper. "Controlling promoter strength and regulation in Saccharomyces cerevisiae using synthetic hybrid promoters". In: *Biotechnology and Bioengineering* 109.11 (2012), pp. 2884–2895. DOI: https://doi.org/10.1002/bit.24552. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/bit.24552. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/bit.24552.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[7] M. Oubounyt, Z. Louadi, H. Tayara, and K. T. Chong. "DeePromoter: Robust Promoter Predictor Using Deep Learning". In: *Frontiers in Genetics* 10 (2019), p. 286. ISSN: 1664-8021. DOI: 10.3389/fgene.2019.00286. URL: https://www.frontiersin.org/article/10.3389/fgene.2019.00286.

[8] T. A. Y. Ayoubi and W. J. M. Van De Yen. "Regulation of gene expression by alternative promoters". In: *The FASEB Journal* 10.4 (1996), pp. 453–460. DOI: https://doi.org/10.1096/fasebj.10.4.8647344. URL: https://faseb.onlinelibrary.wiley.com/doi/abs/10.1096/fasebj.10.4.8647344.

[9] J. H. Gibcus and J. Dekker. "The context of gene expression regulation". eng. In: *F1000 biology reports* 4 (2012). 8[PII], pp. 8–8. ISSN: 1757-594X. DOI: 10.3410/B4-8. URL: https://doi.org/10.3410/B4-8.

[10] R. V. Davuluri, Y. Suzuki, S. Sugano, C. Plass, and T. H.-M. Huang. "The functional consequences of alternative promoter use in mammalian genomes". In: *Trends in Genetics* 24.4 (2008), pp. 167–177. ISSN: 0168-9525. DOI: https://doi.org/10.1016/j.tig.2008.01.008. URL: https://www.sciencedirect.com/science/article/pii/S0168952508000590.

[11] J. T. Kadonaga. "Perspectives on the RNA polymerase II core promoter". In: *WIREs Developmental Biology* 1.1 (2012), pp. 40–51. DOI: https://doi.org/10.1002/wdev.21. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/wdev.21. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/wdev.21.

[12] C. Yang, E. Bolotin, T. Jiang, F. M. Sladek, and E. Martinez. "Prevalence of the initiator over the TATA box in human and yeast genes and identification of DNA motifs enriched in human TATA-less core promoters". In: *Gene* 389.1 (2007), pp. 52–65. ISSN: 0378-1119. DOI: https://doi.org/10.1016/j.gene.2006.09.029. URL: https://www.sciencedirect.com/science/article/pii/S0378111906006238.

[13] S. T. Smale and J. T. Kadonaga. "The RNA polymerase II core promoter". English. In: *Annual Review of Biochemistry* 72 (2003), pp. 449–79. ISSN: 00664154.

[14] Y. LeCun, Y. Bengio, and G. Hinton. "Deep learning". In: *Nature* 521.7553 (May 2015), pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[15] M. D. Ferreira, D. C. Corrêa, L. G. Nonato, and R. F. de Mello. "Designing architectures of convolutional neural networks to solve practical problems". In: *Expert Systems with Applications* 94 (2018), pp. 205–217. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2017.10.052. URL: https://www.sciencedirect.com/science/article/pii/S0957417417307340.

[16] A. Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[17] A. Mavuduru. *Fake News Classification with Recurrent Convolutional Neural Networks*. 2020. URL: https://towardsdatascience.com/fake-news-classification-with-recurrent-convolutional-neural-networks-4a081ff69f1a.

[18] S. Hochreiter and J. Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[19] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. 2014. arXiv: 1412.3555 [cs.NE].

[20] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.