Report
Fine-tuning a Pretrained Model Using LoRA

Lyle He

Contents

1 Understanding LoRA

1.1 Concept of LoRA

Low-Rank Adaptation (LoRA) is a new method used for adapting LLMs(Large Language Models, like GPT-3, ReBERTa) efficiently for downstream tasks. It proposes to freeze the original pretrained weights and injects trainable rank decomposition matrices into each layer of the Transformer architecture which is the architecture of LLMs (Edward Hu et al., 2021).

Here is a flowchart of the LoRA method from (Edward Hu et al., 2021):

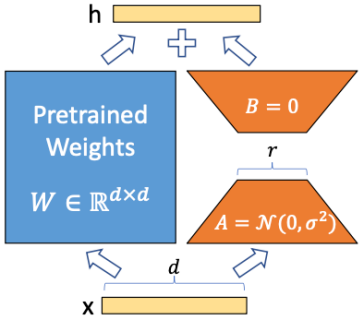

Figure 1: Reparametrization. They only train A and B

They freeze the shared model and efficiently switch tasks by replacing the matrices A and B in Figure 1, reducing the storage requirement and task-switching overhead significantly.

1.2 Benefits of LoRA

- **Much less parameters to be fine-tuned**: Significantly decreases the number of fint-tuning parameters by up to 10,000 times less than the full model and 3 times of GPU memory requirements.
- **No Additional Inference Latency**: Unlike methods that add layers or No Additional Inference Latency, LoRA does not.
- **Allow Many Task-Specific Adaptation and Good for Deployment**: LoRA allows for the pretrained model to be adapted to a wide variety of tasks by only changing a small set of parameters.

1.3 Mechanism of LoRA

- LoRA insert trainable low-rank matrices into each layer of the Transformer model. In detail, it applies a low-rank decomposition to represent the updates to adaptation of the weight matrices of the Transformer.
- The original weight matrices of the pretrained model are kept frozen, and only the parameters of the low-rank matrices are updated during the fine-tuning process.
- After fine-tuning, these low-rank updates can be merged back with the original weights for deployment.

1.4 Suitability of Pretrained Language Models for Code-Related QA Tasks

At the moment, the pretrained LLMs (such as GPT-3 and ReBERTa) have shown impressive performance on a wide range of NLP tasks. They have strong language interpretation and generation capabilities. At the same time, the programming language also have a lot of similarities with natural language including syntax and semantics. Therefore, the pretrained LLMs are suitable for code-related QA tasks.

1.5 Advantages of Using LoRA for Fine-Tuning on Code-Related Tasks

As mentioned above, LoRA needs much less parameters to be fine-tuned, which makes it efficient for adapting pretrained language models to code-related tasks. Based on this, it can be trained on a larger dataset to achieve better performance, which needs much less computation resources.

LoRA's characteristic of no additional inference latency achieves quick experimentation and deployment of models for new code-related tasks or programming languages.

2 Dataset Preparation

2.1 Overview of the "flytech/python-codes-25k" Dataset and Exploratory Data Analysis

The "flytech/python-codes-25k" dataset is a cleaned python dataset covering 25,000 instructional tasks [2]. This dataset can be used for code generation tasks, coding language understanding models, behavioral analysis based on the given tasks and codes, and educational purposes to understand coding styles and task variations.

There are four columns in the dataset: "instruction", "input", "output", and "text".

