Final Year Project Report

Alex[redacted]
Student ID: [redacted]

Supervisor: Dr. [redacted]

University of [redacted]
BSc Artificial Intelligence [redacted] FT

April 2023

Abstract

In recent years, Transformers have become the state of the art models in natural language processing. The attention mechanism of the Transformer model allows it to better understand the inherent structure of the given data. It is versatile and easily adaptable to a great number of problems. In this study, we explore the performance of the Transformer for drug discovery in the context of prediction of the blood-brain barrier permeability (BBBP) of small molecules. As most BBBP models in literature take input in the form of molecular descriptors, a series of values describing a molecule, we hypothesize about the importance of the information offered by graph representation of molecules, and whether using them might improve the performance on Transformers. We provide an affirmative answer and show the importance of using molecular graphs to train Transformers for this task by comparing against the current state of the art Transformer model for processing molecular data.

Contents

1 Introduction

The blood-brain barrier (BBB) is a membrane in the brain which protects it from neurotoxic substances from the blood [17]. It is highly selective and semipermeable, and plays a critical role in maintaining a stable environment for the brain. Since drugs targeting the brain need to penetrate the BBB in order to take effect, this makes it a difficult obstacle for drug development [23]. The blood-brain barrier permeability (BBBP) is the property of a molecule which denotes whether or not it can penetrate the BBB [21]. Accurately predicting the BBBP of a molecule is a crucial task in drug discovery, as it can help identify potential drug candidates with higher chances of crossing the BBB and reaching the target brain region.

Since their introduction, Transformer models [33] have been widely adopted in various fields due to their ability to effectively capture and model complex relationships in the input data. However, there are currently no Transformer-based models in literature explicitly designed for predicting the BBBP of molecules. In this study we consider the advantages of using molecular graphs as input for such problems. We compare adapting the traditional approach of BBBP models of using descriptor-based data to Transformers, with using molecular graphs as input instead.

2 Background

In this section we provide an overview of relevant literature, discussing previous BBBP models, and then provide a comprehensive understanding of the Transformer. We begin by introducing text tokenization, a crucial technique used by Transformers to process text. We then give a thorough explaination of the Transformer architecture and design choices. Further on, we introduce molecular fingerprinting, a technique used to represent molecules as numbers in order to be used with AI models, as well as graph neural networks. Finally, we discuss the Graphormer [37, 28] and RoBERTa [15] models, two Transformer-based models which we chose to use to solve research question.

2.1 BBBP Models in Literature

Previous methods in literature for predicting the BBBP [23, 27, 21, 34, 29, 30, 16] use molecular descriptors [5] (which are a set of mathematical representations of a molecule's properties) with more basic machine learning models such as random forests and support vector machines. The most notable ones are LightBBB, which uses a LightGBM approach [27], and DeepBBBP, which uses a mix of a multi-layer perceptron and a convolutional neural network [21].

2.2 Tokenization

In order for a machine learning model to process text, it needs to be converted into numbers. Translating the text into numbers requires it to be split into tokens. This is done with a tokenization algorithm, called a tokenizer, whose goal is to create a vocabulary for the model. There are multiple approaches of tokenization which can be primarily classified into three main categories: word-, subword- and character-based tokenization [20].

- **Word-based tokenization** assumes the input text is a sentence and splits it into words. The problem with this approach is that, becuase similar or related words (such as "token" and "tokens") are given different IDs, the model will learn different embeddings for them and treat them as entirely separate words. Another issue is that rare words may be left out of the vocabulary. During inference, these words are converted into an `[UNKNOWN]` token, stripping a number of words of their meaning.
- **Character-based tokenization** splits the input text into characters. This results in a much smaller vocabulary size and avoids unknown words, however, characters do not hold the same amount of information as words do. This approach also results in much longer sequence sizes, which will reduce the size of text a model is able to take in at once [11].
- **Subword-based tokenization** finds a middle ground between the word and character-based tokenizers. They rely on the principle that frequent words should be kept and rare words decomposed into meaningful subwords (i.e. the word "tokenization" is split into "token" and "ization") [11]. This way, semantic meaning is kept in the tokens, the vocabulary does not need to include large amounts of rare words, and input sequences can be longer. Most state of the art transformer models use subword-based tokenization algorithms [13, 25, 36].

The resulting tokens are given unique IDs which are used by the model.

2.3 The Transformer Model

The Transformer model was introduced in 2017 by Vaswani et al. in the paper *Attention Is All You Need* [33]. It is a powerful deep learning model for natural language processing and sequence to sequence modelling, converting one sequence of tokens into another. This is useful for tasks requiring the generation of a variable length sequence such as translation, question answering or summarization. The transformer model has quickly become the new state of the art model architecture for natural language processing [35]. This is primarily attributed to the attention mechanism implemented which allows the Transformer to easily capture and use long range dependencies in the input sequence without using any form of recurrence or convolution.

2.3.1 Transformer Architecture

Like many sequence to sequence models before it [26], the Transformer model uses an encoder-decoder architecture. The encoder receives a sequence of variable length of input tokens $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ and outputs a vector representation of the input as a vector $\boldsymbol{z}$. This encoded representation of the input is useful for capturing the inherent structure and meaning of the input. It is then used by the decoder to generate the output sequence $\boldsymbol{y} = (y_1, y_2, \ldots, y_m)$, one token at a time. All outputted tokens are subsequently given back as input to the decoder until a special `[END]` token is generated, marking the output sequence has been completed. Figure 1 provides a visual representation of the Transformer architecture.

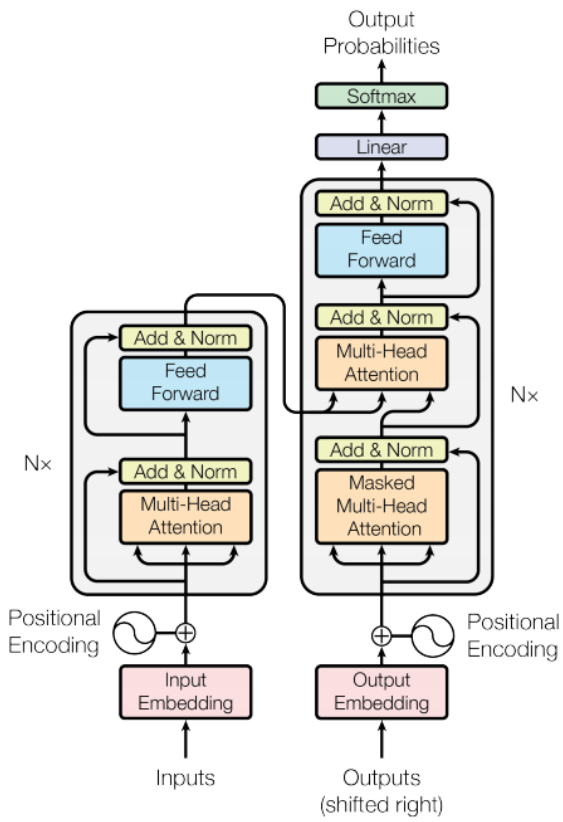

Figure 1: The Transformer model architecture as presented in *Attention Is All You Need* [33].

2.3.2 Embeddings and Positional Encoding

The generated input tokens are converted into embedding vectors of fixed size by projecting them using learned weights. An embedding vector is a list of weights representing a particular token. Since

the attention mechanism alone is permutation invariant, if left as it is, the attention mechanism will not be able to distinguish the relative order of the tokens in the input sequence. Without any positional information, the Transformer will not be able to distinguish between the different occurences of the word "bear" in the sentence "The most dangerous bear is the grizzly bear or the polar bear." Since recurrence or convolution are not present in the model architecture, positional information needs to be injected into the embedding.

Positional encoding in the Transformer is done by generating vectors of the same size as the embeddings containing the values of sine and cosine functions at different frequencies for each dimension of the embeddings:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_e}}\right) \tag{1}$$

$$\mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_e}}\right) \tag{2}$$

where pos is the position of the token in the input sequence, i is the depth of the positional encoding vector, and d_e is the dimension of the embeddings.

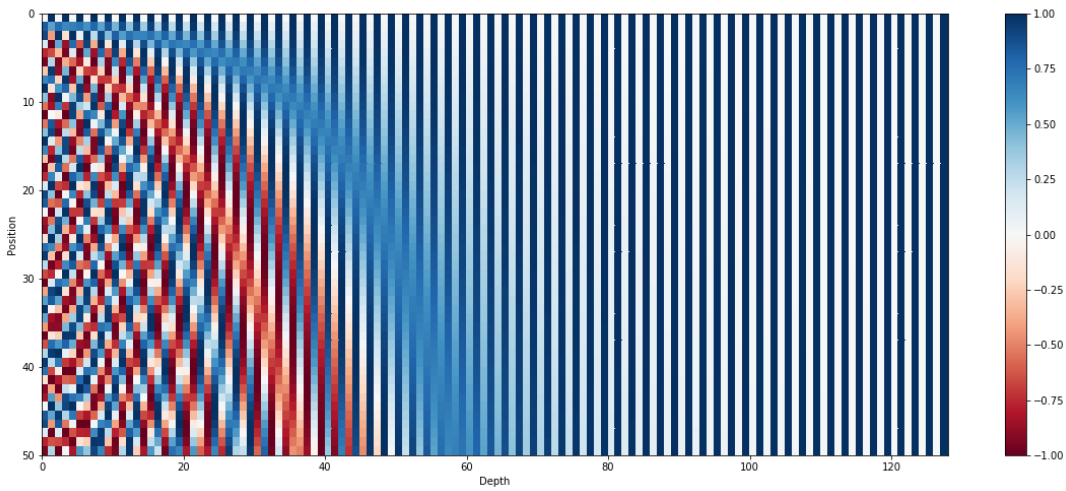

Figure 2: Positional encodings generated for 128-dimensional embeddings, each row p represents the encodings for embedding vector $\boldsymbol{e}_p$ [12].

Once generated, the positional encodings are simply added to the embedding vectors. This has the effect of perturbing the position of the embedding by a small amount in a "circular" direction, with tokens close by in the sequence being perturbed in similar directions while tokens far apart are perturbed in different directions [22]. See figure 3 for an example.

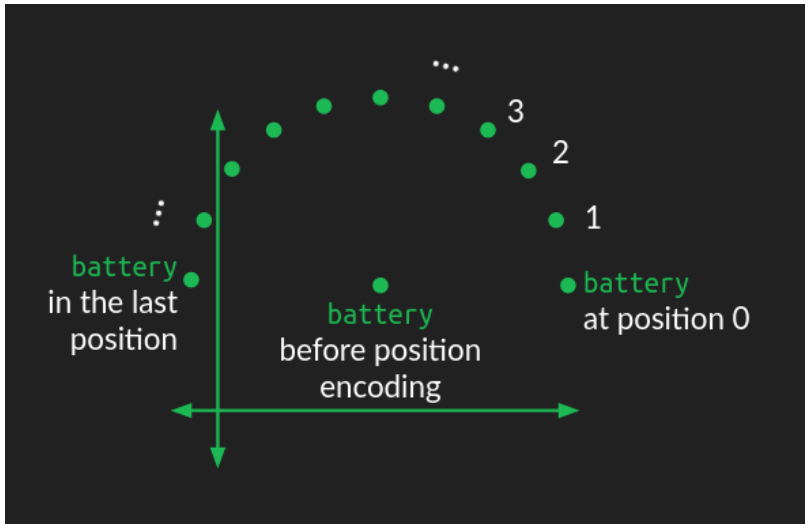

Figure 3: Results of adding positional encodings to the 2-dimensional embedding of the word "battery" at various positions in the sequence [22].

