HyperCode: Encoding Abstract Syntax Tree Using Hyperbolic Geometries for Code Matching Task

Anoushka Vyas
Department of Computer Science, Virginia Tech, Blacksburg, VA, USA.
anoushkav@vt.edu

ABSTRACT

Abstract Syntax Trees or ASTs are tree representations of code. They are a fundamental part of the way a compiler works. Hyperbolic spaces have recently gained momentum in the context of machine learning due to their high capacity and tree-likeliness properties. Several Euclidean graph models have been adapted to work in the hyperbolic space and the variants have shown a significant increase in performance. ASTs represent a hierarchical way of representing a piece of code and they can be encoded using hyperbolic graph models. In this project, we are going to use hyperbolic graph models to encode ASTs to increase the performance of models like CodeT5 in the tasks of code understanding and generation. Our algorithm, referred as *HyperCode*, provides an end-to-end learning of a graph model and a language model for the code matching task.

ACM Reference Format:
Anoushka Vyas. 2022. HyperCode: Encoding Abstract Syntax Tree Using Hyperbolic Geometries for Code Matching Task . In *Proceedings of Companion Proceedings of the Web Conference 2022 (WWW '22 Companion).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Graphs have been an integral part of research in the web domain. From inter-connected webpages that form the internet to the relational databases that support the software infrastructure, development of graph processing techniques and tools has been critical for several web applications [26]. Recently, the advancements in this topic has been primarily driven by the advent of neural networks, specifically, graph neural networks (GNNs).

Hyperbolic spaces, which have the capacity to embed tree structures without distortion owing to their exponential volume growth, have recently been applied to machine learning to better capture the hierarchical nature of data. Shifting the arithmetic stage of a neural network to a non-Euclidean geometry such as a hyperbolic space is a promising way to find more suitable geometric structures for representing or processing data. Owing to its exponential growth in volume with respect to its radius ([14], [13], a hyperbolic space has the capacity to continuously embed tree structures with arbitrarily low distortion ([13], [20]).

Pre-trained language models like BERT [5], GPT [17], and T5 [18] have significantly improved performance across a broad range of natural language processing (NLP) applications. They usually use a pre-train then fine-tune approach to create generic language representations from vast amounts of unlabeled data, which may subsequently be used to a variety of downstream tasks, notably ones with minimal data annotation. Many recent attempts to adapt these pre-training approaches for programming language (PL) ([22], [11], [7]) have shown promising outcomes on code-related tasks, due to their success. Despite their success, most of these models rely on either an encoder-only model like BERT ([7], [22]) or a decoder-only model like GPT [11], which are both unsuitable for generation and understanding tasks. When used for the code summarization job, CodeBERT [7] requires an extra decoder, which does not benefit from the pre-training.

CodeT5 [29] is a pre-trained encoder-decoder model that considers the token type information in code. The CodeT5 builds on the T5 architecture [18] that employs denoising sequence-to-sequence (Seq2Seq) pre-training and has been shown to benefit both understanding and generation tasks in natural language. CodeT5 is pre-trained on the CodeSearchNet [10] data following [7] that consists of both unimodal (PL-only) and bimodal (PL-NL) data on six PLs.

An AST as shown in Figure 1 is created by converting code tokens into a tree that represents the actual structure of the code. For instance, code tokens might only tell us that at this position in the code we only had a pair of "()" but an AST will tell us whether it's a function call, a function definition, a grouping or something else. In general an AST is a tree structure where every node has at least a type specifying what it is representing. For example a type could be a "Literal" that represents an actual value or a "CallExpression" that represents a function call. The "Literal" node might only contain a value while the "CallExpression" node contains a lot of additional information that might be relevant like "what is being called" (callee) or what are the arguments that are being passed in.

In summary, we present one of the first unified encoder-decoder models *HyperCode* which uses hyperbolic graph models to encode ASTs to support both code related understanding and generation tasks. The contributions of this paper are:

- We leverage CodeT5 [29], one of the first unified encoderdecoder models that supports both code-related understanding and generating tasks as well as multi-task learning.
- We propose a novel unified model that leverages the AST. The model is a combination of Hyperbolic Graph Convolution Network (HGCN) [3] which is used to encode the ASTs and a CodeT5 encoder which is finetuned on the downstream task of code matching. We also leverage the NL-PL pairs that are naturally available in source code to learn a better cross-modal alignment.

