Boosting Performance of Code Completion Algorithms with Abstract Syntax Trees and Convolutional Graph Autoencoders

Sam Donald
samdonald@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Abstract

Typical natural language processing (NLP) sequence-sequence models solely utilize input text, from which relationships between words are modeled and used to generate output. When these models are applied to code prediction, additional information dense representations in the form of abstract syntax trees (ASTs) can be generated for free by compiling the source code. As ASTs contain the codes syntactic structure, it is assumed that a models predicative performance can be boosted though its inclusion. This work proposes a method in which this graph based AST knowledge can be instilled into sequence-sequence prediction models, though the use of a convolutional graph autoencoder. The success of this method is demonstrated by seeding an long short-term memory (LSTM) based sequence-sequence model with encoded AST information, and applying this model to the task of next token prediction for python code.

CCS Concepts: • **Computing methodologies** → **Information extraction**; **Machine translation**; **Neural networks**.

Keywords: Convolutional graph autoencoder, abstract syntax tree, code completion, encoder decoder

1 Introduction

Applying neural networks to code completion tasks is a new application within the deep learning community, and is one that has recently gained increased attention in part due to the release of GPT-3 and GitHub Copilot [2]. These tools, specifically GitHub Copilot, can be utilized by developers to write code more efficiently through completion of individual tokens, lines, and entire functions.

These code completion techniques can be placed into two broad categories, code-to-code in which code is predicted based on the code preceding it, and text-to-code in which comments or natural language instructions are instead used. Recently a subclass of code-to-code prediction models has emerged utilizing Abstract Syntax Trees (ASTs), an information rich graph used to capture the codes syntactic structure containing all constructs and their subsequent rules. While these are commonly used by compilers, their syntactic representation have been successfully utilized in code matching and classification tasks [4][8][9][10].

This project aims to incorporate the syntactic knowledge of ASTs into existing code prediction models by encoding the relationships within an AST with a Convolutional Graph Autoencoder (GAE). As ASTs capture the inherent structure of code, it is assumed that the models overall understanding and predictive power can be improved through the inclusion of this knowledge. As training state of the art transformer models from scratch requires significant computational resources [10], a focus is instead placed on a simplified LSTM based encoder decoder model for code prediction, with additional encoded AST information.

2 Related Work

CodeXGLUE was recently developed for a variety of machine translation tasks. With respect to code completion, a CodeGPT Transformer model was trained on the PY150 dataset to generate both token and full lines of output code, achieving token level accuracy of 75.11%. While this code completion method relied solely on code inputs, adjacent methods, focused on code repair and code translation, incorporated various forms of information derived from ASTs [9].

CodeGraph expanded upon CodeGPT through the inclusion of AST and comment data [10]. No noticeable improvements were made when the CodeXGLUE CodeGPT-Small-Java model was fine-tuned with the additional AST data. When trained from scratch the token level BLEU score increased to 88.68%. It is of note the AST information utilized was heavily prepossessed, with the AST graph being used to generate a simplified data flow chart denoting variables and their types which was then provided to the GPT based transformer model.

GraphCodeBERT utilized a similar methodology by including a data preprocessing step to distill AST graphs into a data flow representation [4]. This was shown to further boost performance within code translation and refinement tasks. AST BERT uses a similar technique to GraphCodeBERT, by refining AST data via pruning the graph into subtrees [8].

ASTNN explicitly translates AST data into a vector representation to be utilized for effective representation learning in downstream tasks [13]. This is accomplished through a Bidirectional Gated Recurrent Unit (Bi-GRU), in addition to a AST traversal algorithm designed to prune the tree into a sequence of statement trees.

Graph Convolutional Networks (GCNs), in the form of a Graph Autoencoder (GAE) have been demonstrated to learn meaningful latent embeddings on link prediction tasks [1][3], while their probabilistic variant, variational graph autoencoders (VGAEs) have been successfully applied to tasks such as molecule design [7][12]. To the authors understanding, GAEs have not been applied to ASTs, and as such the approach outlined by [12] will be investigated within this project.

The remainder of this paper is organized as follows. Section 3 outlines the dataset of interest and prepossessing steps. The model architecture for both the Graph Autoencoder and downstream sequence-sequence token predictor are described in Section 4. Experimental results are presented within Section 5, with Sections 6 and 7 discussing results and concluding the paper.