2.3.3 Attention

Attention is the mechanism which allows models to focus on specific parts of the input sequence by calculating the relationship between tokens. It receives three input vectors called the *key* (K), *query* (Q) and *value* (V) vectors, and outputs a weighted sum vector of the value vector based on the semantic correlation between tokens. The attention calculation is defined in equation 3.

$$\text{Attention}(K, Q, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_K}}\right)V \tag{3}$$

where d_K is the dimension of the K vector.

Intuitively, the attention mechanism asks the question "*How much should token* a *pay attention to token* b*?*" for each combination of tokens in the input sequence. The K, Q, and V vectors are obtained by projecting an input vector (e.g. the embedding vectors) with three different learned weight matrices W_K, W_Q, and W_V.

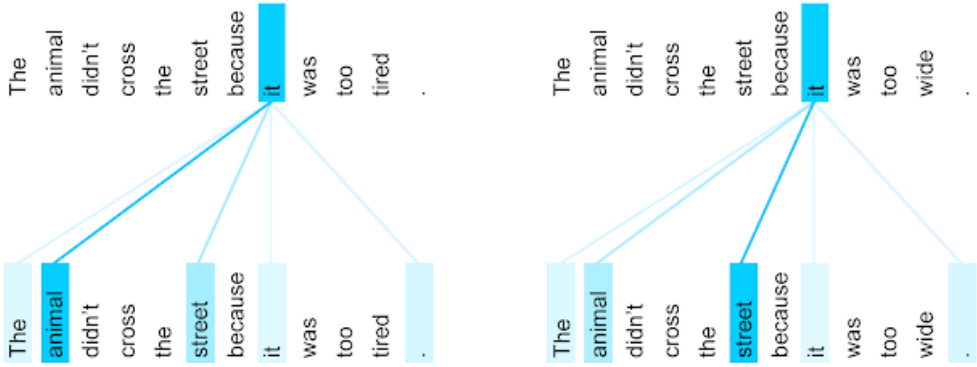

Figure 4: Example attention weights for the word "it" which pays more attention to different words depending on what the word refers to in different contexts [32].

Attention solves the issue of capturing long range dependencies. The attention mechanism's global receptive field allows each token in a given sequence to attend to each other token directly. Previously, capturing long range was done using recurrent (RNN) or convolutional neural networks (CNN), however, these had limitations:

- RNNs allow information from a token to flow through the entire sequence. The information is processed with every step, however, which limits the amount of influence tokens far apart can have on each other.

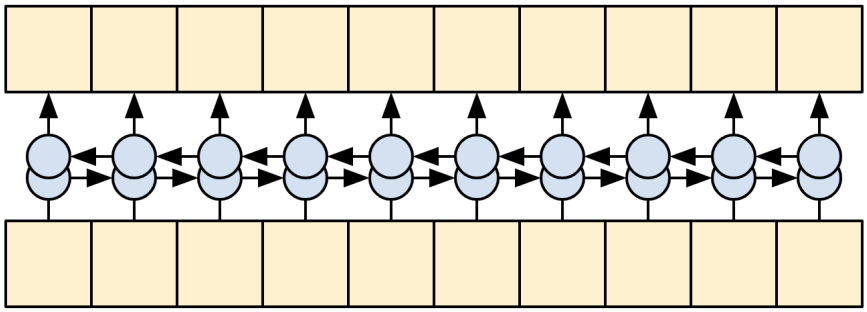

Figure 5: Representation of a bidirectional RNN, where tokens (orange squares) must pass through multiple processing steps (blue circles) in order to reach tokens further apart [31].

- CNNs have a limited receptive field which makes it hard for information from tokens far apart in the sequence to reach. Adding multiple processing layers will solve this problem, however,

the information from tokens gets processed a number of times before it reaches further tokens and adding layers increases the size and resource requirement of the model.

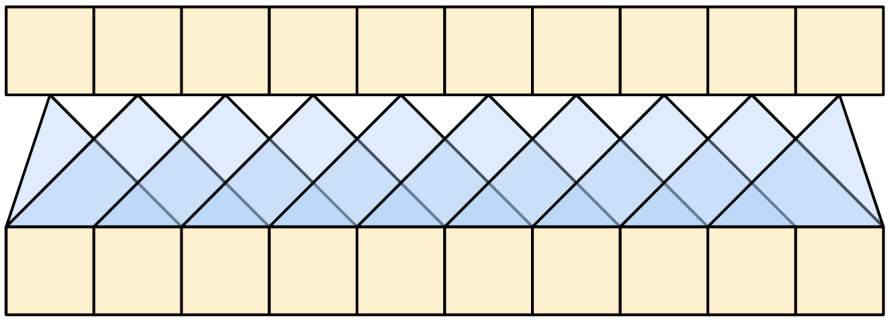

Figure 6: Representation of a CNN, where input tokens (bottom orange squares) are processed together only with tokens in their short vicinity [31].

- Attention allows the input tokens to attend to each other input token directly.

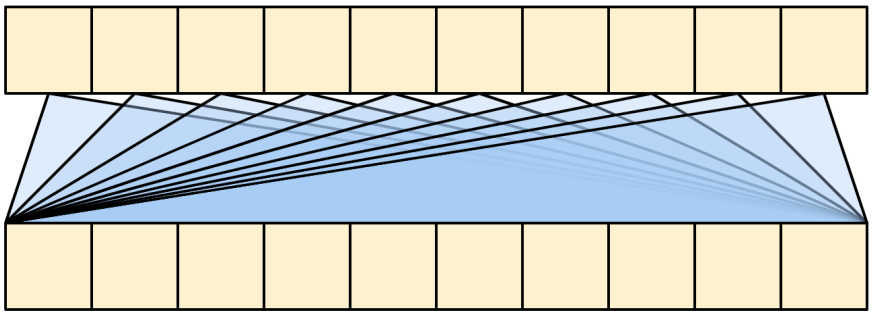

Figure 7: Representation of an attention calculation, where each input token has access to every other input token at the same time [31].

2.3.4 Multi-Head Attention

An attention calculation with a set of K, Q, V is called an *attention head*. For each attention layer in the Transformer, the K, Q, V get projected multiple times with different weight matrices so multiple heads can be computed in parallel. The benefit of having multiple attention heads is that each head can capture different relationships in the input sequence, some could focus on capturing local dependencies while others could capture global, long range dependencies. The results of each head are concatenated together and projected by a weight matrix to bring it back to the desired

inner size. Combining the results of each head helps generate a more comprehensive representation of the input sequence. The whole attention mechanism is defined in equation 4.

$$\text{MultiHeadAttn}(X) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W^O \tag{4}$$

$$\text{where head}_i = \text{Attention}(XW_i^K, XW_i^Q, XW_i^V) \tag{5}$$

where the Concat function is vector concatenation.

2.3.5 The Transformer Encoder

The encoder is composed of two sublayers, a multi-head self-attention sublayer and a two layer fully connected feed forward network sublayer with a ReLU activation [4] in the middle layer. The attention mechanism in the encoder is called *self-attention* because the input sequence attends to itself (the K, Q, V vectors are all generated from itself). Around each sublayer a residual connection is implemented by taking the sublayer's input, adding it to its own output and normalizing the result. This ensures that the vectors are modified by the sublayer, not replaced, allowing the Transformer to retain some information from the original input throughout its layers. By normalizing the output of sublayers, the Transformer can improve training speed and reach convergence faster. The Transformer encoder is formally defined in algorithm 1.

Algorithm 1 Transformer Encoder Layer

Input: X ▷ Input vector
$a' \leftarrow$ MultiHeadAttn(X) ▷ Defined in equation (4)
$a \leftarrow \text{LN}(a' + \text{X})$ ▷ Layer Normalization
$h' \leftarrow \max(0, aW_1 + b_1)W_2 + b_2$ ▷ Feed forward network
$H \leftarrow \text{LN}(h' + a)$ ▷ Layer Normalization
Output: H ▷ Output hidden representation vector

A Transformer encoder can be made up of multiple encoder layers stacked on top of one another, using the output of one layer as the direct input for the next, with no further processing needed in between. The final output of the encoder is a vector representation of the input sequence. It can be used as input for other models for various natural language processing tasks, such as the Transformer decoder for text generation or a feed forward network for sentiment analysis.

2.3.6 The Transformer Decoder

The transformer is an autoregressive model, meaning that its predicted values are based on previous predictions. Instead of the given input sequence, the decoder receives a sequence formed of a special `[START]` token and the output of the decoder for all previously seen tokens in the sequence. To preserve the autoregressive property, each token generated by the decoder attends only to itself and

past predictions. This is done by implementing a mask in the attention calculation which sets the attention weights of all connections between a token and tokens that come after it to $-\infty$:

$$\text{MaskedAttention}(K, Q, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_K}} + M\right)V \tag{6}$$

where M is the mask.

The architecture of the decoder layer is similar to that of the encoder, with only two modifications. It is composed of three sublayers, a masked multi-head self-attention sublayer, a cross-attention sublayer, and a two layer fully connected feed forward network. The cross-attention sublayer is a normal attention layer where the K and V vectors are generated from the output of the encoder and the Q vector from the output of the previous decoder sublayer. The feed forward network is the same as the one in the encoder. Residual connections between sublayers are also applied. The Transformer decoder layer is formally defined in algorithm 2.

Algorithm 2 Transformer Decoder Layer

Input: $\text{Y}^{(t-1)}$, H ▷ Previously generated decoder output and encoder output
$a'_1 \leftarrow \text{MaskedMultiHeadAttn}(\text{Y}^{(t-1)})$ ▷ Uses masked attention from equation (6)
$a_1 \leftarrow \text{LN}(a'_1 + \text{Y}^{(t-1)})$ ▷ Layer Normalization
$a'_2 \leftarrow \text{CrossMultiHeadAttn}(a_1, \text{H})$ ▷ K, V from H, Q from a
$a_2 \leftarrow \text{LN}(a'_2 + a_1)$ ▷ Layer Normalization
$h' \leftarrow \max(0, a_2 W_1 + b_1) W_2 + b_2$ ▷ Feed forward network
$O \leftarrow \text{LN}(h' + a_2)$ ▷ Layer Normalization
Output: O ▷ Output hidden vector

Similarly to the encoder, the decoder uses multiple stacked decoder layers. Finally, on top of the stack of the decoder layers, a final sublayer is implemented. This sublayer is a linear layer which projects the output of the decoder layers with the transposed weight matrix used for the embeddings. Using the same weight matrix used for embeddings but transposed can be seen as the inverse of the operation converting tokens to embeddings. The result is then put through a softmax function to generate the probabilities of each token in the vocabulary to be the next token in the output sequence. If the special `[END]` token is chosen, the output sequence is considered complete and the cycle stops, otherwise the chosen token is appended to the decoder's input and the next word is predicted again.

2.4 Molecular Fingerprinting

Molecular fingerprints are a vector representation of molecules which can easily be used with numerical models [5] for various tasks, such as downstream predictions. Similarly to molecular descriptors, fingerprints are generated by extracting pre-defined features of a given molecule. These fingerprints are problem-independent as they are fixed representations of molecular structures. Since fingerprints

are simply a numerical vector representation, this vector can be inferred from an AI model instead. Predicting fingerprints instead of using their fixed alternatives can lead to better performance [7] as they adapt to the problem they are generated for.

As shown above, the Transformer encoder is an excellent choice for data representation. Its output is highly expressive and the self-attention mechanism makes it capable of capturing complex patterns and relationships in sequential data. As such, the representations inferred by a Transformer encoder can be considered and used as molecular fingerprints.