WWW '22 Companion, April 25–29, 2022, Virtual Event, Lyon, France

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

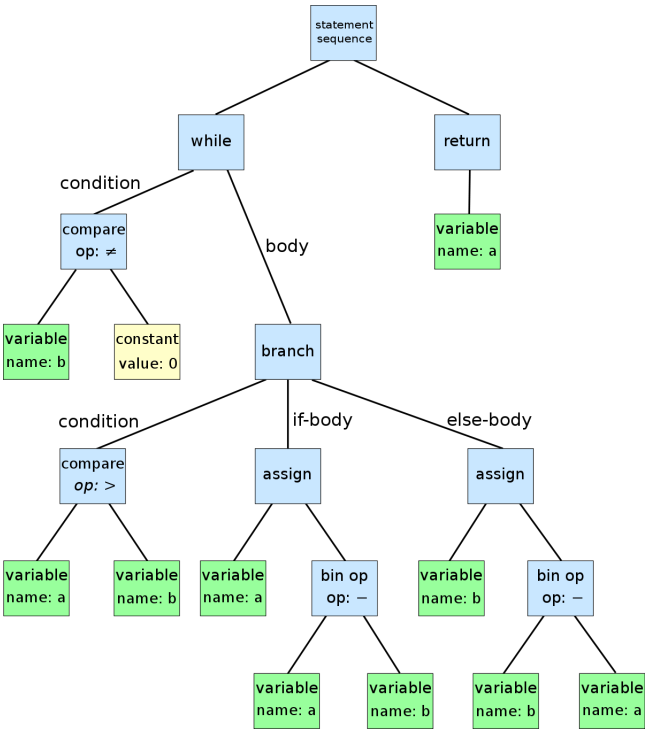

Figure 1: An example of an AST.

2 RELATED WORK

On a wide range of NLP tasks, pretrained models based on Transformer topologies [24] have achieved state-of-the-art performance. Encoder-only models like BERT [5], RoBERTa [16], and ELECTRA [4], decoder-only models like GPT [17], and encoder-decoder models like MASS [21], BART [15, 18]. Encoder-decoder models can handle both sorts of jobs better than encoder-only and decoder-only models, which favour understanding and creation tasks, respectively. Denoising sequence-to-sequence pre-training objectives are frequently used, which distort the source input and force the decoder to recover it. CodeT5 [29] is an extension to the programming language, and uses a novel identifier-aware denoising target to help the model understand the code better.

Pretraining for programming languages is a new area, with a lot of recent research attempting to apply NLP pre-training approaches to source code. The two pioneer models are CuBERT [11] and CodeBERT [7]. To train NL-PL cross-modal representation, CuBERT uses BERT's sophisticated masked language modelling goal, while CodeBERT adds a replacement token detection [4] job. [22] and [16] use GPT and UniLM [6] for the code completion job, respectively, in addition to BERT-style models. Transcoder [19] investigates unsupervised programming language translation. Apart from this, PLBART [1], which is based on the BART encoder-decoder concept, can also help with comprehension and generating jobs. All of the previous work, on the other hand, simply analyses code in the same manner that natural language does, essentially ignoring code-specific properties. Instead, we propose that for pre-training, we use the identification information in code.

[19] offer a deobfuscation aim to harness the structural component of PL, whereas GraphCodeBERT [9] merges the data flow abstracted from the code structure into CodeBERT. These models are solely focused on improving a code-specific encoder. The relative distances between code tokens should be captured across the code structure, according to [31].

To leverage the explicit structural information in programs, these approaches often use abstract syntax tree (AST) as the input of their models [27, 28, 30]. A typical example of these approaches is CDLH [27], which encode code fragments by directly applying Tree-LSTM [23] on binarized ASTs.

3 BACKGROUND

Graph neural networks (GNNs) are able to achieve significant attention in last few years due to their efficiency in graph representation and performance on downstream tasks. Most of the GNNs can be expressed in the form of the message passing network mentioned below [8]:

$$h_v^l = COM^l\left(\left\{h_v^{l-1}, AGG^k\left(\{h_{v'}^{l-1} : v' \in \mathcal{N}_G(v)\}\right)\right\}\right)$$

Here, h_v^l is the representation of node v of graph G in l-th layer of the GNN. The function AGG (Aggregate) considers representation of the neighboring nodes of v from the $(l-1)^{th}$ layer of the GNN and maps them into a single vector representation. As neighbors of a node do not have any ordering in a graph and the number of neighbors can vary for different nodes, AGG function needs to be permutation invariant and should be able to handle different number of nodes as input. Then, COM (Combine) function uses the node representation of v^{th} node from $(l-1)^{th}$ layer of GNN and the aggregated information from the neighbors to obtain an updated representation of the node v.