3 Data Preparation

3.1 Dataset

The python subset of the CodeSearchNet dataset is used within this project, containing 457,461 unique functions and associated comments divided in accordance with an approximate 90-5-5 train-validate-test ratio [5].

Of the entire CodeSearchNet training and validation subsets, 5% is allocated for training, validating and testing the graph autoencoder (GAE), while 15% is allocated for the downstream token prediction model. The resulting dataset splits for each model are described in Table 1. The entirety of the CodeSearchNet dataset was not used due to its excessively large size intended for training of complex Transformer based models.

Table 1. Datset splits

Model	Train	Validation	Test
GAE	16,469	2,058	2,058
Token Prediction	49,425	6,174	6,175

For each function within the CodeSearchNet dataset, the following information is provided:

- Repository name
- Function path in repository
- Function name
- Entire function string
- Language
- Function code string
- Function code tokens
- Function documentation string
- Function documentation tokens
- Function code url

Of this data, the function code string and code tokens are used for AST generation and inputs to the token prediction model respectively. This is done such that redundant natural language information such as the function documentation string is not included.

3.2 Abstract Syntax Trees

The proposed Graph Autoencoder (GAE) requires input data in the form of a pytorch geometric data object. This data object in turn requires a tensor of edge indices and node data as inputs.

To generate these tensors, CodeSearchNet function code strings are parsed to generate an AST using Pythons inbuilt AST library. The resultant graph is then walked to extract node names and link identifiers such that a human readable AST graph can be formed. Comments are removed during this step to prevent excessively large nodes containing documentation. A rendered AST graph generated from this process and the associated python function used to generate it are displayed within Figure 2 and 1 respectively. Once the human readable graph is assembled, node names are tokenized, with most frequent 9,999 node identifiers allocated a unique value and the remaining mapped to a shared value of 10,000. These values are then one-hot encoded and used as the nodes feature values.

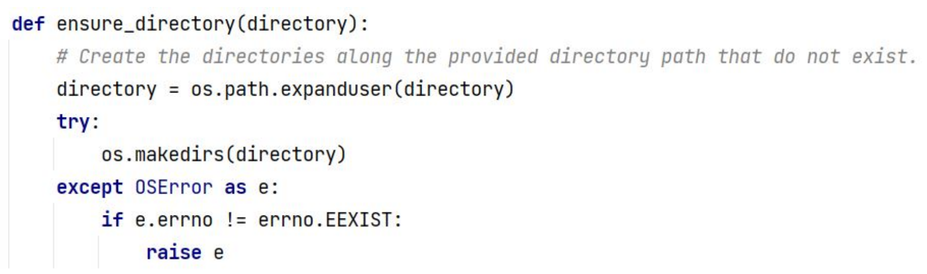
For an single AST graph with N nodes, the output of this process is a tensor containing encoded nodes of size $N \times 10000$, and a tensor of edge indices describing connections between nodes. Edge indices are represented by 2 dimensional tensor describing the source and target node id. As an AST is a tree representation, a child node is only ever connected a singular parent though a unidirectional edge. As such, the resulting edge index tensor is of size $2 \times (N - 1)$. While not strictly representing the AST, a bidirectional representation is also possible and results in an edge index tensor of size $2 \times 2(N - 1)$.

3.3 Tokenization

Microsoft's pretrained tokenizer from CodeGPT-small-py is applied to the CodeSearchNet function tokens, mapping each token to a one hot encoded vector of size 50,001 [9]. Functions containing more than 768 tokens are discarded. A value of 768 was chosen due to CodeGPT being a potential downstream encoder-decoder model, with its upper limit for input tokens being 768. As CodeGPT was not investigated, this restriction is unnecessary, yet constrains the complexity of inputs provided to both the GAE and token prediction model. This threshold value of 768 tokens results in approximately 0.83% of the CodeSearchNet functions being discarded.