2.5 Graph Neural Networks

A graph can be represented as a tuple $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ is the list of nodes (or vertices) in the graph and $E = \{e_1, e_2, \ldots, e_m\}$ a list of edges connecting said nodes. The v_i and e_i can be represented as feature vectors of a node or edge. A graph neural network (GNN) aims to learn a representation of a given graph [24] by continuously updating the representation of its nodes by aggregating information from neighbouring nodes. GNNs are used for various tasks such as node, edge or graph classification or link prediction.

2.6 Graphormer

The Graphormer model was introduced in 2021 by Ying et al. in the paper *Do Transformers Really Perform Bad for Graph Representation?* [37] with minor modifications in its architecture made later [28]. The Graphormer is built directly on top of the Transformer encoder architecture and achieved state of the art results on a number of graph level prediction tasks. Like any GNN before it, the goal of the Graphormer is to generate a representation of a given graph that captures its underlying structural information and relationship between the nodes and edges.

2.6.1 Graphormer Architecture

At its core, the Graphormer is an extension of the Transformer encoder. As such, it receives for input a graph as a list of nodes and a list of edges (where a node or an edge is represented by a feature vector), and outputs the representation. In order to make use of the structural and relational information provided by graphs, a number of modifications have been made to the way it computes attention and handles input sequences.

Since graphs are represented as a list of nodes and edges, tokenization is no longer required. A sequence is now formed only by a graph's nodes, each node being treated as a token. Nodes of a graphs are not inherently ordered so positional encodings are no longer required to be injected in the embeddings. This demands a permutation invariant way to process the nodes of a graph which the attention mechanism is ideal for.

2.6.2 Graphormer Attention

The attention mechanism includes three changes which add structural encodings of the graph into the Graphormer's attention calculation:

- **Node centrality** is a measure of how important a node is in the graph. This is an important property of a node, which is ignored in the original attention calculation. In order to make use of it, centrality encodings are calculated for each node using degree centrality, and added to each respective node feature vector. The degree centrality of a node is the number of edges it has connected to it. If the graph is undirected, computing only one degree vector is enough. If the graph is directed, both indegree (edges pointing to the node) and outdegree (edges connected to the node pointing out) values are added to each node's features as two separate new features:

$$h_i = x_i + deg^-(v_i) + deg^+(v_i) \qquad \text{if the graph is directed} \tag{7}$$

$$h_i = x_i + deg(v_i) \qquad \text{if the graph is undirected} \tag{8}$$

  where x_i is the feature vector of node v_i, deg is the degree of node v_i, and deg^+ is outdegree and deg^- is indegree. This way, the attention mechanism can capture both the semantic correlation and the node importance.

- **Spatial information** needs to be explicitly added to the attention mechanism since no positional encodings are used. The nodes in a graph, however, exist in non-Euclidean space and distances between them can not be measured. The shortest path distance (SPD) between nodes is used instead. For each pair of nodes v_i and v_j, the SPD is computed. If two nodes are not connected, the SPD is considered -1. For each different value obtained by the SPD, a learnable scalar is attributed to it. A matrix is created, whose element $b_{i,j}$ is the learnable scalar attributed to the SPD of nodes v_i and v_j. This matrix is added to the scaled dot product of the K and Q in the attention calculation as a bias term. This effectively adds the spatial encodings to the nodes, modifying the attention nodes pay to each other by the learnable scalar b. For example, if $b_{i,j}$ decreases as $\text{SPD}(v_i, v_j)$ increases, the nodes will pay less attention to nodes further away, and more attention to their immediate neighbours.

- **Edge information** can also be present in a graph, e.g. in a molecular graph (the graph representation of a molecule), bonds between atoms will be of different types. Edge information is encoded similarly to spatial information. For each pair of nodes v_i and v_j, (one of) the shortest path is found as a list of edges $\text{SP}_{i,j} = (e_1, e_2, \ldots, e_N)$. A new matrix is created, whose element $c_{i,j}$ is the average of the dot products of the edge features of each edge e_k in $\text{SP}_{i,j}$ and a learnable embedding. This matrix, representing the edge encodings, is added to the attention weights alongside the spatial encodings. This has the effect of modifying the attention nodes pay to each other based on the features of the edge between them.

The calculation step of each attention weight between two nodes v_i and v_j in the Graphormer is defined in equation 9.

$$A_{i,j} = \frac{(h_i W^Q)(h_j W^K)^T}{\sqrt{d_K}} + b_{i,j} + c_{i,j}, \text{ where } c_{i,j} = \frac{1}{N} \sum_{n=1}^{N} x_{e_n} (w_n^E)^T \tag{9}$$

where h_i is the node feature vector of node v_i with centrality encoding added, $b_{i,j}$ is the learned scalar associated with the value of $\text{SPD}(v_i, v_j)$, x_{e_n} is the feature vector of the n-th edge in $\text{SP}_{i,j}$ and w_n^E is the weight of the learnable embeddings for edge information at position n. See figure 8 for a detailed diagram of the new attention calculation.

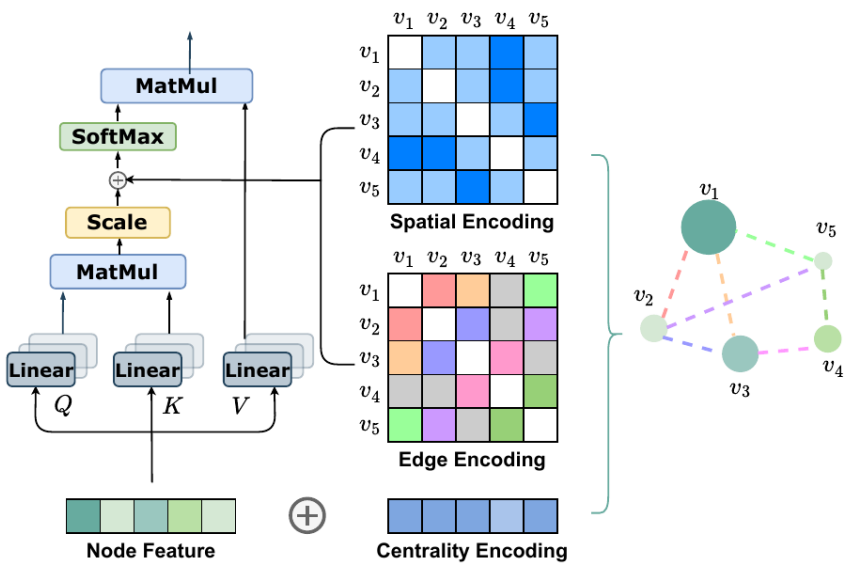

Figure 8: Diagram representing the modified attention calculation of the Graphormer. Darker squares in the spatial encodings matrix denote a longer SPD between two nodes, the color of the squares in the edge encoding matrix denotes the different types of edges and darker squares in the centrality encoding vector denotes a bigger node degree centrality [37].

A virtual node is also implemented. It is a special node connected to every other node in the graph which allows the Graphormer to capture global structural information about the graph. Since $\text{SPD}(v_{virtual}, v_j) = \text{SPD}(v_i, v_{virtual}) = 1$ but the connection between it and the other nodes is virtual, a different, unique learnable scalar $b_{virtual,j}$ and $b_{i,virtual}$ is used for the spatial encodings.

With these modifications, the Graphormer achieves state of the art results in multiple graph representation benchmarks, proving it to be the most powerful general graph neural network in literature [37, 28].

2.7 RoBERTa

The Transformer-based model we chose to compare the Graphormer against is the RoBERTa model. Introduced in 2019 by Liu et al. in the paper *RoBERTa: A Robustly Optimized BERT Pretraining Approach* [15]. It is based on the BERT [6] model architecture and improves on it by optimizing its pretraining process. The aim of the RoBERTa (and implicitly BERT) model is to learn a representation of text which can capture the complex relationships and nuances of language.

2.7.1 RoBERTa Architecture and Training Procedure

Similar to Graphormer, BERT (Bidirectional Encoder Representations from Transformers) is a Transformer encoder. Unlike Graphormer, BERT is a direct implementation of the Transformer encoder with no modifications done to it which uses raw text instead of graphs. Since the aim of the BERT/RoBERTa models is to generate potent representations of text, the key aspects of these models are in their training methods. Architecturally, RoBERTa and BERT are the same, however, RoBERTa differs from BERT by using a different pretraining method, a different training corpus and a number of miscellaneous parameter modifications. RoBERTa uses the byte-pair encoding [25] (BPE) tokenizer. It is a subword tokenizer which works by first splitting the text into words and merging the most frequent pair of consecutive bytes (or character sequences) of each word.

Pretraining is the act of training a model on a large, unlabelled dataset with the goal of understanding the structure of the data. Pretrained models are useful for *fine-tuning*. Fine-tuning involves taking a pretrained model and further training it on a smaller, labelled dataset for a specific task (e.g. for classification). The general understanding of data can help greatly improve the performance of training large models such as Transformers on small datasets.

RoBERTa is pretrained using a *masked language modelling* objective. Given an input sequence of tokens, a random sample of tokens is selected. Of that sample, 80% of them are replaced by a special `[MASK]` token, 10% are replaced by a randomly selected token in the vocabulary, and the rest are left unchanged. During training, a simple feed forward neural network head is attached to the end of the encoder. It consists of two layers, the last of which outputs the probability of what word each of the masked tokens are.

2.7.2 Fine-Tuning a Pretrained Model

Once pretrained, the masked language modelling head is removed and the model now outputs the representation of the data. To fine-tune, another head specific to the desired task needs to be fitted on top of the encoder. For example, for sentiment analysis (a classification task), a simple single linear layer can be used, which takes as input the output representations of the encoder and outputs the probability of each class in the problem. The pretrained weights are loaded for the encoder and the whole model is now further trained on labelled data.

Using these simple modifications, the RoBERTa model has become the state of the art choice in models for natural language processing [15].

2.7.3 MFBERT

The reason we chose to compare the Graphormer model against RoBERTa is thanks to the publication of the MFBERT model. MFBERT (Molecular Fingerprinting through BERT) was introduced in 2022 by Abdel-Aty et al. in the paper *Large-Scale Distributed Training of Transformers for Chemical Fingerprinting* [1]. MFBERT is a RoBERTa model pretrained on a dataset of 1.2 billion molecules in the form of SMILES [5] making it the largest deep learning model for molecular fingerprinting in literature. SMILES (simplified molecular-input line-entry system) are a way to represent molecules as plain text which makes them fit for use with a language model. MFBERT achieves state of the art performance on multiple virtual screening benchmarks [1], proving the feasibility of the RoBERTa model in drug discovery tasks.

3 Hypothesis

The goal of this study is to investigate whether using molecular graphs as inputs for Transformer-based models would improve their performance. As such, we formulate our hypothesis as follows:

Hypothesis: Incorporating graphs as inputs into Transformer-based models for drug discovery tasks will improve their performance, as the graph inputs will enable the model to better capture and utilize relational and structural information present in the data.

Graphs are a natural representation of many real world problems. They have the potential to hold much more information than generated molecular descriptors or SMILES. With the introduction of the global receptive field of the Transformers, graphs can now be processed much more effectively. Given the architectural similarity between Graphormer and RoBERTa, and RoBERTa's proven use in the domain, we consider the comparison of the two models to be well suited to address the question at hand. We expect that the Graphormer will outperform RoBERTa in predicting the BBBP of molecules.

4 Method

In this section we describe the methodology used to compare the Graphormer and RoBERTa models in the context of predicting the BBBP of molecules. We describe the data preparation, discuss implementation choices of the chosen model and explore a number of architecture variations, as well as the hyperparameter choices for the models. Finally, we detail our training procedure and explain our evaluation metrics.