3.1 Graph Convolutional Network (GCN)

Let $\mathcal{N}_G(i)$ denote a set of neighbors of $i \in V$, W be the weight parameter for layer and $\sigma(.)$ be a non-linear activation function. General GCN message passing rule at layer l for node i then consists of:

$$h_i^l = \sigma\Big(\sum_{j \in \mathcal{N}_G(i)} A_{ij} W h_j^{l-1}\Big) \tag{1}$$

3.2 Graph Attention Network (GAT)

The graph attention network (GAT) [25] incorporates the attention mechanism into the propagation step. It computes the hidden states of each node by attending to its neighbors, following a self-attention strategy. For a graph G, we use a trainable parameter matrix W to transform the initial feature vector x_i as Wx_i. We use a trainable attention vector $a \in \mathbb{R}^{2K}$ to learn the importance α_{ij} for any two neighboring nodes in a graph as follows:

$$\alpha_{ij} = \frac{exp\Big(\text{LeakyReLU}\Big(a \cdot [Wx_i || Wx_j]\Big)\Big)}{\sum_{j' \in \mathcal{N}_G(i)} exp\Big(\text{LeakyReLU}\Big(a \cdot [Wx_i || Wx_{j'}]\Big)\Big)} \tag{2}$$

where $\mathcal{N}_G(i)$ is the neighboring nodes of i (including i itself) in the graph G, $\cdot$ represents dot product between the two vectors and $||$ is the vector concatenation. These normalized importance parameters

are used to update the node features as:

$$h_i = \sigma\Big(\sum_{j \in \mathcal{N}_G(i)} \alpha_{ij} A_{ij} W x_j\Big) \tag{3}$$

where A_{ij} is the $(i, j)$the element of A, i.e., weight of the edge (i, j) in G. W is the weight matrix associated with the linear transformation which is applied to each node, and a is the weight vector of a single-layer MLP.

Moreover, GAT utilizes the multi-head attention to stabilize the learning process. It applies K independent attention head matrices to compute the hidden states and then concatenates their features (or computes the average), resulting in the following two output representations:

$$h_i = \sigma\Big(\frac{1}{K}\sum_{k=1}^{K}\Big(\sum_{j \in \mathcal{N}_G(i)} \alpha_{ij}^k A_{ij} W x_j\Big)\Big) \tag{4}$$

$$h_i = ||_{k=1}^{K}\sigma\Big(\sum_{j \in \mathcal{N}_G(i)} \alpha_{ij}^k A_{ij} W x_j\Big) \tag{5}$$

Here α_{ij}^k is the normalized attention coefficient computed by the k^{th} attention head.

3.3 Hyperbolic Graph Convolutional Network (HGCN)

Hyperbolic geometry is a non-Euclidean geometry with a constant negative curvature, where curvature measures how a geometric object deviates from a flat plane. Here, we work with the hyperboloid model for its simplicity and its numerical stability. First, since input features are often Euclidean, a mapping from Euclidean features to hyperbolic space is generated. Next, two components of graph convolution is derived, the analogs of Euclidean feature transformation and feature aggregation in the hyperboloid model.

HGCN first maps input features to the hyperboloid manifold via the exp map. Let $x^{0,E} \in R^d$ denote input Euclidean features and K is the trainable curvature. The hyperbolic mapping is:

$$x^{0,H} = exp_o^K((0, x^{0,E})) = \Big(\sqrt{K}\cosh(\frac{||x^{0,E}||_2}{\sqrt{K}}, \sqrt{K}\sinh(\frac{||x^{0,E}||_2}{\sqrt{K}}\frac{x^{0,E}}{||x^{0,E}||_2}\Big) \tag{6}$$