3.4 Dataflow Pipeline

The AST graph generation and tokenization procedures are summarized by the data-flow pipeline displayed within Figure 3. Due to their large size, the pipeline is designed to limit



```python
def ensure_directory(directory):
    # Create the directories along the provided directory path that do not exist.
    directory = os.path.expanduser(directory)
    try:
        os.makedirs(directory)
    except OSError as e:
        if e.errno != errno.EEXIST:
            raise e
```

Figure 1. Input function

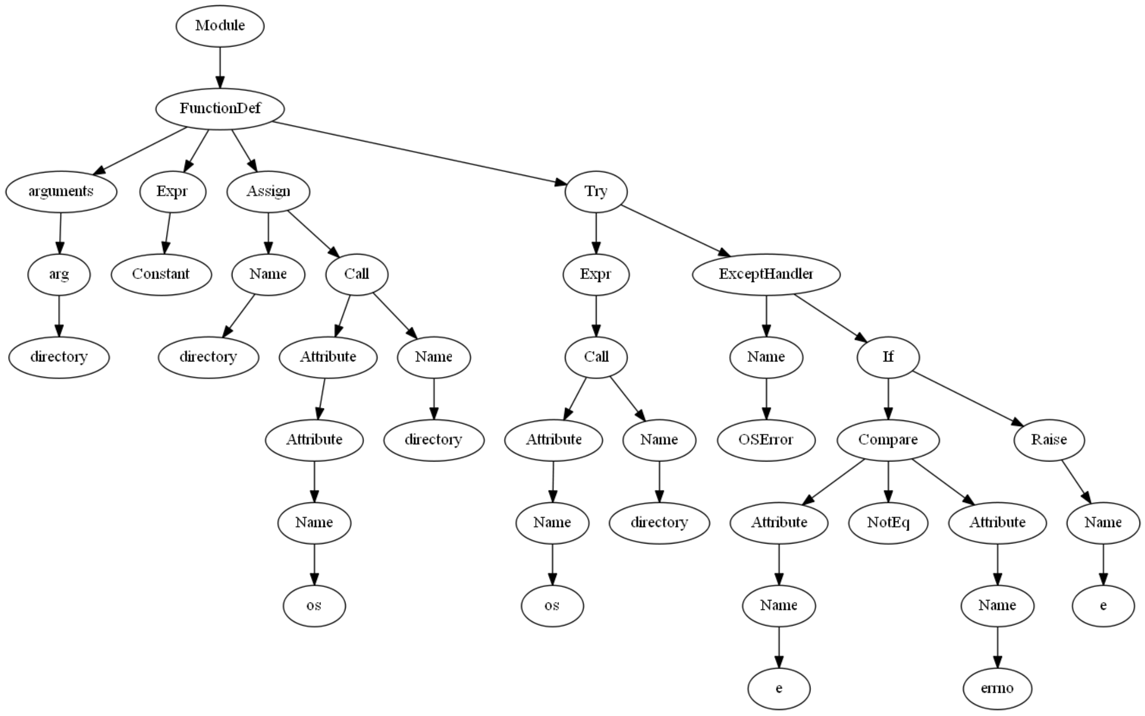


Figure 2. Output AST Graph

the amount of data saved in the form of a AST graphs. Once the GAE is trained it is then used to directly encode the AST graphs, with the encoded representation being saved and fed to the downstream token prediction model.

4 Model Descriptions

4.1 Graph Autoencoder

The proposed Graph Autoencoder comprises of an encoder and decoder module, with the encoder itself comprising of two Graph Convolutional layers. The AST graph containing node and edges is passed into the encoder to produce the latent variable Z, with the decoder then reconstructing the input graph. Of note the nodes of the input graph are not reconstructed within this configuration, only their edges. The architecture of this Graph Autoencoder is described by the blue blocks within Figure 4. An input AST graph is represented by A and X, where A is the $n \times n$ node edge matrix representing connections between each of the n AST nodes, and X is the diagonal feature matrix representing the one hot encoded node values. The diagonal elements within node edge matrix A are set to one.

4.1.1 Graph Convolutional Layers. The following section outlines the Graph Convolutional layers used within the GAE encoder, along with its associated decoder [12][7].

An $n \times n$ degree matrix D can be constructed based on the node index matrix A in accordance with equation 1, where $1 \leq i, j \leq n$.

$$D_{ij} = \begin{cases} \sum_{k=1}^{n} A_{ik} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{1}$$