In order to fully assess the potential of the Graphormer, we conduct two different experiments. First, we train and compare a set of Graphormer and RoBERTa models initialised from random weights. Second, we fine-tune and compare a pretrained Graphormer and MFBERT. By conducting both experiments, we aim to explore the value of pretraining for the Graphormer model and to provide a more comprehensive evaluation having explored both major methods of applying the model.

The pretrained Graphormer model we chose to compare against MFBERT is trained on the OGB PCQM4Mv2 dataset [2]. It is a large-scale quantum chemistry dataset composed of 3.3 million molecules that is part of the Open Graph Benchmark (OGB). While it is much smaller in size than the dataset MFBERT was trained on, it is the most suitable pretrained weight set of the Graphormer available.

4.1 Models Implementation

The Graphormer model implementation was released by Microsoft on GitHub [8]. The model was built using the Fairseq framework [19], a toolkit designed to allow researchers to train sequential models from the command line. This is not ideal as the model implementation was abstracted and very difficult to work with and pair with other libraries and tools. Fortunately, in January of 2023 a Graphormer implementation was added to the HuggingFace `transformers` library [35]. We use the `4.27.0.dev0` version of the development branch of the library, as the model was not released on a stable branch directly. Given the recent implementation, the usage of the model is poorly documented, however, the architecture was standardised to fit with other HuggingFace models and extensions which made it easier to understand how to work with it.

In its study, MFBERT was implemented using the RoBERTa model from the HuggingFace `transformers` library. For consistency, we chose to use the same architecture with both the MFBERT and the Graphormer implementations so that any experiments or modifications can be carried out on all models in the same manner.

4.2 Data Preparation

We chose to train the models with the B3DB dataset [17]. It is compiled from 50 different sources, making it the largest publicly available BBBP dataset in literature. A small subset contains numerical values denoting the value of the logarithm of brain-plasma concentration ratio. The main part of the dataset, however, contains categorical labels denoting whether a compound is BBB permeable (BBB+) or not (BBB-). We use only the categorical part as each molecule in the numerical side also has a categorical label attached to it.

The dataset contains 7807 small molecules with labels attached (4956 BBB+ and 2851 BBB-) in the form of SMILES. We filter out monoatomic molecules with a script which attempts to convert each SMILES into an RDKit [14] `Mol` object and checks the number of bonds. If no bonds are found, the molecule is discarded. We discard monoatomic molecules for compatibility with the Graphormer.

This method is useful for filtering invalid molecules as well, as RDKit will fail to convert SMILES strings into `Mol` objects if the SMILES describe an invalid molecule (e.g. a molecule with atoms with valences greater than realistically possible), which show up in the benchmark dataset we later use.

Once the dataset has been filtered, we shuffle the dataset and split it with an 8:1:1 train validation test ratio. The same splits are used for each model to ensure the comparison is fair and unbiased.

Since both the Graphormer and RoBERTa implementations we use come from the HuggingFace library, we chose to use their `Dataset` and `DatasetDict` objects, found in the HuggingFace `datasets` [10] library to handle and store the data.

4.2.1 Tokenizing SMILES

As SMILES cannot be split into words, the BPE tokenizer RoBERTa uses cannot be applied. A unigram SentencePiece [13] tokenizer is used instead. SentencePiece treats the input as a raw stream of characters, as opposed to BPE which assumes the input is plain text and splits it into words beforehand. We use the pretrained tokenizer configuration and vocabulary used by MFBERT.

4.2.2 Converting SMILES to Graphs

To ensure the best compatibility and performance when fine-tuning the PCQM4Mv2 Graphormer, we chose to convert the SMILES dataset to graphs using OGB's method. The conversion algorithm uses RDKit [14], a cheminformatics package which provides tools for working with molecules and chemical data in Python and C++, to extract a number of descriptors for each atom and bond in the molecule and build a Python dictionary representing the graph.

The final graph structure is detailed in table 1:

Key	Description
`edge_index`	A list comprised of two lists of ints. They hold the indices of the nodes each edge points at. Example: given a graph with nodes `A, B, C` and connections `A` $\rightarrow$ `C` $\leftrightarrow$ `B` $\rightarrow$ `A`, the edge index array would be `[[A, C, B, B], [C, B, C, A]]`.
`edge_attr`	List of lists of ints. Each list represents an edge (bond) feature vector. Each holds the following features: • bond type • bond stereochemistry • value indicating whether the bond is part of a conjugated system
`node_feat`	List of lists of ints. Each list represents a node (atom) feature vector. Each holds the following features: • atomic number • atom stereochemistry • atom degree (number of directly bonded molecules, including H atoms) • atom formal charge • total number of H atoms linked to atom • number of unpaired electrons of atom • atom hybridization • value indicating whether the atom is part of an aromatic system • value indicating whether the atom is in a ring
`num_nodes`	Int. Number of atoms in molecule.
`smiles`	String. SMILES of molecule.
`y`	List of one int, 0 or 1. Value denoting whether or not it penetrates the BBB.

Table 1: Dictionary fields of the graph representation of molecules.

Once the dataset has been converted into graphs, a preprocessing function included in the HuggingFace `transformers` library needs to be applied to each graph in order to compute additional information needed by the Graphormer (e.g. spatial information).

4.3 Architecture Modifications Exploration

When dealing with a small dataset such as B3DB it can be challenging to train a model that can accurately capture the underlying patterns in the data. This is especially true for Transformer-based models which have a large number of parameters making them very computationally heavy and prone to overfitting. To overcome this challenge, we explore three different architecture variations to find the option which can achieve the best performance. These architecture variations are as follows:

- `NORMAL`: this is the base architecture as used in the papers introducing the Graphormer and RoBERTa models. It is also the same architecture the pretrained PCQM4Mv2 Graphormer and MFBERT models use.
- `SMALL`: the small architecture uses fewer layers and reduces the dimensionality of the encoder (hidden size) down to that of the original Transformer. Reducing the number of layers can have significant impact on the input data as attention is calculated fewer times. The main effect of computing attention fewer times is reducing the model's ability to capture complex patterns, however it also prevents it from capturing noise or irrelevant connections between tokens.
- `SLIM`: the slim model greatly reduces the dimensionality of the encoder and the number of attention heads in order to greatly reduce the total parameter size. Reducing the encoder dimensionality enough will impede the expressivity of the final representations. The purpose of this variation is to avoid overfitting by using a greatly diminished parameter number.

The architecture configurations for these variations can be found in table 2.

Model	**Attention Heads**	**Layers**	**Hidden Size**	**Total Parameters**
Graphormer	32	12	768	47,676,737
RoBERTa	12	12	768	87,899,906
$\text{Graphormer}_{\text{SMALL}}$	32	6	512	22,281,793
$\text{RoBERTa}_{\text{SMALL}}$	8	6	512	26,977,794
$\text{Graphormer}_{\text{SLIM}}$	8	12	80	952,513
$\text{RoBERTa}_{\text{SLIM}}$	8	12	80	6,492,306
PCQM4Mv2 Graphormer	32	12	768	47,676,737
MFBERT	12	12	768	87,899,906

Table 2: Parameters for each model architecture vairation.

4.4 Hardware, Optimization, and Logging

We performed all experiments using a standard laptop with an Intel i7-9750 processor, a NVIDIA GeForce RTX 2060 Mobile GPU and 16 GB of RAM. Although the laptop's hardware is not as performant as a dedicated server, we were able to successfully perform all experiments. Our choice of hardware reflects the practical constraints faced by many researchers in the field and highlights the importance of designing models that are robust and efficient in a variety of settings. We used a Conda environment with Python 3.9.15 and CUDA 11.8 for GPU acceleration on Manjaro Linux.

Due to the limited hardware resources, we used a smaller batch size of 4 samples per batch size and 16 gradient accumulation steps to reduce the computational cost. Gradient accumulation is a technique where gradients are accumulated over multiple small batches, instead of being computed and updated after each batch. Our optimization scheme effectively simulates a training batch size of 64 samples, which enables us to balance training efficiency and model performance under constrained hardware resources.

The use of Google Colaboratory [9], a web environment for prototyping and training machine learning models on remote GPUs and TPUs, was considered and experimented with. We decided against using Google Colaboratory as the performance was inadequate, disk reading and writing times were poor and the overall efficiency was greatly diminished.

To log the performance of each trained model accross all experiments we used WandB [3], a platform for visualizing and tracking the performance of machine learning models.

4.5 Training Strategy

As previously stated, we have four main models to train (or fine-tune): Graphormer, RoBERTa, PCQM4Mv2 Graphormer and MFBERT. As predicting the BBBP is a binary classification task, in order to train the models, we fit a classification head on top of the encoders. It is a two layer fully connected feed forward network whose hidden layer is the same size as the output embeddings and the output layer is two nodes, one for each class. The model is then trained using binary cross entropy loss as the loss function.

To ensure the best performance for each model, we employed a random hyperparameter search strategy to explore the most optimal training settings. We ran 35 trials of two epochs each for each of the four models types, training a total of 140 individual models, randomly sampling from a pre-defined hyperparameter grid. This allowed us to evaluate the impact of different hyperparameters on model performance and identify the best-performing models for further exploration. Each trial was logged on the WandB platform for later analysis.

The hyperparameter space explored is detailed in table 3.

Parameter	**Grid**	**Description**
architecture	`["normal", "small", "slim"]`	Architecture modifications explorations described in section 4.3. Does not apply to the pretrained models.
learning rate	`(1e-5, 1e-3)`	The step size at each iteration. Chosen at random from a log-uniform distribution with defined range.
warm-up steps	`[0, 20, 60, 100]`	Number of steps over which to gradually increase learning rate to the predefined value in order to prevent impact from sudden new data exposure. Values represent 0%, ∼10%, ∼30% and ∼50% of total training steps.
dropout rate	`[0, 0.1, 0.25]`	The probability of ignoring a neuron during training. Not changed for pretrained models as their dropout rate is already chosen in the pre-defined config.
weight decay	`[0, 0.1, 0.25, 0.5]`	Term added to weights which encourages smaller weights and prevents overfitting.

Table 3: Hyperparameter space.

The number of epochs are not explicitly explored as the best iteration of each model in the trials was saved at each evaluation step. Evaluation occured every 20 steps and on the final step of training. Following the 140 2-epoch trials, the parameter sets of the top three trials for each model type were retrained for five epochs and evaluated using the same logging and saving strategy. The best-performing model for each model category was then trained for an additional five epochs. This allows us to efficiently determine the best number of epochs.

4.6 Evaluation Metrics

We measure the performance of the models using the following evaluation metrics:

- **ROC-AUC**: The area under the curve of the receiver operating characteristic. It is a performance metric commonly used to evaluate classification models which measures the ability of a model to distinguish between positive and negative classes. The ROC curve is a plot of the true positive rate against the false positive rate at various classification thresholds. The ROC-AUC score ranges between 0 and 1, where 1 is a perfect performance and 0.5 indicates the model is only guessing at random. MoleculeNet recommends the ROC-AUC score as the preferred evaluation metric for predicting the BBBP [18].
- **Accuracy**: Measures the percentage of correctly predicted labels out of the total number of predictions made by the model on a given dataset:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}^{1} \tag{10}$$

- **Precision**: The ratio of correctly predicted positive labels to the total predicted positive labels. It measures how reliable the model is for *predicting* positive labels:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{11}$$

- **Recall**: The ratio of correctly predicted positive labels to the total actual positive labels. It measures how reliable the model is for *detecting* positive labels:

$$\text{Recall} = \frac{TP}{TP + FN} \tag{12}$$

- **F1 Score**: The harmonic mean of precision and recall, combining both metrics in order to indicate how well the model balances between the two. A high F1 score indicates that the model is able to correctly identify both positive and negative instances with high accuracy:

$$\text{F1} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{13}$$

[1] Where TP = number of true positives, TN = number of true negatives, FP = number of false positives, FN = number of false negatives

5 Results

In this section we detail the results obtained from the experiments performed. We split this section in two subsections. First, we explain the hyperparameter search experiments. For each model type, we create a parallel coordinate plot showing the parameter sets tried, look at parameter importance and correlation with the ROC-AUC score and, for the non-pretrained models, we examine the best architecture variations. The ROC-AUC scores and other evaluation metrics are all calculated on the validation dataset.