Linear transformation requires multiplication of the embedding vector by a weight matrix, followed by bias translation. To compute matrix vector multiplication, we first use the logarithmic map to project hyperbolic points x^H to $T_o H^{d,K}$. Thus the matrix representing the transform is defined on the tangent space, which is Euclidean and isomorphic to R^d. We then project the vector in the tangent space back to the manifold using the exponential map. Let W be a $d'xd$ weight matrix. We define the hyperboloid matrix multiplication as:

$$W \otimes^K x^H = exp_o^K\Big(W log_o^K(x^H)\Big) \tag{7}$$

The hyperbolic addition for the bias is defined as:

$$x^H \oplus^K b = exp_{x^H}^K\Big(P_{o \to x^H}(b)\Big) \tag{8}$$

where $P_{o \to x^H}(.)$ is the parallel transport from tangent space at o which is $T_o H^{d',K}$ to tangent space at x^H which is $T_{x^H} H^{d',K}$.

In GCNs, attention develops a concept of neighbour relevance and collects messages based on their importance to the centre node. The focus on Euclidean embeddings, on the other hand, ignores the hierarchical architecture of many real-world networks. Given hyperbolic embeddings (x_i^H, x_j^H), we first map x_i^H and x_j^H to the tangent space of the origin to compute attention weights w_{ij} with concatenation and Euclidean Multi-layer Percerptron (MLP). Figure 2 shows the neighbourhood aggregation process. The hyperbolic aggregation is defined as:

$$w_{ij} = SOFTMAX_{j \in \mathcal{N}(i)}(MLP(log_o^K(x_i^H)||log_o^K(x_j^H))) \tag{9}$$

$$AGG^K(X^H)_i = exp_{x_i^H}^H\Big(\sum_{j \in \mathcal{N}(i)} w_{ij} log_{x_i^H}^K(x_i^H)\Big) \tag{10}$$

Analogous to Euclidean aggregation, HGCN uses a non-linear activation function, $\sigma(.)$ such that $\sigma(0) = 0$, to learn non-linear transformations. Hyperbolic non-linear activation between layer l and $l-1$ with curvatures K_l and K_{l-1} respectively is defined as:

$$\sigma^{\otimes^{K_{l-1},K_l}}(x^H) = exp_o^{K_l}(\sigma(log_o^{K_{l-1}}(x^H))) \tag{11}$$

Thus, combining all of the equations defined above creates the achitecture of HGCN. HGCN then stacks multiple hyperbolic graph convolution layers. At each layer HGCN transforms and aggregates neighbour's embeddings in the tangent space of the center node and projects the result to a hyperbolic space with different curvature.

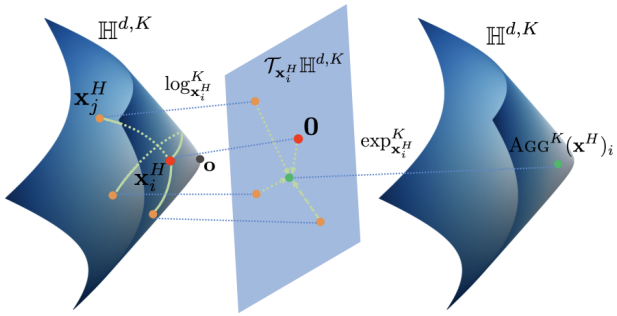

Figure 2: HGCN neighborhood aggregation first maps messages/embeddings to the tangent space, performs the aggregation in the tangent space, and then maps back to the hyperbolic space [29].

4 HYPERCODE

This section describes in detail the model designed along with figures.

4.1 Graph Formation

Given a piece of code of a programming language it can be parsed into an abstract syntax tree using tree-sitter library[1]. The result from the tree-sitter is a tree with each of the nodes of the tree having node types, starting and ending point as node attributes. As a result, the numbers of ASTs generated is equal to the size of the dataset.

[1]https://tree-sitter.github.io/tree-sitter/

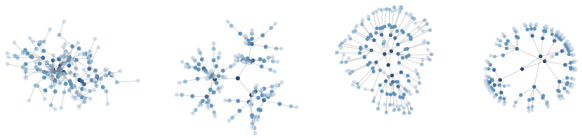

Figure 3: The first two figures are GCN embeddings in first and last layers which hardly capture hierarchy (depth indicated by color). In contrast, the last two figures show that HGCN preserves node hierarchies [29].