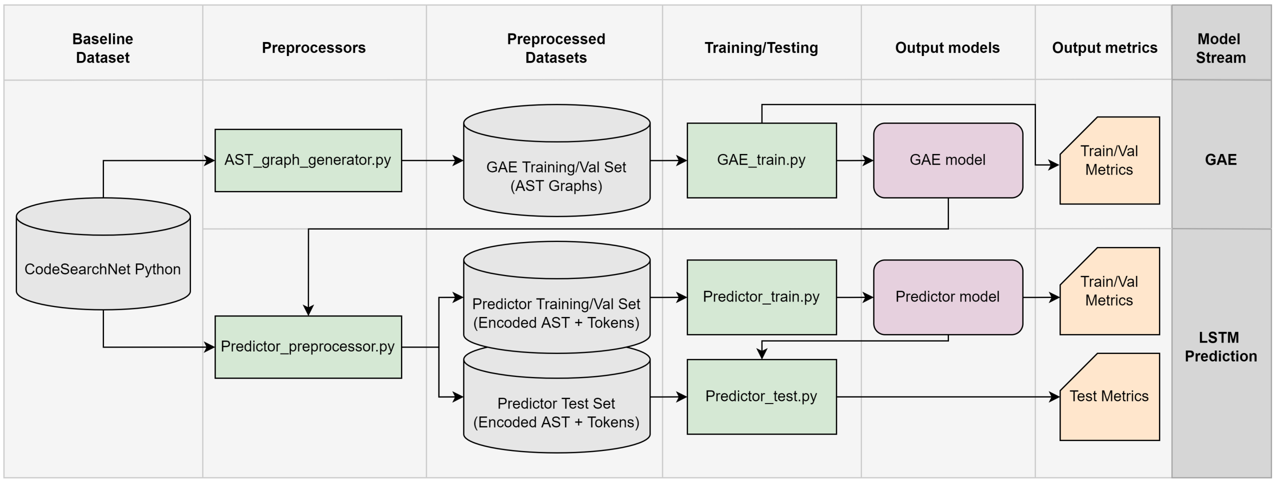


Figure 3. Data processing pipelines

The output of the l^{th} graph convolutional layer $Z^{(l)}$ is represented by equation 2, where $1 \leq l \leq k$, the input to the first layer is the feature matrix X such that $Z^0 = X$, and weight matrix $D^{-1/2}$ for a given layer l defined by equation 3

$$Z^{(l)} = f^{(l)}(D^{-1/2}AD^{-1/2}Z^{(l-1)}W^{(l)}) \tag{2}$$

$$D_{ij}^{-1/2} = \begin{cases} D_{ii}^{-1/2} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{3}$$

Where $W \in \mathbb{R}^{CxF}$ are the learnable parameters, with C denoting the output channels and F the corresponding feature maps. $f^{(l)}$ within equation 2 represents the activation functions for layer l. For the proposed two layer configuration the following activation functions are implemented:

$$f^{(l=1)}(t) = ReLU(t) = max(0, t) \tag{4}$$

$$f^{(l=2)}(t) = t \tag{5}$$

The decoder then reconstructs the adjacency matrix A', with cross entropy loss used as a loss function between A an A'.

$$A' = sigmoid(ZZ^T) \tag{6}$$

4.2 Code Prediction Encoder-Decoder

4.2.1 Baseline Model. The downstream token prediction model implemented is based on the long short-term memory (LSTM) architecture for code completion described within Microsoft's CodeXGLUE paper [9]. The encoder portion consists of a embedding layer transforming one hot encoded input tokens into a vector of size 768. This embedding is then fed to an LSTM, with the resultant hidden states decoded via a two layer FNN to produce output token probabilities with a softmax layer. For a given input containing n tokens, the input and output dimension will be $n \times 50000$. This baseline encoder-decoder model is described by the orange blocks within Figure 4.

The encoder and decoder weights within this configuration are randomly initialized between −0.1 and 0.1. While this allows for the model to dynamically learn word embeddings during training, it significantly increases the problems' complexity. An alternative would involve pre-training an encoding layer, or utilize existing weights from a similar python token model. Additionally, these weights can be be tied together, which has been demonstrated to improve performance in encoder-decoder models [6][11]. Cross entropy loss is additionally used as a loss function.

4.2.2 Instilling Graph Knowledge. The encoder output for a given AST graph with n nodes will be of size $C \times n$ where C is the number of channels within the GAE, and is transformed to a constant size for ease of incorporation into the downstream token prediction model. This is accomplished though a LSTM linking module used to process the encoded AST graph representation, with the final hidden variable H_n from the LSTM link module initializing the LSTM within the code prediction encoder-decoder model. This linking module is described by the yellow blocks within Figure 4 and is trained in conjunction with the token prediction model.

5 Evaluation and Results

Training was completed on multiple RTX6000 GPUs via Lambda Cloud servers, while data preprocessing of the AST graphs was done locally on an Intel Core i7-1065G7 CPU.