5.1 Hyperparameter Search

From the first run of 140 2-epoch trials, a number of trials resulted in a ROC-AUC score of 0.5, indicating that the model's performance was no better than random guessing. This suggests that these trials failed to learn any meaningful representation of the data, which can occur when a poor set of hyperparameters is selected. We consider the models with a ROC-AUC of 0.5 failed. Table 7 displays the ROC-AUC scores and their statistical measures for each model.

5.1.1 Graphormer and RoBERTa

We first take a look at the trials of the Graphormer and RoBERTa models initialised from random weights and trained only on the task dataset. Figures 9 and 10 show the parallel coordinate plots for their trials.

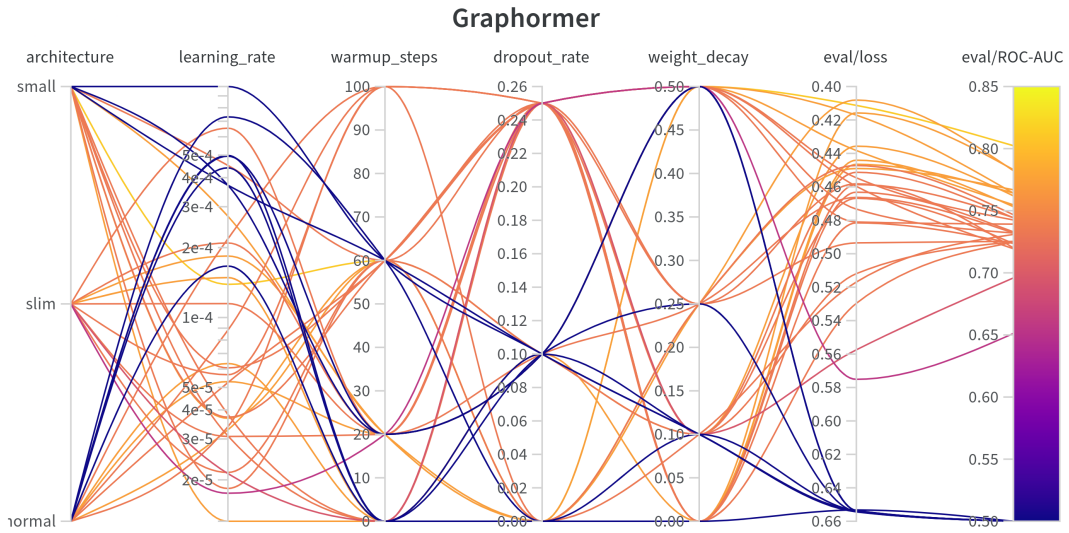

Figure 9: Parallel coordinate plot of Graphormer hyperparameters.

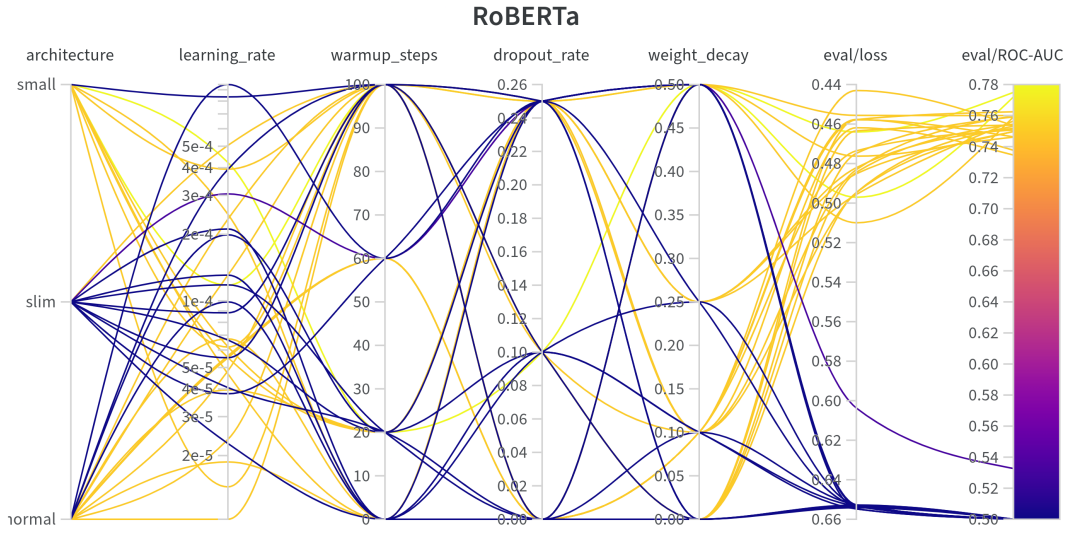

Figure 10: Parallel coordinate plot of RoBERTa hyperparameters.

The outcomes of the Graphormer model display a higher level of variability compared to those of

RoBERTa. This suggests that the Graphormer model has greater potential but also that greater optimization efforts are required to achieve more consistent performance on a small dataset. The results from RoBERTa demonstrate comparatively lower variance, allowing for more consistent expectations. This could also mean that the model is restricted in its performance on smaller datasets. With little training time, the RoBERTa model can perform better on average, but can already be well outperformed by the Graphormer.

5.1.2 Architecture Variation Impact

The importance and correlation of each architecture choice with the ROC-AUC score were computed. A tree-based model was used to determine the importance, while a linear model was used to assess the correlation. The computed scores for these measures are as detailed in table 4. Figures 11 and 12 show the mean, maximum and minimum ROC-AUC score of each architecture choice, as well as number of failed trials in table 5.

Parameter	**Graphormer**		**RoBERTa**	
	Importance	Correlation	Importance	Correlation
Architecture: `NORMAL`	0.007	0.105	0.001	0.180
Architecture: `SMALL`	0.019	0.316	0.002	0.222
Architecture: `SLIM`	0.084	-0.405	0.462	-0.680

Table 4: Architecture importance and correlation with ROC-AUC scores.

Architecture	**Mean**	**Std Dev**	**Min**	**Max**	**Failed**
Graphormer					
`NORMAL`	0.7457	0.0234	0.7194	0.7831	5
`SMALL`	0.7512	0.0253	0.7236	0.8024	3
`SLIM`	0.7255	0.0331	0.6511	0.7669	0
RoBERTa					
`NORMAL`	0.7528	0.0081	0.7346	0.7624	4
`SMALL`	0.7561	0.0119	0.7376	0.7749	1
`SLIM`	0.6430	0.1559	0.5327	0.7532	9

Table 5: Architecture summary. Failed denotes the number of failed trials (ROC-AUC of 0.5).

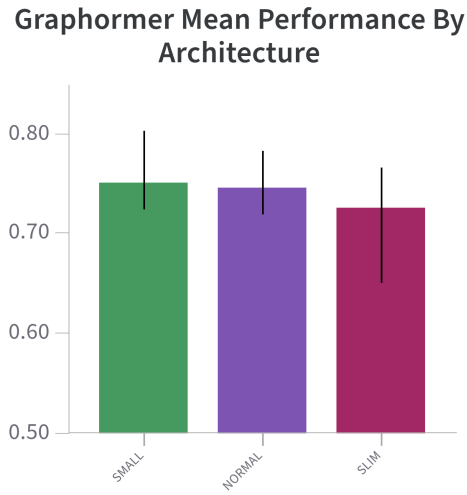

Figure 11: Mean, min and max performance by architecture for Graphormer.

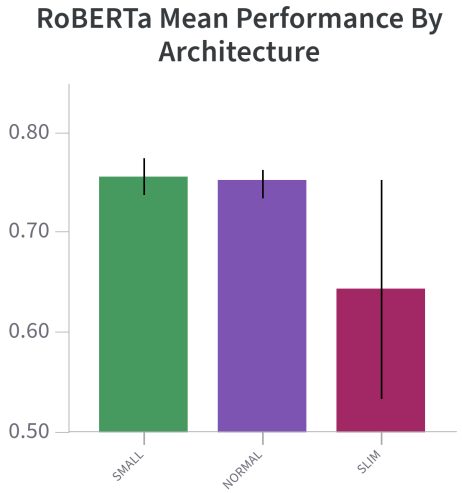

Figure 12: Mean, min and max performance by architecture for RoBERTa.

The `SMALL` variant performed the best while the `SLIM` variant performed the worst for both the Graphormer and RoBERTa models. With the reduced size of the dataset, it is plausible that fewer attention calculations are required to fully understand the relationships of the data. The difference between the `SMALL` and `NORMAL` variants, despite being small, is still significant. Besides the better performance, the `SMALL` is also much more efficient, to its model size being less than half of the `NORMAL` variant. The `SLIM` variant has performed the worst in both cases. Its greatly reduced dimensionality had a detrimental effect on the expressiveness of the molecular representations. It is worth noting that none of the $\text{Graphormer}_{\text{SLIM}}$ trials failed, managing to make good use of the smaller encodings, whereas most $\text{RoBERTa}_{\text{SLIM}}$ trials failed.

5.1.3 PCQM4Mv2 Graphormer and MFBERT

We now look at the trials fine-tuning the pretrained PCQM4Mv2 and MFBERT models. Figures 13 and 14 show the parallel coordinate plots for the trials.

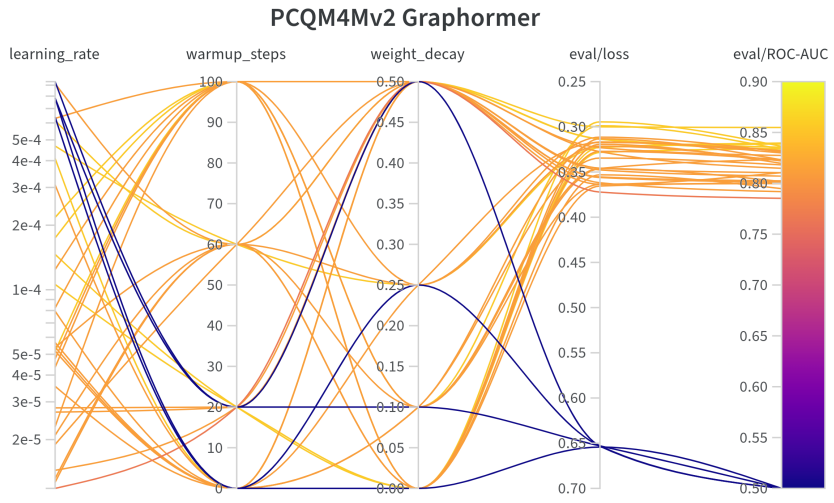

Figure 13: Parallel coordinate plot of PCQM4Mv2 Graphormer hyperparameters.

