Assignment 4 Report

Name: Hongzhao Tan

MacID: Tanh10

Student #: 400136957

1. Understanding LoRA

In ordinary adaptation of pre-trained model to specific downstream tasks, usually it is done by fine-tuning which updates all the parameters in the pre-trained model. However, as the size of the model (i.e. the number of parameters in the model) gets larger, deployment of all these fine-tuned versions of the pretrained models has become a critical challenge. With the hypothesis which the change in parameters (specifically the weights) during model fine-tuning/training has low "intrinsic rank", the Low-Rank Adaptation (LoRA) approach has been proposed. This approach trains the dense layers in neural network by optimizing the low-rank decomposition matrices (B and A) of the change (i.e. $the\ \Delta W\ term\ in\ W0 + \Delta W, where\ \Delta W = BA$) of the layers, instead of directly updating the pre-trained model's weights $W0$. Assuming the weights that originally needed to be updated is of shape $d \times k$, the shapes of low-rank decomposition matrices B and A would be $d \times r$ and $r \times k$ respectively, where $r \ll \min(\mathrm{d}, \mathrm{k})$. By only updating the matrices B and A and setting the original pre-trained weights frozen (untrainable), the number of trainable parameters in the model will be reduced significantly ($d \times r + r \times k\ \ vs. d \times k$), which will introduce several benefits: (i). The pre-trained module can be used to build many LoRA modules for different tasks by simply switching the matrices A and B, which will greatly reduce the storage requirements and switching overhead. (ii). Make training/fine-tuning much more efficient since we don't need to calculate the gradients for most of our parameters anymore. (iii). The simple linear design of LoRA allows the trainable matrices to merge with the untrainable pre-trained weights when deployed, so that introducing no inference latency comparing to fully fine-tuned model.

Pre-trained language models are suitable for code-related QA because they have already been trained with large and high-quality datasets of natural language text, which makes them be able to effectively understand the meaning of the natural language questions. Hence, theoretically, only the parameters in the part of the pre-trained model that is responsible for generating outputs (e.g., the decoder in a transformer-based encoder decoder model) will need to be fine-tuned. In addition, most the programming languages' syntax are well formatted and many of the keywords in programming languages are spelled the same as some words in natural language English, which would make the fine-tuning easier.

2. Dataset Preparation

The python-codes-25k dataset contains 4 attributes/columns, namely "instruction", "input", "output", and "text". The instruction column contains the natural language instruction from human about what the generated code should do. The input column contains short and introductive part of AI response or empty. The output column contains the Python code that accomplishes the task that need to be done for the instruction. The text column contains strings which are concatenations of all other 3 columns' values. Under a QA setting, intuitively, the values in the instruction column will be the question, while the output column values will be the answers.

Even though some of the keywords and built-in functions' names have the same or similar spellings as some natural language words, there are still many keywords and built-in function names that do not exist in natural languages' vocabularies (e.g., 'def', 'del', 'frozenset'). In addition, most of the natural language words that are spelled the same as the keywords and function names are not able to represent the meanings that the corresponding keywords and function names have in the Python programming language (e.g. the natural language word 'while' does not contain the meaning of the starting keyword for Python while loop). Hence, new tokens need to be added for each of the Python keywords and built-in functions' names. To distinguish these new tokens for Python from the original natural language tokens, the Python keywords and built-in functions' names have been prefixed with "pytkw:" and "pytfunc:" respectively in their corresponding tokens. The new tokens have been added into the vocabulary of the tokenizer of the pretrained transformer-based model that was employed for this assignment. There were also several special characters, specifically the new-line character "\n", "{", and "}", found to be not in the tokenizer's vocabulary, which have also been added into the tokenizer without any additional change.

Because the Python keywords and built-in function names have been prefixed in the new tokens, the string values in the output column of the dataset have been preprocessed by firstly using regular expression patterns to find all occurrences of keywords and built-in functions in each of the code snippets, then prefixing each of these occurrences (if there is any) with the corresponding prefix ("pytkw:" or "pytfunc:"). Furthermore, it has been found that all code snippets in the output column start with "```python" and end with "```". Since these two substrings are not part of the code and do not contain actual information, they have been removed from each of the values in the output column.