5.1 Graph Autoencoder

The various GAE model architectures and associated hyperparameters tested are outlined within Table 2. Of these, a

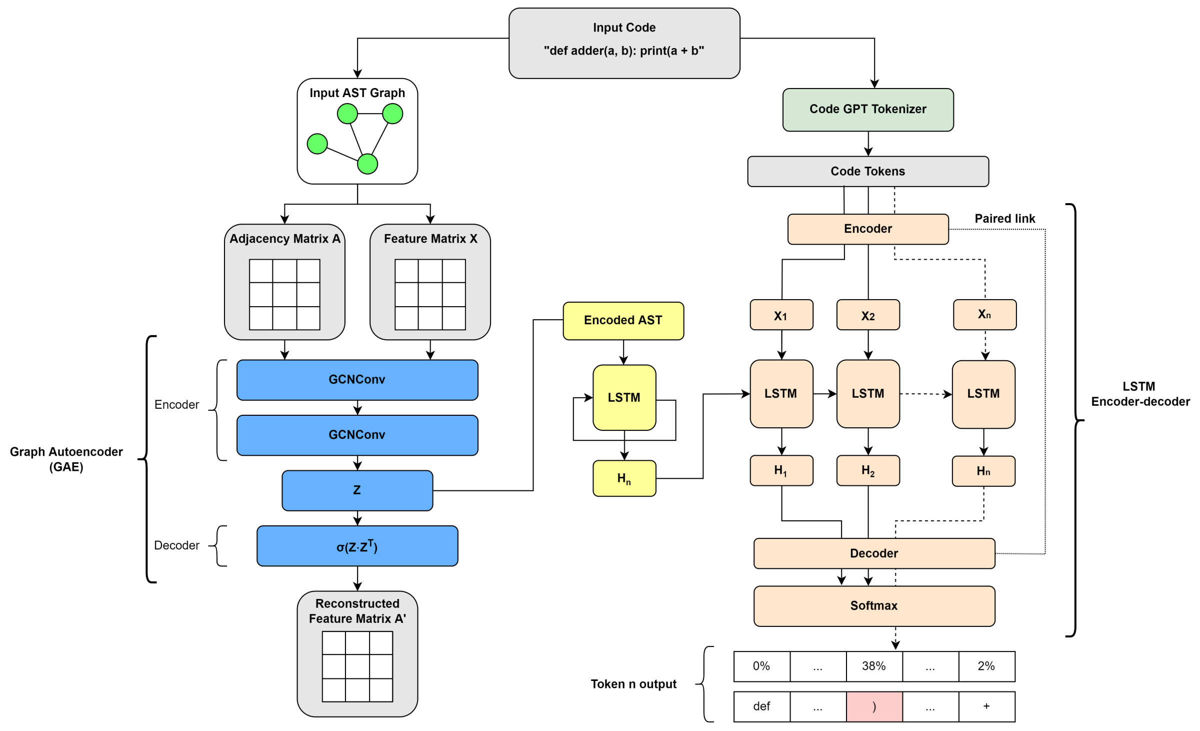


Figure 4. Overall model architecture

two layer encoder with eight GAE channels slightly outperformed all others with respect to minimum validation loss and with test accuracy. This encoder configuration was then further tested using a bidirectional AST edge representation along with a lower learning rate of 1e-4, neither of which resulted in improved test accuracy.

As there is minimal difference between these configurations, the aforementioned model using a two layer encoder, eight GAE channels, a learning rate of 1e-3 and unidirectional edge encoding was selected. Early stopping was used to stop training of the model at approximately epoch 9. The model was then saved and used to encode the AST graphs for the downstream token prediction model.

Loss plots during training for the selected model and its bidirectional equivalent are displayed within Figure 6 and 7 respectively, along with trends in various evaluation metrics for the selected unidirectional configuration within Figure 5.