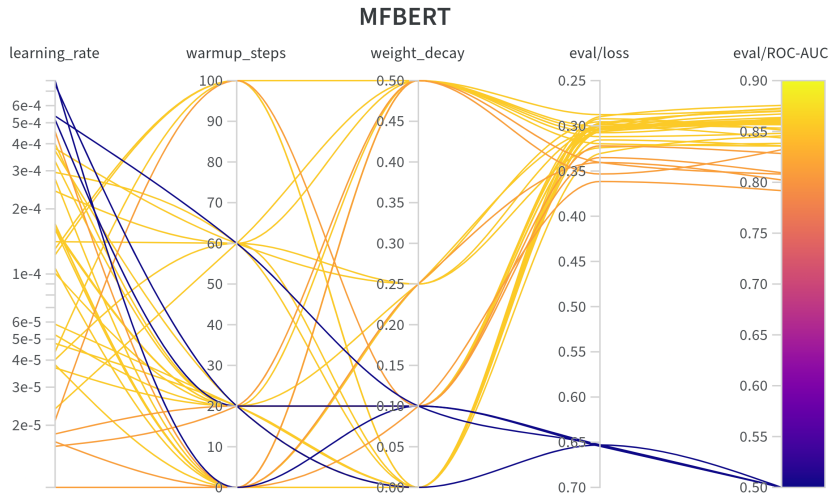

Figure 14: Parallel coordinate plot of MFBERT hyperparameters.

Even after only two epochs, much higher ROC-AUC scores and a smaller variation can be noticed. This can be attributed to the pretraining of the models to represent the input data effectively, leaving only the task of identifying relevant patterns and relationships in the input data required for BBBP prediction. Given that fine-tuning is primarily an optimization challenge, the importance

of each hyperparameter is evaluated by computing its correlation with ROC-AUC scores.

Parameter	PCQM4Mv2 Graphormer		MFBERT	
	Importance	Correlation	Importance	Correlation
learning rate	0.573	-0.252	0.938	-0.547
warmup steps	0.226	0.156	0.033	0.036
weight decay	0.201	0.204	0.029	0.217

Table 6: Hyperparameter importance and correlation with ROC-AUC scores when fine-tuning.

The learning rate has the most significant effect on fine-tuning, with a larger learning rate being more detrimental to the final ROC-AUC score. A large learning rate makes the model change the weights excessively during training, which can negatively impact the pretrained weights. Specifically, when the learning rate is set too high, the model's ability to optimize the loss function can be impeded due to the frequent changes in the weights. The model may overshoot the optimal weights and produce poor results.

Model Type	Count	Mean	Std Dev	Min	25%	50%	75%	Max
Graphormer	27	0.7403	0.0289	0.6511	0.7248	0.7392	0.7584	0.8024
RoBERTa	20	0.7543	0.0094	0.7346	0.7498	0.7537	0.7593	0.7749
PCQM4Mv2 Gr.[2]	30	0.8192	0.0166	0.7852	0.8035	0.8225	0.8331	0.8549
MFBERT	31	0.8489	0.0216	0.7918	0.8371	0.8579	0.8632	0.8756

Table 7: Statistics of the ROC-AUC score for models with a ROC-AUC over 0.55 for the hyperparameter search. Count represents the number of models with ROC-AUC over 0.55; and 25%, 50% and 75% represent quartiles.

The fine-tuned MFBERT performed consistently better compared to the fine-tuned PCQM4Mv2 Graphormer. We attribute this to the great imbalance in their pretraining dataset and computational resources.

5.2 Further Training

We selected the best performing three models for each model type and used their corresponding hyperparameter sets to train new models for a total of five epochs. The hyperparameters of the chosen models can be found in table 8 and the evaluation metrics of models trained for five epochs can be found in table 9. Since warmup steps originally represent a percentage of the total training steps, we scale them up accordingly.

[2]Graphormer, shortened to shrink table size.

Model	Arch.	LR.	Warmup	Dropout	W. Decay	Loss	ROC-AUC
Graphormer 1	SMALL	0.000138	150	0	0.5	0.4112	0.8023
Graphormer 2	NORMAL	0.000032	150	0	0	0.4157	0.7830
Graphormer 3	SMALL	0.000272	50	0	0.25	0.4081	0.7822
RoBERTa 1	SMALL	0.000422	50	0.1	0.5	0.4640	0.7749
RoBERTa 2	SMALL	0.000117	250	0.25	0.5	0.4970	0.7719
RoBERTa 3	NORMAL	0.000130	250	0	0.1	0.4577	0.7624
PCQM4Mv2 Gr. 1		0.000410	0	0.5		0.3000	0.8548
PCQM4Mv2 Gr. 2		0.000105	50	0		0.3225	0.8395
PCQM4Mv2 Gr. 3		0.000468	150	0.25		0.3228	0.8387
MFBERT 1		0.000162	50	0.1		0.2898	0.8756
MFBERT 2		0.000294	150	0		0.3040	0.8728
MFBERT 3		0.000130	250	0.5		0.2876	0.8722

Table 8: Hyperparameter sets for best trials for each model types. Names shortened, arch. is architecture, LR. is learning rate, warmup is warmup steps, dropout is dropout rate, w. decay is weight decay.

Model	ROC-AUC	Accuracy	F1 Score	Loss	Precision	Recall
MFBERT 2	**0.881**	0.8897	0.9138	0.3273	0.9157	0.912
PCQM4Mv2 Gr. 3	**0.8779**	0.8897	0.9145	0.3031	0.9091	0.92
MFBERT 3	**0.8745**	0.8885	0.9139	0.3463	0.9041	0.924
MFBERT 1	**0.8705**	0.8833	0.9096	0.3296	0.9034	0.916
PCQM4Mv2 Gr. 2	**0.8586**	0.8782	0.9071	0.363	0.8872	0.928
Graphormer 3	**0.8124**	0.8462	0.8859	0.3435	0.8442	0.932
Graphormer 1	**0.8122**	0.8449	0.8847	0.3529	0.8452	0.928
Graphormer 2	**0.81**	0.841	0.8812	0.3669	0.8456	0.92
RoBERTa 1	**0.7788**	0.8141	0.8618	0.4254	0.8233	0.904
RoBERTa 3	**0.7774**	0.8154	0.8636	0.4229	0.8201	0.912
RoBERTa 2	**0.7674**	0.8115	0.8627	0.4418	0.8091	0.924
PCQM4Mv2 Gr. 1	**0.5**	0.641	0.7813	0.653	0.641	1.0

Table 9: Evaluation metrics for each of the best models trained for five epochs, ordered best to worst.

The Graphormer models are already beginning to surpass the RoBERTa models in performance after five epochs of training. Among the models initialised from random weights, all Graphormer models perform significantly better than the RoBERTa models. In terms of pretrained and fine-tuned models, the MFBERT models are still the most performant option. However, the PCQM4Mv2 Graphormer models deliver very similar results, on par with MFBERT.

A noteworthy observation is the performance of PCQM4Mv2 Graphormer 1 which overfit after five epochs. With a recall metric of 1, the model projected all inputs as positive labels. This behavior is likely due to a high weight decay and learning rate without any warmup steps. However, given the

non-deterministic nature of training deep learning models, the exact reason for overfitting cannot be definitively established.

We save the weights of the best of each model kind, then further train them for another five epochs. We make sure to evaluate the performance during training every 50 steps (five epochs are approximately 500 steps) and save the weights as the new best model if there is an improvement in the ROC-AUC. Once the further training is complete, we load the best weights of each model and evaluate their performance on both the validation dataset, the previously unseen test dataset, and the benchmark dataset for BBBP proposed by MoleculeNet [18]. The results can be found in table 10.

Model	**ROC-AUC**	**Accuracy**	**F1 Score**	**Loss**	**Precision**	**Recall**
Validation						
PCQM4Mv2 Gr.	**0.8905**	0.9	0.9221	0.3026	0.9203	0.924
MFBERT	**0.881**	0.8897	0.9138	0.3273	0.9156	0.912
Graphormer	**0.816**	0.8487	0.8876	0.3655	0.8472	0.932
RoBERTa	**0.7788**	0.814	0.8617	0.4253	0.8233	0.904
Test						
MFBERT	**0.8604**	0.8705	0.8966	0.3580	0.8920	0.9012
PCQM4Mv2 Gr.	**0.8560**	0.8692	0.8965	0.3241	0.884	0.9094
Graphormer	**0.8238**	0.8474	0.8825	0.3510	0.8481	0.9197
RoBERTa	**0.7917**	0.8192	0.8616	0.4027	0.8236	0.9032
MoleculeNet Benchmark						
PCQM4Mv2 Gr.	**0.9162**	0.9397	0.9605		0.9602	0.9608
MFBERT	**0.8366**	0.8754	0.9183		0.9284	0.9083
Graphormer	**0.8136**	0.8798	0.9223		0.9032	0.9424
RoBERTa	**0.7052**	0.7783	0.8529		0.8596	0.8463

Table 10: Evaluation metrics for the final models, ordered best to worst for each dataset.

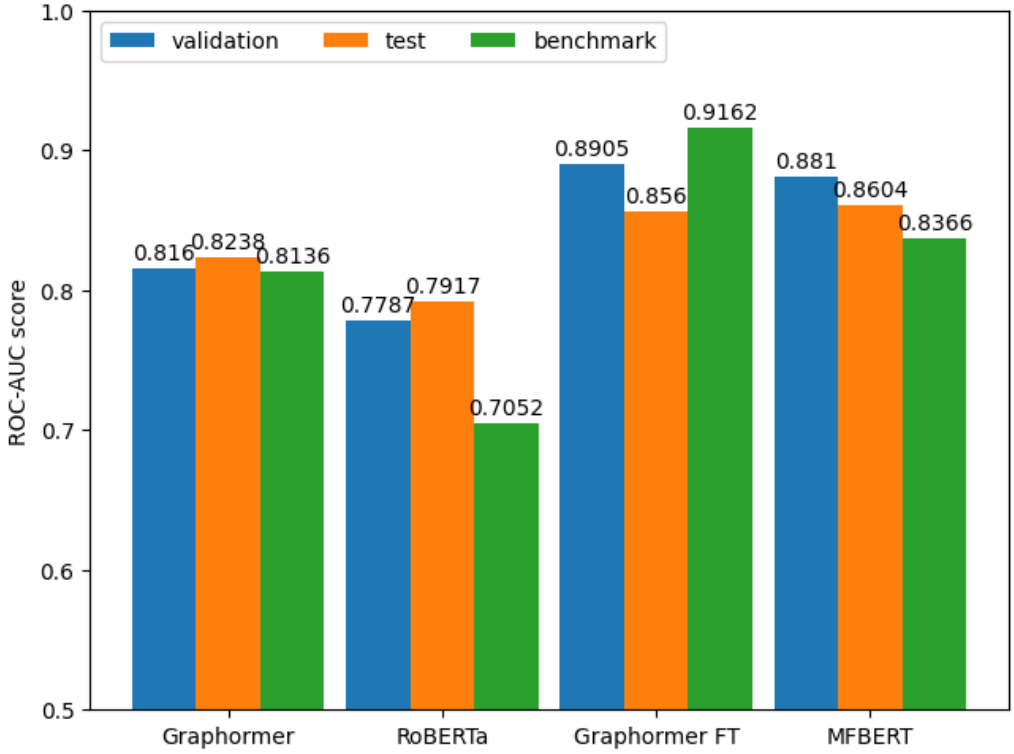

Figure 15: Validation, test and benchmark scores for final models.

The Graphormer-based models achieve better scores for nearly every metric compared to the RoBERTa-based models, greatly outperforming them on the MoleculeNet benchmark dataset. Neither the RoBERTa model nor the fine-tuned MFBERT seem to benefit from longer training times, as their best weight sets both come from the first run of five epochs. This indicates that the RoBERTa-based models are faster and more computationally inexpensive than the Graphormer to converge.