The AST is pre-processed to an edge list for each code snippet using depth first search. The root node is numbered as node 0 and then depth first search is performed on the root node and its children are numbered in an incremental order and so on. The result is a dictionary where keys are the node numbers and the values are the nodes which are connected by edges to the key node. Even though node types are not unique, the levels in which they are present is unique. Hence, a DFS-based approach will be able to capture the hierarchy in the AST.

The AST generated from tree-sitter has node types associated with each node. A dictionary is created with unique node types from all the ASTs for one PL. To create the feature matrix, a one hot encoding of the global unique node types is created which is converted to feature matrix.

4.2 Code Matching

The main model used is CodeT5 (Figure 5) which is a encoder-decoder model which aims to derive generic representations for programming language (PL) and natural language (NL) via pre-training on unlabeled source code. The initial task in consideration is that of code matching where, code snippets are to be matched to their code description. The code tokens and code description tokens are encoded using CodeT5 pre-trained model. The adjacency matrix and feature matrix created from the AST is used to get graph embeddings from one of the graph models like HGCN.

The embeddings from the CodeT5 model and the HGCN model are aggregated to calculate scores for the downstream task. The aggregated code embeddings and description embeddings are multiplied to get the predictions followed by the application of softmax function. The target values are diagonal matrix with all ones in the diagonal. Cross entropy loss is used to train the entire pipeline. In addition to this, we aim to experiment with different Euclidean graph models like Graph Convolutional Networks [12], Graph Attention Networks [25], etc.

Figure 4 shows the entire architecture of *HyperCode*. Figure 3 shows that hyperbolic methods can capture hierarchy as compared to Euclidean graph embeddings. This shows that ASTs which is a hierarchical form of representation of code can be better represented by hyperbolic embeddings.

5 EXPERIMENTS

5.1 Datasets

The datasets used are from CodeSearchNet[2] [10] and CodeT5[3] [29] which consists of six PLs with both unimodal and bimodal data. In total, there are around 8.35 million instances for pretraining. Table 1 shows some basic statistics. To obtain the identifier labels from code, the tree-sitter library was used to convert the PL into an abstract syntax tree and then extract its node type information. Observe that PLs have different identifier rates, where Go has the least rate of 19% and Ruby has the highest rate of 32%.

Roberta[4] [16] Tokenizer from CodeT5 pretrained model is used as a tokenizer to tokenize the code and the code description. We add additional special tokens ([PAD], [CLS], [SEP], [MASK0], ..., [MASK99]).

Table 1: Dataset statistics. "Identifier" denotes the proportion of identifiers over all code tokens for each PL.

PLs	W/ NL	W/o NL	Identifier
Ruby	49,009	110,551	32.08%
JavaScript	125,166	1,717,933	19.82%
Go	319,132	379,103	19.32%
Python	453,772	657,030	30.02%
Java	457,381	1,070,271	25.76%
PHP	525,357	398,058	23.44%
PHP	525,357	398,058	23.44%
C	1M	-	24.94%
CSharp	228,496	856,375	27.85%
Total	3,158,313	5,189,321	8,347,634

5.2 Algorithms

The algorithms considered in our experiments for encoding the AST are: Euclidean Linear (E-Linear), Multilayer Perceptron (MLP), Graph Convolution (GCN) and their hyperbolic variants (H-Linear, H-MLP, and H-GCN). Additionally, we also test the Graph Attention model (GAT) and the hyperbolic variant H-GAT. All models are available in the package GraphZoo[5] [2].

5.3 Training Settings

In this paper, we want to perform the experiments on the splits given in CodeSearchNet. We will use the grid search cross validation on the validation set to tune the hyperparameters of each mentioned classification methods. The code and description lenghts are truncated to a constant value. The experiments are conducted on a RTX8000 GPU within a limit of 48 GB VRAM. We conduct extensive hyperparameter tuning for all the baseline algorithms and report the best results obtained.

