**Pretraining and Fine-tuning a BERT to Predict the Feature of Functions**

**Problem Statement**

The primary objective of this project is to utilize a Transformer-based model, such as BERT, to train a language model that can accurately predict function signatures. Function signatures in this project refer to the parameters type and number of a function. By training the model on a selected dataset of function signatures, the aim is to enable it to generate or function signatures given partial information[2].

To achieve this, the project will pretrain the language model using a specific assembly code corpus with function signatures extracted. It will mainly use masked language modeling, to capture the inherent patterns and dependencies between function signatures and function assembly code. Subsequently, the project will fine-tune the pretrained model on specific downstream tasks related to function signature prediction.

The project will be evaluated based on the performance of the trained language model in accurately predicting function signatures. The evaluation metrics will include precision, recall, and F1 score, measuring the model's ability to generate correct function signatures given partial input.

**Methodology**

**3.1 Data Collection**

For training purposes, a dataset of public x86-64 binary code was collected and analyzed. The data collection process involved the following steps:

1)Gathering Binary Code Samples: I downloaded the identified binary code samples from the sources[1]. To analyze the collected assembly code, I utilized Ghidra, a software reverse engineering tool. Ghidra offers features for disassembling, decompiling, and analyzing binary code. In this case, I used it to transform the binary code into sets of machine instructions, making it easier for Ghidra to analyze the functions' features.

2)Utilizing Ghidra's Headless Mode: To automate the analysis and feature extraction steps, I employed Ghidra's Headless Mode, which provides a command-line interface for batch processing. I developed a Python script to automate the analysis process. By using this mode, the developed script automatically executed on the collected assembly code samples, eliminating the need for manual intervention. Throughout this process, I successfully extracted all the function names, argument types, and argument numbers.

In my dataset, the structure goes like:  
{[function name] [function argument number] [function argument type] [machine instruction]}

This dataset served as the foundation for the subsequent stages of the project, including pretraining and fine-tuning.

**3.2 Pretraining**

For my analysis, I utilized the BERT model from the transformer architecture. Due to its size, I chose Roberta, which is faster in the pre-training process and better suited for machine instructions. It is more suitable for machine instructions. For tokenizing the input, I utilized the FastRobertaTokenizer as it fits better with Roberta's specifications compared with other tokenizers.I implemented the Masked Language Model (MLM) technique using the RobertaForMaskedLM in the Hugging Face library, which effectively combines both MLM and Roberta capabilities.

For the training dataset, I used LineByLineTextDataset. Since the inputs wat in the single text file and each line represents a single function, this library transformed the text file into a set of input\_ids, attention\_mask and word\_id. This set of tools allowed me to achieve precise and detailed results in my project.

Given the relatively slow training process, I was not using an extensive dataset. Instead, I only used 53 functions files from [\*\*], with the data text file itself totaling 134 MB and containing 60,987 functions.

During the pre-training period, I utilized a basic configuration for Roberta, defined as follows:

config = RobertaConfig(

vocab\_size=52\_000,

max\_position\_embeddings=514,

num\_attention\_heads=12,

num\_hidden\_layers=6,

type\_vocab\_size=1,

)

I utilized the DataCollatorForLanguageModeling to construct a data collator. In this context, I set the MLM probability to 0.15 to ensure a more precise training process.

training\_args = TrainingArguments(

output\_dir="./path//to/model",

num\_train\_epochs=1,

per\_gpu\_train\_batch\_size=64,

save\_steps=10\_00,

save\_total\_limit=2,

prediction\_loss\_only=True,

)

{'train\_runtime': 13636.1682, 'train\_samples\_per\_second': 2.205, 'train\_steps\_per | 0/470 [00:00<?, ?it/s]Y\_second': 0.034, 'train\_loss': 3.7836241377160906, 'epoch': 1.0}

**3.3 Fine-tuning**

The fine-tuning process involved data subset selection, consistent tokenization, specific approaches for the MLM task, dataset formatting, and the calculation of perplexity as an evaluation metric[3].

In my opinion, pre-training and fine-tuning are similar, but fine-tuning focuses more on the content. So for the dataset in the fine-tuning process, I used a random 30% subset from the original training dataset for training, testing, and evaluation. Regarding the tokenizer, I followed the same approach as the pretraining process. I formatted the dataset and created an id2label mapping, and padding and chunk splitting techniques were also applied in the project. To address the MLM task during fine-tuning, I used the MLM DataCollatorForLanguageModeling, in the huggingface library which provided random masking.

One of the challenges was establishing a connection between the features and functions. Given that explicit labels were not available for LinebylineDataset, I finally chose an unsupervised approach. In my project, I decided to calculate the perplexity of the pretrained model by using the model.evaluate() method. This involved computing the cross-entropy loss on the test set and then taking the exponential of the result.

**Results and Analysis**

(not ready yet..the prediction doesn’t make sense)

**Discussion**

The strengths of pretraining is that we don’t need to use the entire BERT model, which may slow down the training process and also may be over fit for this task. Instead I chose Roberta for the task, which can identify machine instruction patterns and improve precision of prediction. Fine-tuning allowed for the adaptation of the pretrained model to a specific task by utilizing unsupervised training of a smaller dataset, which enhanced performance in language modeling.

Establishing a robust connection between features and functions machine instructions was a challenge to me in the project. Also, I didn’t get the desired outcomes when implementing padding and chunk splitting techniques. Moreover, future work should aim to address the issue of explicit labeling in the fine-tuning process, as it remains an unsolved task for me in the project.

**References**

[1]<https://github.com/UnPPCode/uppc>

[2]<https://www.comp.nus.edu.sg/~prateeks/papers/Eklavya.pdf>

[3]https://huggingface.co/learn/nlp-course/chapter7/3?fw=tf