From among the models initialised from random weights, Graphormer outperforms RoBERTa across all evaluation metrics. The fine-tuned PCQM4Mv2 Graphormer also achieves higher scores than MFBERT on the validation set. However, MFBERT performs better on the test set. Nevertheless, the Graphormer model shows competitive performance against MFBERT, despite the significantly larger amount of resources MFBERT used for its pretraining.

Intriguingly, the test scores for the randomly initialised Graphormer and RoBERTa are higher than the validation scores. It is unusual for a model to perform better on the test data split than on the validation split. One plausible explanation for this is that the test dataset split may have represented an easier subset of the data due to random chance, given that the dataset was randomly shuffled before splitting. Since both the test and validation splits were of small size, with only 780

samples each, this explanation appears reasonable.

6 Conclusion

In this report we hypothesized about the potential of a graph-based approach at applying Transformers on the drug discovery problem of predicting a molecule's BBBP. We give a thorough explanation of the Transformer architecture, reasoning why it achieves state of the art results and why they are suited for this problem. We consider the best candidates for comparison, deciding on the Graphormer model for its ability to process graphs, and the RoBERTa model for its proven potential in drug discovery tasks. We experiment with different architecture variations in order to optimize the models as much as possible for use on a small dataset such as B3DB and determine that a smaller variation, with fewer layers, is better suited for limited datasets. We also compare a set of Graphormer and RoBERTa models initialised from pretrained weights that we further fine-tune for our problem. These models were pretrained on molecular datasets with the purpose of representing molecules as vectors of numbers. To validate our hypothesis, we ran a number of thorough experiments in order to assess the performance of these two models.

Finally, we conclude that incorporating graph inputs into Transformer-based models does help improve their performance on drug discovery tasks. The Graphormer models we trained consistently performed better than the RoBERTa models. Even when constrained by limited embedding space, the Graphormer managed to attain great results, proving it more versatile and adaptable. Despite the extensive amount of resources that went into pretraining MFBERT, as well as its dataset of 1.2 billion molecules, which the Graphormer pretrained on the PCQM4Mv2 dataset lacked, it still greatly outperformed MFBERT on the MoleculeNet benchmark for the BBBP prediction task.

References

[1] Hisham Abdel-Aty and Ian R. Gould. "Large-Scale Distributed Training of Transformers for Chemical Fingerprinting". In: *Journal of Chemical Information and Modeling* 62.20 (2022). PMID: 36195574, pp. 4852–4862. DOI: 10.1021/acs.jcim.2c00715. eprint: https://doi.org/10.1021/acs.jcim.2c00715. URL: https://doi.org/10.1021/acs.jcim.2c00715.

[2] Open Graph Benchmark. *PCQM4Mv2*. URL: https://ogb.stanford.edu/docs/lsc/pcqm4mv2/ (visited on 04/05/2023).

[3] Lukas Biewald. *Experiment Tracking with Weights and Biases*. Software available from wandb.com. 2020. URL: https://www.wandb.com/.

[4] Jason Brownlee. *A Gentle Introduction to the Rectified Linear Unit (ReLU)*. URL: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/ (visited on 04/01/2023).

[5] Laurianne David et al. "Molecular representations in AI-driven drug discovery: a review and practical guide". In: *Journal of Cheminformatics* 12.1 (Sept. 2020). DOI: 10.1186/s13321-020-00460-5. URL: https://doi.org/10.1186/s13321-020-00460-5.

[6] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: (2018). DOI: 10.48550/ARXIV.1810.04805. URL: https://arxiv.org/abs/1810.04805.

[7] David Duvenaud et al. "Convolutional Networks on Graphs for Learning Molecular Fingerprints". In: (2015). arXiv: 1509.09292 [cs.LG].

[8] Microsoft GitHub. *microsoft/Graphormer*. URL: https://github.com/microsoft/Graphormer (visited on 04/05/2023).

[9] *Google Colaboratory*. URL: https://colab.research.google.com/ (visited on 04/08/2023).

[10] HuggingFace. *Datasets*. URL: https://huggingface.co/docs/datasets/index (visited on 04/05/2023).

[11] HuggingFace. *Tokenizers Course*. URL: https://huggingface.co/course/chapter2/4 (visited on 03/31/2023).

[12] Amirhossein Kazemnejad. *Transformer Architecture: The Positional Encoding*. URL: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/ (visited on 03/31/2023).

[13] Taku Kudo and John Richardson. "SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing". In: (2018). arXiv: 1808.06226 [cs.CL].

[14] Greg Landrum. *RDKit: Open-source cheminformatics*. URL: http://www.rdkit.org (visited on 03/31/2023).

[15] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. DOI: 10.48550/ARXIV.1907.11692. URL: https://arxiv.org/abs/1907.11692.

[16] Ines Filipa Martins et al. "A Bayesian Approach to in Silico Blood-Brain Barrier Penetration Modeling". In: *Journal of Chemical Information and Modeling* 52.6 (June 2012), pp. 1686–1697. DOI: 10.1021/ci300124c. URL: https://doi.org/10.1021/ci300124c.

[17] Fanwang Meng et al. "A curated diverse molecular database of blood-brain barrier permeability with chemical descriptors". In: *Scientific Data* 8.1 (Oct. 2021), p. 289. ISSN: 2052-4463. DOI: 10.1038/s41597-021-01069-5. URL: https://doi.org/10.1038/s41597-021-01069-5.

[18] *MoleculeNet*. URL: https://moleculenet.org/ (visited on 04/08/2023).

[19] Myle Ott et al. "fairseq: A Fast, Extensible Toolkit for Sequence Modeling". In: (2019). arXiv: 1904.01038 [cs.CL].

[20] Aravindpai Pai. *What is Tokenization in NLP? Here's All You Need To Know*. URL: https://www.analyticsvidhya.com/blog/2020/05/what-is-tokenization-nlp/ (visited on 03/31/2023).

[21] Sheryl Cherian Parakkal, Riya Datta, and Dibyendu Das. "DeepBBBP: High Accuracy Blood-brain-barrier Permeability Prediction with a Mixed Deep Learning Model". In: *Molecular Informatics* 41.10 (Apr. 2022), p. 2100315. DOI: 10.1002/minf.202100315. URL: https://doi.org/10.1002/minf.202100315.

[22] Brandon Rohrer. *Transformers from Scratch*. URL: https://e2eml.school/transformers.html (visited on 03/31/2023).

[23] Hiroshi Sakiyama, Motohisa Fukuda, and Takashi Okuno. "Prediction of Blood-Brain Barrier Penetration (BBBP) Based on Molecular Descriptors of the Free-Form and In-Blood-Form Datasets". In: *Molecules* 26.24 (Dec. 2021), p. 7428. ISSN: 1420-3049. DOI: 10.3390/molecules26247428. URL: http://dx.doi.org/10.3390/molecules26247428.

[24] Franco Scarselli et al. "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* 20 (2009), pp. 61–80.

[25] Rico Sennrich, Barry Haddow, and Alexandra Birch. "Neural Machine Translation of Rare Words with Subword Units". In: (2016). arXiv: 1508.07909 [cs.CL].

[26] *seq2seq Model in Machine Learning*. URL: https://www.geeksforgeeks.org/seq2seq-model-in-machine-learning/ (visited on 03/31/2023).

[27] Bilal Shaker et al. "LightBBB: computational prediction model of blood-brain-barrier penetration based on LightGBM". In: *Bioinformatics* 37.8 (Oct. 2020), pp. 1135–1139. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btaa918. eprint: https://academic.oup.com/bioinformatics/article-pdf/37/8/1135/38156029/btaa918.pdf. URL: https://doi.org/10.1093/bioinformatics/btaa918.

[28] Yu Shi et al. "Benchmarking Graphormer on Large-Scale Molecular Modeling Datasets". In: (2022). DOI: 10.48550/ARXIV.2203.04810. URL: https://arxiv.org/abs/2203.04810.

[29] Manvi Singh et al. "A classification model for blood brain barrier penetration". In: *Journal of Molecular Graphics and Modelling* 96 (May 2020), p. 107516. DOI: 10.1016/j.jmgm.2019.107516. URL: https://doi.org/10.1016/j.jmgm.2019.107516.

[30] Claudia Suenderhauf, Felix Hammann, and Jörg Huwyler. "Computational Prediction of Blood-Brain Barrier Permeability Using Decision Tree Induction". In: *Molecules* 17.9 (Aug. 2012), pp. 10429–10445. DOI: 10.3390/molecules170910429. URL: https://doi.org/10.3390/molecules170910429.

[31] TensorFlow. *Neural machine translation with a Transformer and Keras*. URL: https://www.tensorflow.org/text/tutorials/transformer (visited on 03/31/2023).

[32] Jakob Uszkoreit. *Transformer: A Novel Neural Network Architecture for Language Understanding*. URL: https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html (visited on 04/01/2023).

[33] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[34] Zhuang Wang et al. "In Silico Prediction of Blood-Brain Barrier Permeability of Compounds by Machine Learning and Resampling Methods". In: *ChemMedChem* 13.20 (Sept. 2018), pp. 2189–2201. DOI: 10.1002/cmdc.201800533. URL: https://doi.org/10.1002/cmdc.201800533.

[35] Thomas Wolf et al. *HuggingFace's Transformers: State-of-the-art Natural Language Processing*. 2020. arXiv: 1910.03771 [cs.CL].

[36] Yonghui Wu et al. "Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation". In: (2016). arXiv: 1609.08144 [cs.CL].

[37] Chengxuan Ying et al. "Do Transformers Really Perform Bad for Graph Representation?" In: *CoRR* abs/2106.05234 (2021). arXiv: 2106.05234. URL: https://arxiv.org/abs/2106.05234.

A Project Details

The project can be found on the University's GitLab at the url: `https://git.`[redacted]. The `README.md` file in the repository offers detailed information on how to replicate the environment and run the code. It can be found at `https://git.`[redacted]`/blob/master/README.md`

B Prediction Server

This appendix presents a prediction server we made that showcases the final models trained. The prediction server was designed to provide a user-friendly interface for making predictions using the AI models described in the report. The server is built using a Django 4.1.7 backend and plain HTML and CSS for the frontend in order to keep loading times to a minimum and the interface concise and simple to use. jQuery is used to handle asynchronous HTTP requests between the frontend and the server.

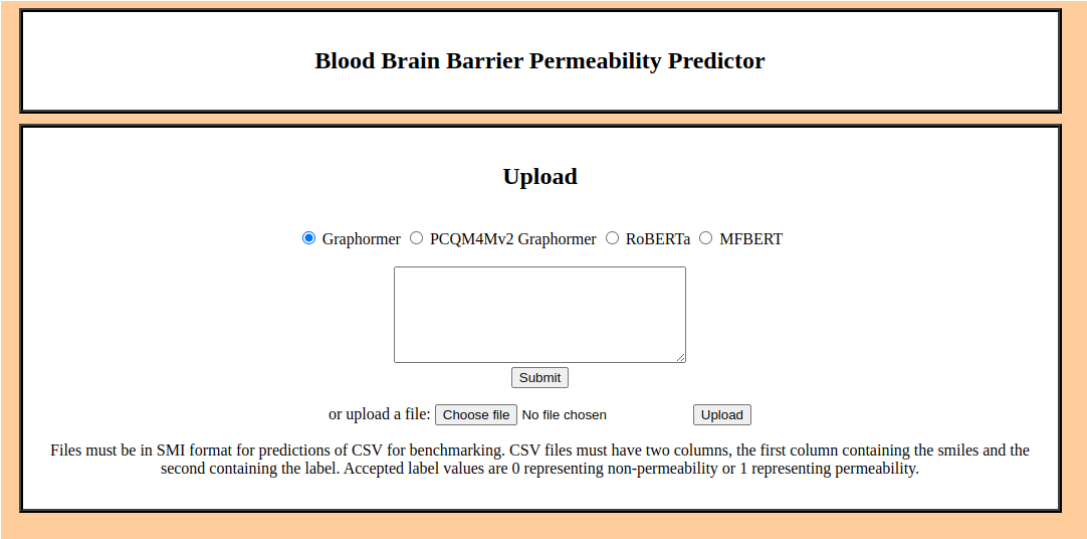

Figure 16: Server front (and only) page.