3. Model Fine-tuning with LoRA

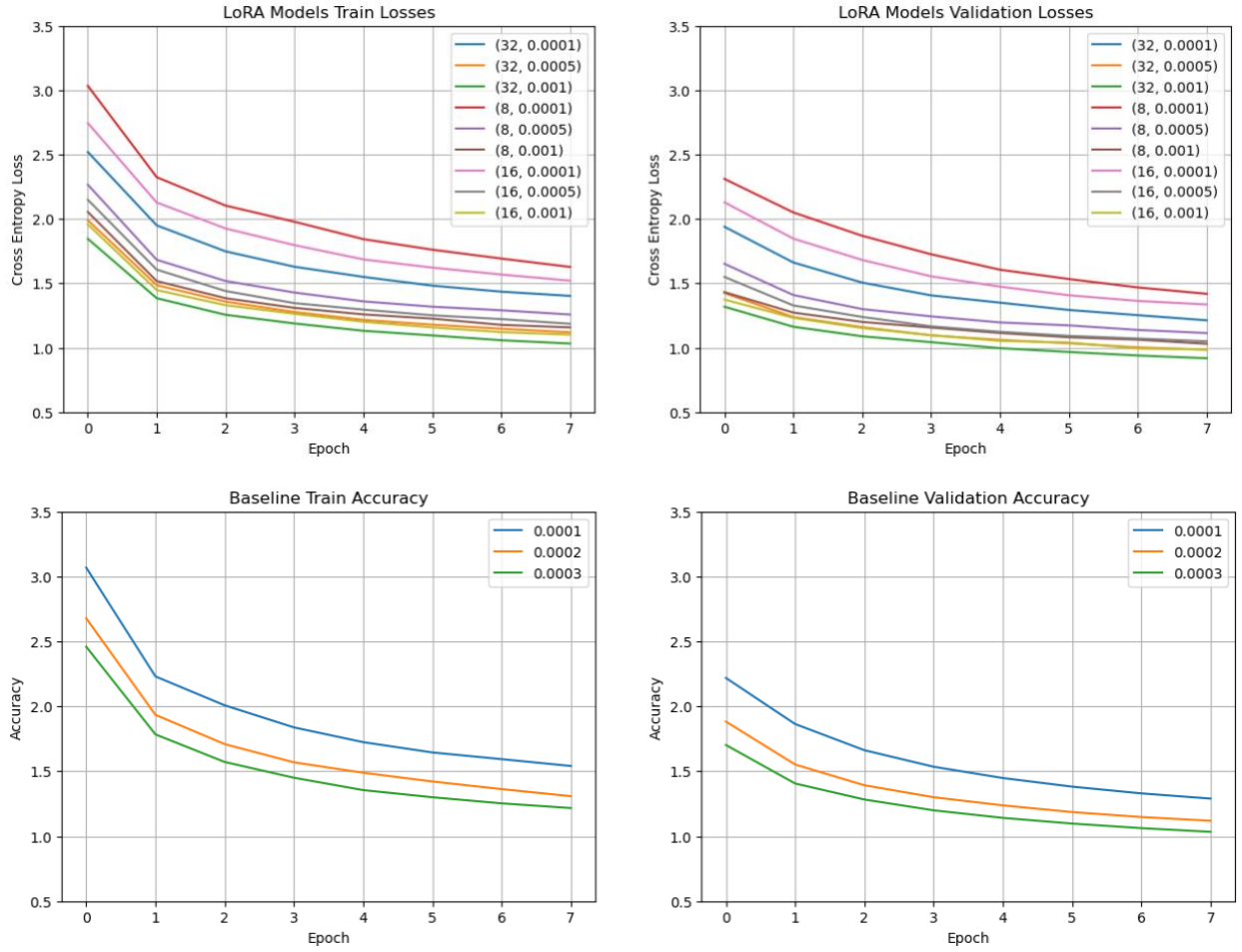
Because there is neither passages (information source) which the model can comprehend for short answer segment from, nor candidate answers which the model can select from, provided in the python-codes-25k dataset. A transformer-based encoder decoder generative language model will be needed for this code-related QA problem to encode instructions and decode into Python code. Hence, T5 model (T5-small in specific) has been selected as the pretrained model to be finetuned in this assignment's work. The pretrained encoder module of T5 model should be able to effectively encode/understand the information in the instructions, because the instructions are mostly simple and short natural language English sentences, and the T5 model has been pretrained with large and high-quality natural language dataset. The parameters in the decoder module will need to be fine-tuned to learn the syntax of Python and be able to generate Python code. As mentioned in Section 2 that new tokens have been added into the vocabulary of the tokenizer for the Python keywords and built-in function names. Because of this, new parameters have been added into the embedding layer, so the parameters in the embedding layer of the T5 model have been reset and needed to be retrained.