Overall, the batch size considered for training, validation, and testing is 5 due to limitations of the GPU. The learning rate is $1e-3$,

[2]https://huggingface.co/datasets/code_search_net#data-splits
[3]https://huggingface.co/docs/transformers/model_doc/t5#transformers.T5EncoderModel
[4]https://huggingface.co/docs/transformers/model_doc/roberta
[5]https://github.com/AnoushkaVyas/GraphZoo

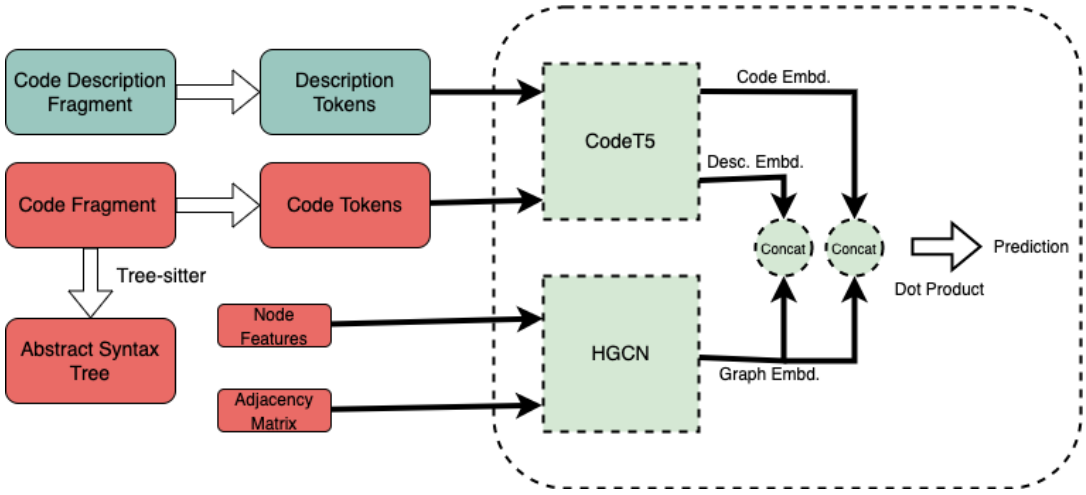

Figure 4: Architecture of *HyperCode*.

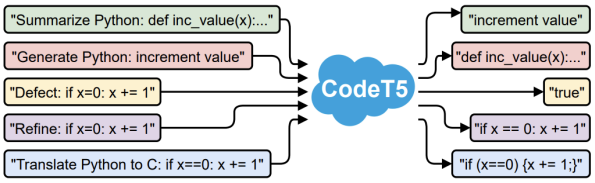

Figure 5: Illustration of our CodeT5 for code-related understanding and generation tasks [29].

and the weight decay is $5e-3$. The training is done for 100 epochs and validation is done for every 5 epochs.

We used four different metrics to analyze the performance of the algorithms during the test period. They are F1 Score, Accuracy, Precision and Recall. All the metrics measure the accuracy of the algorithm. So higher the value of them better is the quality. The expected outcome is improvement in the metrics compared to the baselines which do not use the AST encodings. The improvement is expected because of the underlying hierarchical information the ASTs hold about the code structure which are not captured by the existing encoder-decoder models.

5.4 Results

We run each algorithm 10 times on each dataset and reported the average metrics over all the epochs. Table 2 shows the performance of all the baselines. It can be observed that hyperbolic models do perform better than the Euclidean methods and the simple CodeT5 model. Among the graph neural networks and neural networks, graph neural networks perform better because of their ability to propagate information via message passing algorithm. Graph algorithms can encode both graph structure and node features in a way that is useful in capturing the information of nodes stored in abstract syntax trees.

Hyperbolic methods outperform Euclidean methods for most of the PLs, in most of the metrics because of their ability to encode hierarchy. They offer improved expressiveness for hierarchical graph data.

6 DISCUSSION

The reviews given to the project proposal were addressed in this paper. Reviewer 2 pointed out that," There were several instances of grammatical oddities but this is likely from the author having English as a second language and is therefore not problematic. There was a single typo with a hyphen being placed in the middle of the word hyperparameters." A careful review of the entire paper was conducted to current any grammatical and presentation errors. Reviewer 4 pointed out that, "The author could have thrown more light in the related work section maybe by referring to more literature that could throw more light on the ongoing research and the various directions taken by different authors in the domain.", and "Finally in the expected outcome the author could elaborate more on the use cases of the research." These issues are addressed in the paper in the Background, Related Word, and Discussion sections of the paper. Reviewer 5 mentions that, " the proposal lacks the mention of baseline models.", which has been addressed in the paper.