The server is able to offer two services, inference and benchmarking.

B.1 Inference

There are two ways to predict labels from data, either by pasting SMILES separated by newlines in the textbox, or by uploading a .SMI[3] file. Once the data is added, select the preferred model to use for inference from the options above and click upload (if a file is used) or submit (if the textbox is used).

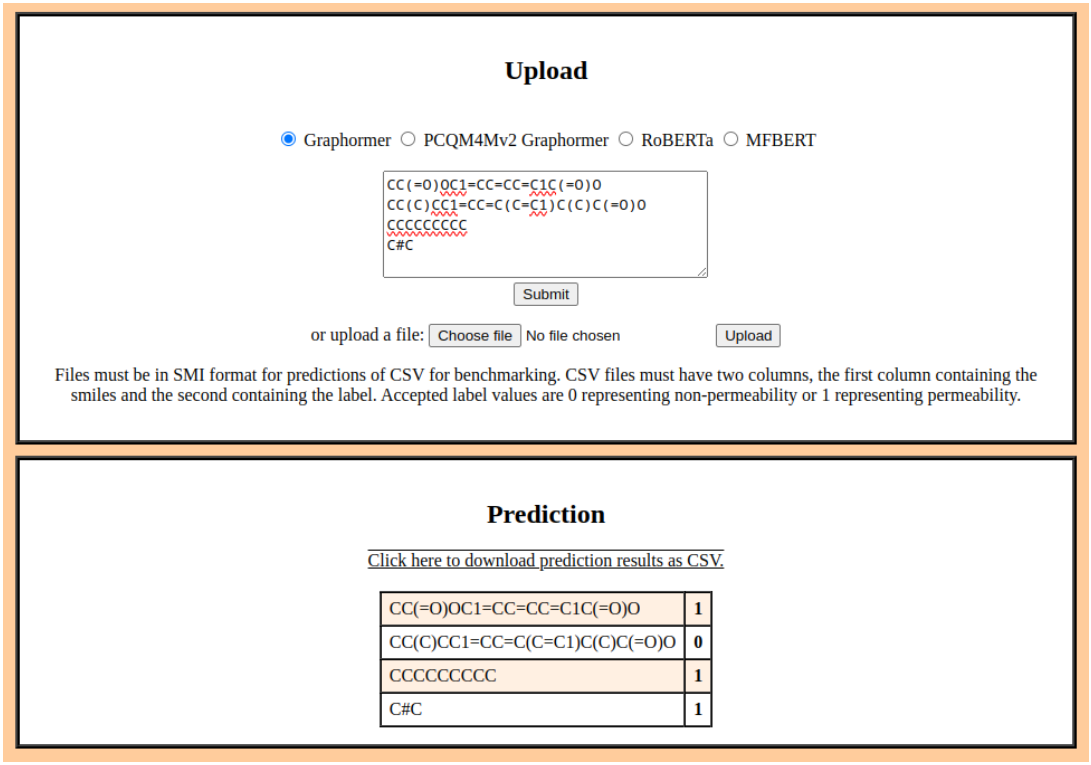

Figure 17: Textbox inference example.

A new "Prediction" section will appear, showing a table with the predicted results, as well as an option to download the results as a .CSV file. The .CSV contains two columns, "SMILES" which holds the submitted SMILES, and "bbb" which holds the labels, 1 denotes a positive label, 0 denotes a negative label.

[3] A .SMI file contains molecule SMILES separated by newlines. It is identical to the input the textbox takes.

```
SMILES,bbb
CC(=O)OC1=CC=CC=C1C(=O)O,1
CC(C)CC1=CC=C(C=C1)C(C)C(=O)O,0
CCCCCCCCC,1
C#C,1
```

Figure 18: Inference .CSV output example.

B.2 Benchmarking

In order to benchmark all four models, a .CSV file can be uploaded instead. The uploaded .CSV must have two columns, the first containing molecule SMILES and the second their labels. Once uploaded, a "Benchmark" section will appear, providing the evaluation metrics for the dataset computed on each of the four models. The "Prediction" section also appears, showing the prediction of each of the models alongside their real labels.

Benchmark

Click here to download benchmark results as CSV.

Model	ROC-AUC	accuracy	f1	precision	recall
Graphormer	0.8	0.8182	0.8571	0.75	1
Graphormer (PCQM4Mv2)	0.9167	0.9091	0.9091	1	0.8333
RoBERTa	0.7333	0.7273	0.7273	0.8	0.6667
MFBERT	0.8333	0.8182	0.8	1	0.6667

Prediction

Click here to download prediction results as CSV.

SMILES	Real	Graphormer	Graphormer (PCQM4Mv2)	RoBERTa	M
C(=O)(OC(C)(C)C)CCCc1ccc(cc1)N(CCCl)CCCl	1	1	1	1	0
c1cc2c(cc(CC3=CNC(=NC3=O)NCCSCc3oc(cc3)CN(C)C)cc2)cc1	0	1	0	0	0
CC(C)(C)NCC(O)COc1cccc2C[C@@H](O)[C@@H](O)Cc12	1	1	0	0	0
CCC(=O)N(c1ccccc1)C2(CCN(CCc3sccc3)CC2)COC	1	1	1	1	1
CCNC(=NC#N)NCCSCc1c(cccn1)Br	0	1	0	1	0
CC(=O)OC1C=CC(C)=CCC(O)C=CC(C)=CC(NC(=O)C(C)=O)C2(C)C(=O)OC(C1)C(C)C2=O	0	0	0	0	0
CC(=O)OC1CC2(C)C(CC(O)C3C4(C)CCC(O)C(C)C4CCC32C)C1=C(CCC=C(C)C)C(=O)O	0	0	0	0	0
CC(=O)OC1CC2C(COC(=O)CC(C)C)=COC(OC(=O)CC(C)C)C2C12CO2	1	1	1	0	1
CC(=O)OC1CCC2(C)C3CCC4(C)C(C(C)CCCC(C)C)CCC4C3CC(Br)C2(Br)C1	1	1	1	1	1
CC1(Cn2ccnn2)C(C(=O)O)N2C(=O)CC2S1(=O)=O	0	0	0	0	0
CC1(c2ccccc2)OC2CC3C4CCC5=CC(=O)C=CC5(C)C4(F)C(O)CC3(C)C2(C(=O)CO)O1	1	1	1	1	1

Figure 19: Benchmarking example.

As before, a .CSV version of the tables displayed will be available to download.

B.3 API

The server uses an API to make async requests between the backend and the frontend. There are two API endpoints, one for inference and one for benchmarking.

- `/api/inference`

  The API endpoint used for inference. It accepts a POST request with two inputs: `smiles` is a string containing the molecule SMILES to be used for inference, separated by the `\n`

character; and `model`, containing the model identifier string for the chosen model to be used. The model identifiers are as follow:

Model	**Identifier**
Graphormer	`graphormer`
PCQM4Mv2 Graphormer	`graphormer-ft`
RoBERTa	`mfbert`
MFBERT	`mfbert-ft`

Table 11: Model identifiers for prediction server API.

The response will be a dictionary containing two entries: `status`, 0 if no errors occured, and `data`, which contains the actual response data. The data dict contains three entries: `model`, the model identifier for the model used, `predictions`, a dictionary in which the keys are the predicted SMILES and the values are their label, and `smiles_removed`, a boolean indicating whether invalid SMILES had to be removed.

```
>>> print(json.dumps(response, indent=4))
{
    "status": 0,
    "data": {
        "model": "graphormer",
        "predictions": {
            "CC1CN(C2CCC(C#N)(c3ccc(F)cc3)CC2)CCC1(C(=O)O)c1ccccc1": 1,
            "CCCC[N+]1(C)[C@@H]2CC(OC(=O)[C@@H](CO)c3ccccc3)C[C@@H]1[C@H]1O[C@H]12": 0,
            "CCN=C1Nc2ccc(C1)cc2[C@](C)(c2ccccc2)O1": 1,
            "CCNCNCCSCc1nccccc1Br.N#CN": 1,
            "CC1(C)[C@H](NC/C(=N/OCC(=O)O)c2csc(N)n2)C(=O)N1OS(=O)(=O)O": 0,
            "CC1(C)[C@H]2CC[C@@]1(C)C(=O)C2": 1,
            "CCN(CC)C(=O)[C@H]1CN2CCc3cc(OC)c(OC)cc3[C@H]2C[C@H]1OC(C)=O": 1,
            "CCN(CC)CCOC(=O)[C@H]1N2C(=O)[C@@H](NC(=O)c3c(OC)cccc3OC)[C@H]2SC1(C)C": 0
        },
        "smiles_removed": false
    }
}
```

Figure 20: Inference API example response.

- `/api/benchmark`

  The API endpoint used for benchmarking. It accepts a POST request with one input: `smiles` is a string containing the contents of the CSV file descibed above, each line separated by the `\n` character. The response is similar to the inference API, with the same `status` and `data` keys. The data dict contains three entries: `evaluation`, a dict which holds the evaluation dicts for every model, `predictions`, similar to the inference API response, except the values are dicts holding the label predicted by each model, and `smiles_removed`, a boolean indicating whether invalid SMILES had to be removed, same as the inference API. Please see figure 21 for a detailed example.

```
>>> print(json.dumps(response, indent=4))
{
    "status": 0,
    "data": {
        "evaluations": {
            "graphormer": {
                "accuracy": 0.8182,
                "precision": 0.75,
                "recall": 1,
                "f1": 0.8571,
                "ROC-AUC": 0.8
            },
            "graphormer_ft": {
                "accuracy": 0.9091,
                "precision": 1,
                "recall": 0.8333,
                "f1": 0.9091,
                "ROC-AUC": 0.9167
            },
            "mfbert": {
                "accuracy": 0.7273,
                "precision": 0.8,
                "recall": 0.6667,
                "f1": 0.7273,
                "ROC-AUC": 0.7333
            },
            "mfbert_ft": {
                "accuracy": 0.8182,
                "precision": 1,
                "recall": 0.6667,
                "f1": 0.8,
                "ROC-AUC": 0.8333
            }
        },
        "predictions": {
            "C(=O)(OC(C)(C)C)CCCc1ccc(cc1)N(CCC1)CCC1": {
                "real": 1,
                "graphormer": 1,
                "graphormer_ft": 1,
                "mfbert": 1,
                "mfbert_ft": 0
            },
            "c1cc2c(cc(CC3=CNC(=NC3=O)NCCSCc3oc(cc3)CN(C)C)cc2)cc1": {
                "real": 0,
                "graphormer": 1,
                "graphormer_ft": 0,
                "mfbert": 0,
                "mfbert_ft": 0
            },
            "CC(C)(C)NCC(O)COc1cccc2C[C@@H](O)[C@@H](O)Cc12": {
                "real": 1,
                "graphormer": 1,
                "graphormer_ft": 0,
                "mfbert": 0,
                "mfbert_ft": 0
            },
```


Figure 21: Benchmark API example response, missing a number of SMILES entries and the `smiles_removed`