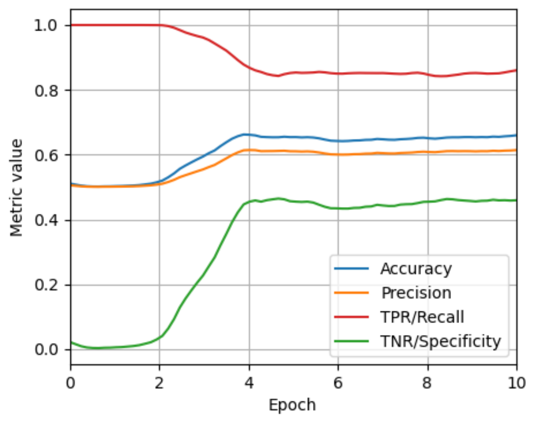


Figure 5. Metric trends of GAE test number 2

5.2 Code Prediction Encoder-Decoder

Descriptions of hyperparamaters used within the token prediction model are outlined within Table 3. As a comparative study, the model was trained and evaluated with and without the encoded AST data, along with tied and untied encoder-decoder weights. Inclusion of the encoded AST data boosted test accuracy by 1.13% when weights were not tied, and by 0.75% when they were. Tying the weights together boosted test accuracy regardless of the inclusion of AST data. This encoder configuration was further tested using bidirectional edges of the AST graph, along with a lower learning rate of 1e-4, neither of which resulted in improvements with respect to test accuracy. A loss plot of training and validation for the best performing model configuration utilizing the encoded AST data and tied encoder-decoder weights is displayed within Figure 8. Validation losses for all four experiments are compared within 9, with their associated test accuracy metrics listed within Table 4.

Table 2. GAE configurations and results

Test number	Encoder channels	Encoder layers	Edge encoding	Learning rate (α)	Validation loss (min)	Test accuracy
1	4	2	Unidirectional	1e-4	1.253	0.621
2	8	2	Unidirectional	1e-4	1.263	**0.657**
3	8	3	Unidirectional	1e-4	1.298	0.612
4	8	2	Unidirectional	1e-3	1.317	0.601
5	8	2	Bidirectional	1e-4	1.278	0.644
6	16	2	Unidirectional	1e-4	1.265	0.641

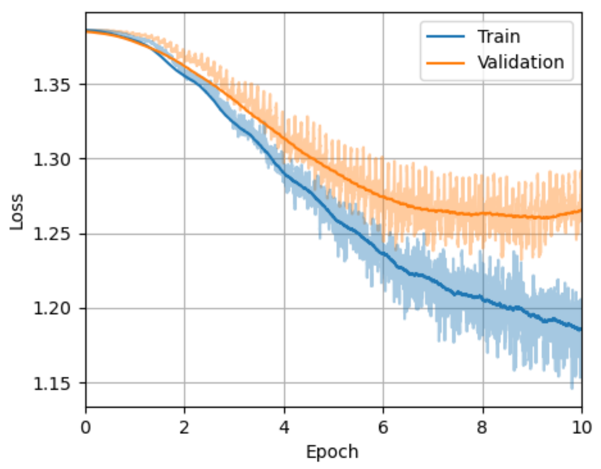


Figure 6. Loss plot of GAE test 2 (Unidirectional)

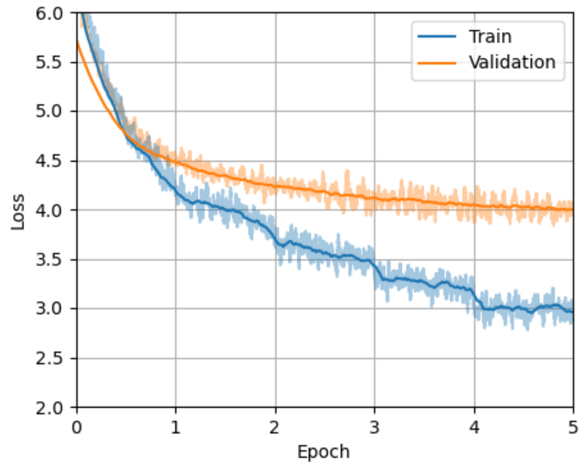


Figure 8. Loss plot of code prediction model, using encoded AST and tied weights

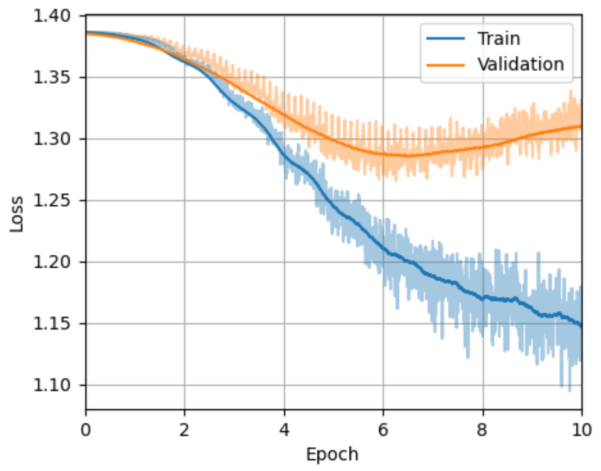


Figure 7. Loss plot of GAE test 5 (Bidirectional)