The impact of this work, would be increase in the performance in existing code tasks. Our work is primarily focused on natural language processing (NLP) applications for software intelligence. Software intelligence research has grown in popularity in both academia and industry over the last decade, with the objective of enhancing software development productivity using machine learning methodologies. Software code intelligence approaches can aid developers in reducing time-consuming repetitive tasks, improving programming quality, and increasing overall software development efficiency. This would significantly cut their working time as well as the computation and operating costs, as a flaw might harm system performance or even cause the system to crash.

The results obtained from the experimentation do not show a major improvement and some possible reasons for it are, although AST can reflect the rich structural information for program syntax, it does not contain some semantic information such as control flow and data flow. Data and control flow graphs can be used to improve the performance of the current work. This could be part of the future work.

	Python				Java				PHP				Javascript			
AST Encoding	F1	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall
CodeT5 (no encoding)	91.12 ± 1.23	90.23 ± 0.76	90.45 ± 0.87	91.13 ± 0.65	90.32 ± 0.23	91.22 ± 0.65	90.43 ± 0.54	90.22 ± 0.65	91.12 ± 0.43	91.54 ± 0.33	91.22 ± 0.55	91.44 ± 0.45	90.22 ± 0.44	90.22 ± 0.45	90.55 ± 0.45	90.12 ± 0.22
E-Linear	92.12 ± 1.43	91.13 ± 0.56	91.53 ± 0.83	91.23 ± 0.45	91.34 ± 0.21	92.23 ± 0.64	92.23 ± 0.44	91.12 ± 0.43	92.14 ± 0.33	92.44 ± 0.23	92.22 ± 0.56	92.45 ± 0.35	93.33 ± 0.45	94.22 ± 0.55	92.55 ± 0.46	92.12 ± 0.42
H-Linear	93.11 ± 0.23	92.23 ± 0.56	92.45 ± 0.83	92.14 ± 0.85	92.32 ± 0.27	92.22 ± 1.65	**93.43 ± 0.44**	92.22 ± 0.55	92.12 ± 0.33	**93.54 ± 0.33**	92.22 ± 0.55	93.44 ± 0.45	92.22 ± 0.44	92.22 ± 0.45	92.55 ± 0.45	92.12 ± 0.22
MLP	93.12 ± 1.23	92.23 ± 0.76	93.45 ± 0.87	91.17 ± 0.55	92.32 ± 1.23	93.22 ± 1.65	91.73 ± 0.44	91.22 ± 0.35	91.22 ± 0.23	92.54 ± 0.43	**93.22 ± 0.25**	92.54 ± 0.25	92.22 ± 0.44	93.22 ± 0.45	92.55 ± 0.45	92.12 ± 0.22
H-MLP	93.42 ± 0.25	92.24 ± 0.75	93.54 ± 0.78	92.71 ± 0.55	92.23 ± 1.32	93.24 ± 1.56	92.37 ± 0.44	92.22 ± 0.53	92.22 ± 0.32	92.45 ± 0.34	**93.22 ± 0.52**	92.45 ± 0.52	92.22 ± 0.44	**93.43 ± 0.54**	92.34 ± 0.54	92.82 ± 0.22
GCN	93.18 ± 1.23	92.23 ± 0.56	92.43 ± 0.85	92.15 ± 0.55	92.35 ± 0.25	**93.25 ± 0.65**	92.53 ± 0.64	92.24 ± 0.65	**93.42 ± 0.53**	**93.54 ± 0.33**	92.22 ± 0.55	92.44 ± 0.45	92.22 ± 0.24	93.22 ± 0.45	**93.55 ± 0.45**	92.15 ± 0.22
H-GCN	**93.78 ± 1.43**	93.23 ± 0.56	93.45 ± 0.85	**93.83 ± 0.65**	**93.32 ± 0.23**	93.22 ± 0.65	**93.43 ± 0.54**	**93.22 ± 0.65**	93.12 ± 0.43	**93.54 ± 0.33**	**93.22 ± 0.55**	93.44 ± 0.45	**93.22 ± 0.43**	93.22 ± 0.45	**93.55 ± 0.45**	**93.12 ± 0.22**
GAT	93.12 ± 1.25	92.25 ± 1.76	92.45 ± 0.85	92.13 ± 0.65	91.32 ± 0.23	92.22 ± 0.65	92.43 ± 0.54	92.22 ± 0.65	92.12 ± 0.43	92.54 ± 0.33	92.22 ± 0.55	92.44 ± 0.45	92.22 ± 0.54	92.22 ± 0.45	92.55 ± 0.45	92.12 ± 0.22
H-GAT	93.42 ± 0.25	**93.24 ± 0.75**	**93.58 ± 0.78**	93.71 ± 0.55	93.23 ± 1.32	93.24 ± 1.56	93.37 ± 0.44	**93.22 ± 0.53**	93.22 ± 0.32	93.45 ± 0.34	**93.22 ± 0.52**	**93.45 ± 0.52**	**93.22 ± 1.14**	**93.43 ± 0.54**	93.34 ± 0.54	**93.12 ± 0.22**