To integrate LoRA into the T5 model, a class (LoRAT5Attention) that inherits the class for computing attention for T5 model (T5Attention) has been created. Within the class, same as the work in the LoRA research paper, two pairs of A and B LoRA parameter matrices have been initialized for the 'query' and 'value' weight matrices. The B parameter matrices have been initialized with zeros, while random Gaussian initialization (mean=0, variance=1) have been used for the A parameter matrices. The scaling on ΔW by $\frac{\alpha}{r}$ has been overlooked in the assignment's implementation, since adjusting the learning rate later during training would have the roughly the same effect as tuning the value of $\frac{\alpha}{r}$.

A new wrapper class has also been created for T5 model to replace the T5Attention instances in T5 model's encoder, decoder, or both, with new LoRAT5Attention instances, and freeze the originally pretrained weights in the T5 model. To provide some flexibility, the initializer method of the new wrapper class has several flag parameters to decide whether to leave the parameters in the embedding layer, the bias, and/or the layer normalization trainable or not.

4. Training and Evaluation

A set of 9 T5 models with LoRA applied on their decoder modules have been finetuned with LoRA rank values 8, 16, and 32, and learning rates 1e-4, 5e-4, and 1e-3 respectively. The parameters in these models' embedding layers and layer normalizations have also been set trainable other than the LoRA parameters. In contrast, 3 baseline T5 models have been finetuned with only the parameters in their last two layers of their decoder modules, embedding layers and layer normalizations set to be trainable, with learning rates 1e-4, 2e-4, and 3e-4 respectively. All these 12 models have been finetuned through 8 epochs, with AdamW as the optimizer, and cross entropy loss as the loss function. The training and validation losses of these models across each of the 8 epochs have been recorded and demonstrated as below:



Both the training and validation losses generally decreased as the value of LoRA rank increased and the learning rate increased.

To evaluate the fine-tuned models' performance, the T5 model with LoRA applied on it, rank value of 32, and learning rate 1e-3, and the baseline model with learning rate 3e-4 have been used to perform code-related QA task on the test set. The tokenized instructions have been sent into the models' encoders to generate Python code from the decoder. Corpus-level BLEU score has been computed between each batch (batch size = 16) of the generated code snippets and each batch the correct code snippets in the output column of the dataset, and the batch BLEU scores have been averaged to get a final BLEU score on the whole test set. As a result, the T5 model with LoRA has gotten an average BLEU score of 0.081, while the baseline model has gotten an average BLEU score of 0.064.

The average BLEU scores on the test set for both the model with LoRA and the baseline model were low. This could because that there are multiple ways to write Python code for the same instruction (e.g., list comprehension mostly can be written with for/while loops in Python). There are also theoretically an infinite number of possible names for Python variables, custom functions, and classes. As a limitation of language model, it is unfeasible to create new tokens for all the unique Python variables', functions', and classes' names. However, having different variable, function, or class names in the generated and the correct code snippets will reduce the BLEU score. That being said, the T5 model with LoRA still slightly outperformed the baseline model on the BLEU score metric. Finetuning all the parameters in the baseline model's decoder module, or even the whole model might be able to achieve a higher BLEU score than the model with LoRA, but the number of trainable parameters will then be drastically larger than that of the model with LoRA, and the increase in BLEU score could eventually be relative negligible, compared to the increase in the number of trainable parameters.