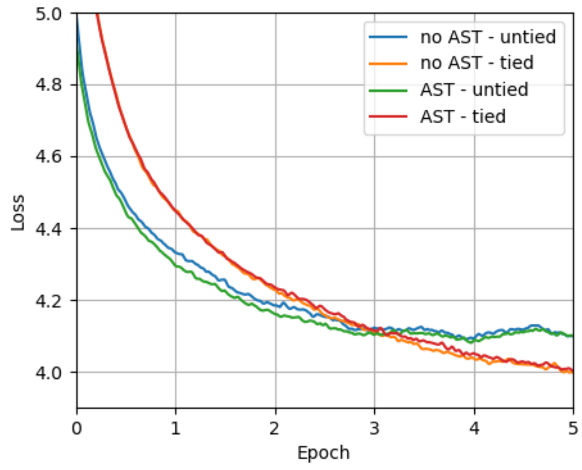


Figure 9. Validation loss of all code prediction configurations tested

Table 3. Token predictor hyperparamaters

Hyperparamater	Value
Encoder width	768
Decoder width	768
LSTM hidden size	768
LSTM layers	1
GAE channels	8
GAE layers	2
Dropout (p)	0.5
Optimizer	AdamW
Learning rate (α)	1e-3
Batch size	32
Validation rate (per iterations)	500

Table 4. Token predictor accuracy metrics

Encoded AST	Tied weights	Test accuracy
No	No	33.62
Yes	No	34.75
No	Yes	34.87
Yes	Yes	**35.62**

6 Discussion

6.1 Graph Autoencoder

While the GAE is clearly learning, as displayed by the decreasing validation loss in Figure 6, it was only able to increase accuracy on the validation set by a 13.4%. It can therefore be concluded that only a small portion of the underlying AST graph structure is successfully encoded within the latent variable Z passed to the code prediction module.

The GAE's difficultly in learning the adjacency matrix A is likely due to its sparsity. The proposed GAE framework was specifically designed for graphs, where each node can have an arbitrary number of edges allowing for cycles to form. This is not the case for a tree, such as an AST, which contains nodes with only a singular parent and therefore no cycles and increased sparsity. This could be addressed though additional pruning procedures of the generated AST to minimize excessively deep trees, and the removal of redundant nodes such as the Name node to reduce the overall matrix size and sparsity. These pruning procedures are noted as being applied within the majority of similar deep learning models utilizing ASTs [9][10][4][8].

It was expected that the bidirectional encoding of the AST graph would improve performance as the A matrix sparsity was approximately halved, yet the opposite was observed with a 1.13% decrease in test accuracy noted within Table 2. This decrease however was coupled with validation loss diverging from training loss in approximately half the number of epochs when compared to a unidirection edge encoding, as displayed in Figures 6 and 7. Reducing the learning rate for the bidirectional encoding configuration is therefore expected to increase performance and potentially result in a model that can outperform that generated through unidirectional encoding.

Finally, the method by which the nodes of the feature matrix X are encoded could be significantly improved upon, as the naive one-hot encoding method implemented provides no information regarding node to node similarity and significantly increases the problem complexity.

6.2 Code prediction

The performance of token level code prediction was boosted by 0.75% through the inclusion of encoded AST information. This improvement ,however, does not confirm that the AST information is being utilized as intended. Critically, the encoded AST is based off the entire function of interest, while the token prediction portion only has restricted access to preceding tokens. Because of this discrepancy, the code predictor has a convoluted path to observe future tokens via the encoded AST. This could be resolved through restricting the AST to previous code, yet would require significant amounts of preprocessing as ASTs can only be generated from compilable code. Additionally, the modified AST may deviate significantly from underlying function and no longer accurately represent its structure. This issue highlights potential domains in which encoded AST information can be more easily applied, specifically code classification or summarization problems where the entire function is accessible, and could be accomplished with minimal modifications to the proposed architecture.