Table 2: F1, Accuracy, Precision, and Recall scores for code matching task for four PLs with their corresponding standard deviations over 10 random parameter initializations. Best results are given in bold.

7 CONCLUSION

We have introduced a model which uses hyperbolic graph neural networks to encode abstract syntax trees for the task of code matching. The paper also reports results with other state-of-the art Euclidean graph neural networks and other Euclidean neural networks. The results indicate that hyperbolic methods help in improving the performance for the task. We also indicate the limitations and the need for future work which would be to improve the representation of ASTs or using data and control flow graphs.

REFERENCES

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211

[2] Mehrdad Khatir Chandan K. Reddy Anoushka Vyas, Nurendra Choudhary. 2022. GraphZoo: A Development Toolkit for Graph Neural Networks with Hyperbolic Geometries. In *Companion Proceedings of the Web Conference 2022* (Lyon, France) *(WWW '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3487553.3524241

[3] Ines Chami, Zhitao Ying, Christopher Ré, and Jure Leskovec. 2019. Hyperbolic graph convolutional neural networks. In *Advances in Neural Information Processing Systems*. 4869–4880.

[4] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555* (2020).

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[6] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. 2019. Unified language model pre-training for natural language understanding and generation. *Advances in Neural Information Processing Systems* 32 (2019).

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[8] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for Quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. 1263–1272.

[9] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCode{BERT}: Pre-training Code Representations with Data Flow. In *International Conference on Learning Representations*. https://openreview.net/forum?id=jLoC4ez43PZ

[10] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[11] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 5110–5121. https://proceedings.mlr.press/v119/kanade20a.html

[12] Thomas N Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907* (2016).

[13] Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. 2010. Hyperbolic geometry of complex networks. *Phys. Rev. E* 82 (Sep 2010), 036106. Issue 3. https://doi.org/10.1103/PhysRevE.82.036106

[14] Dmitri Krioukov, Fragkiskos Papadopoulos, Amin Vahdat, and Marián Boguñá. 2009. Curvature and temperature of complex networks. *Phys. Rev. E* 80 (Sep 2009), 035101. Issue 3. https://doi.org/10.1103/PhysRevE.80.035101

[15] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 7871–7880. https://doi.org/10.18653/v1/2020.acl-main.703

[16] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[17] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[18] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[19] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems* 33 (2020), 20601–20611.

[20] Frederic Sala, Chris De Sa, Albert Gu, and Christopher Re. 2018. Representation Tradeoffs for Hyperbolic Embeddings. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 4460–4469. https://proceedings.mlr.press/v80/sala18a.html

[21] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2019. MASS: Masked Sequence to Sequence Pre-training for Language Generation. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 5926–5936. https://proceedings.mlr.press/v97/song19d.html

[22] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. *IntelliCode Compose: Code Generation Using Transformer*. Association for Computing Machinery, New York, NY, USA, 1433–1443. https://doi.org/10.1145/3368089.3417058

[23] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[25] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. *International Conference on Learning Representations* (2018). https://openreview.net/forum?id=rJXMpikCZ

[26] Ping Wang, Khushbu Agarwal, Colby Ham, Sutanay Choudhury, and Chandan K Reddy. 2021. Self-supervised learning of contextual embeddings for link prediction in heterogeneous networks. In *Proceedings of the Web Conference 2021*. 2946–2957.

[27] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.. In *IJCAI*. 3034–3040.

[28] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st*

IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 87–98.

[29] Shafiq Joty Steven C.H. Hoi Yue Wang, Weishi Wang. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*.

[30] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[31] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-Agnostic Representation Learning of Source Code from Structure and Context. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Xh5eMZVONGF