Tying the weights of the encoder-decoder layers within the code prediction model resulted in notable improvements in both validation loss and test accuracy, as displayed within Figure 9 and Table 4. This implies that the learning of the encoder-decoder weights are limiting overall performance. Pretraining these weights, or using existing ones from similar models is expected to significantly boost performance and should be explored within future works. The lack of pretrained weights likely accounts the significant discrepancies in accuracy between the LSTM based code prediction model implement within this project, and the LSTM model implemented within Microsofts CodeXGLUE paper which was able to achieve a 22.38% higher test accuracy score on a similar python based dataset [9].

7 Conclusion

This project has proposed a viable architecture to generate and encode the structure of abstract syntax trees for downstream tasks.The graph autoencoder implemented, while able to partially learn the AST structure, resulted in a low accuracy of 0.65%. This is expected to be significantly improved by implementing preprocessing steps to reduce the sparsity within the adjacency matrix of interests. The downstream task of next token code prediction was used to validate its performance, with an improvement of 0.75% observed when including the encoded AST data. This improvement may be due to code prediction model having a convoluted path to see future tokens through the encoded AST, and it therefore cannot be concluded that the encoded AST knowledge is providing this boost to performance as intended. This could be verified through future work by segmenting code into partial abstract syntax trees, but the encoded AST knowledge would be better suited for tasks such as code classification where the full AST graph can be utilized.

References

[1] Hongyun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. 2018. A Comprehensive Survey of Graph Embedding: Problems, Techniques and Applications. *arXiv:1709.07604 [cs]* (Feb. 2018). http://arxiv.org/abs/1709.07604 arXiv: 1709.07604.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *arXiv:2107.03374 [cs]* (July 2021). http://arxiv.org/abs/2107.03374 arXiv: 2107.03374.

[3] David Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P. Adams. 2015. Convolutional Networks on Graphs for Learning Molecular Fingerprints. *arXiv:1509.09292 [cs, stat]* (Nov. 2015). http://arxiv.org/abs/1509.09292 arXiv: 1509.09292.

[4] 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. GraphCodeBERT: Pre-training Code Representations with Data Flow. *arXiv:2009.08366 [cs]* (Sept. 2021). http://arxiv.org/abs/2009.08366 arXiv: 2009.08366.

[5] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv:1909.09436 [cs, stat]* (June 2020). http://arxiv.org/abs/1909.09436 arXiv: 1909.09436.

[6] Hakan Inan, Khashayar Khosravi, and Richard Socher. 2017. Tying Word Vectors and Word Classifiers: A Loss Framework for Language Modeling. *arXiv:1611.01462 [cs, stat]* (March 2017). http://arxiv.org/abs/1611.01462 arXiv: 1611.01462.

[7] Thomas N. Kipf and Max Welling. 2016. Variational Graph Auto-Encoders. *arXiv:1611.07308 [cs, stat]* (Nov. 2016). http://arxiv.org/abs/1611.07308 arXiv: 1611.07308.

[8] Rong Liang, Yujie Lu, Zhen Huang, Tiehua Zhang, and Yuze Liu. 2022. AstBERT: Enabling Language Model for Code Understanding with Abstract Syntax Tree. *arXiv:2201.07984 [cs]* (Jan. 2022). http://arxiv.org/abs/2201.07984 arXiv: 2201.07984.

[9] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv:2102.04664 [cs]* (March 2021). http://arxiv.org/abs/2102.04664 arXiv: 2102.04664.

[10] Incheon Paik and Jun-Wei Wang. 2021. Improving Text-to-Code Generation with Features of Code Graph on GPT-2. *Electronics* 10, 21 (Jan. 2021), 2706. https://doi.org/10.3390/electronics10212706 Number: 21 Publisher: Multidisciplinary Digital Publishing Institute.

[11] Ofir Press and Lior Wolf. 2017. Using the Output Embedding to Improve Language Models. *arXiv:1608.05859 [cs]* (Feb. 2017). http://arxiv.org/abs/1608.05859 arXiv: 1608.05859.

[12] Yingfeng Wang, Biyun Xu, Myungjae Kwak, and Xiaoqin Zeng. 2020. A Simple Training Strategy for Graph Autoencoder. In *Proceedings of the 2020 12th International Conference on Machine Learning and Computing (ICMLC 2020)*. Association for Computing Machinery, New York, NY, USA, 341–345. https://doi.org/10.1145/3383972.3383985

[13] 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)*. 783–794. https://doi.org/10.1109/ICSE.2019.00086 ISSN: 1558-1225.