- "instruction": The instructional task to be performed / User input.
- "input": Very short, introductive part of AI response or empty.
- "output": Python code that accomplishes the task.
- "text": The combination of the "instruction", "input", and "output".

	output	input	instruction	text
count	49626.000000	49626.000000	49626.000000	49626.000000
mean	327.063434	9.518519	115.712530	489.743683
std	266.937459	26.135425	105.669716	318.460054
min	1.000000	0.000000	4.000000	57.000000
25%	144.000000	0.000000	63.000000	276.000000
50%	247.000000	0.000000	92.000000	396.000000
75%	420.000000	0.000000	131.000000	589.000000
max	2061.000000	163.000000	1708.000000	2399.000000

Figure 2: Preview of the dataset

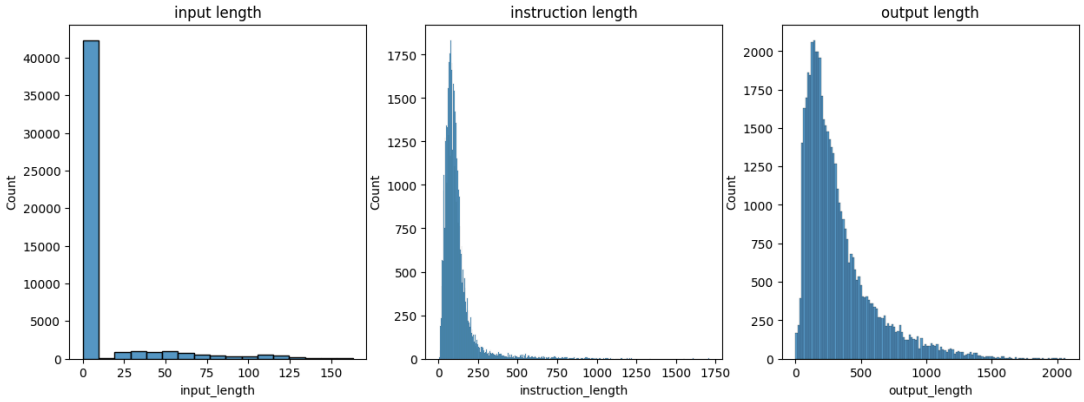

Figure 3: Distribution of the length

2.1.1 Dataset Statistics

Here are some statistics of the "flytech/python-codes-25k" dataset:

- Total samples: 49, 626: the dataset contains 49,626 samples.
- Total Entries: 24,813:
- Unique Instructions: 24,580
- Unique Inputs: 3,666
- Unique Outputs: 24,581
- Unique Texts: 24,813
- Average Tokens per example: 508

And according to input column, I statist the top 20 most frequent first words in the input column:

```
[('AI', 1590), ('Writing', 1022), ('Implementing', 264), ('Downloading', 260),
('Fixing', 246), ('Checking', 158), ('Creating', 148), ('Searching', 134), ('Opening',
112), ('Automatically', 110)]
```

As we can see, what kinds of tasks are most frequent in the dataset. And also, I count the task categories in the dataset by judging if the instruction column contains some keywords. The result is as follows:

```
Task categories counts:
{
'data manipulation': 4428,
'file IO': 1436,
'web scraping': 1526,
'algorithm': 2330
}
```

And I computed the average length of output column:

```
Average number of lines per snippet: 12.23322451940515
```

2.1.2 Dataset Examples

Input: Setting up your daily to-do list…
Output:

```
tasks = []
while True:
    task = input('Enter a task or type 'done' to finish: ')
    if task == 'done': break
    tasks.append(task)
print(f'Your to-do list for today: {tasks}')
```

———

Input: Creating a shopping list…
Output:

```
shopping_list = {}
while True:
    item = input('Enter an item or type 'done' to finish: ')
    if item == 'done': break
    quantity = input(f'Enter the quantity for {item}: ')
    shopping_list[item] = quantity
print(f'Your shopping list: {shopping_list}')
```

———

Input: Calculating weekly phone usage…
Output:

```
total_time = 0
for i in range(1, 8):
    time = float(input(f'Enter phone usage in hours for day {i}: '))
    total_time += time
print(f'You spend approximately {total_time} hours per week on your phone.')
```

2.2 Relevance for a Coding Related QA System

The “flytech/python-codes-25k” dataset is highly relevant for a coding-related QA system. It can be used for the following tasks:

- Code Generation. The instruction feature can used as input queries and the output feature can be used as the corresponding solutions. This achieves a autmatic converting from natural languages to code.
- Code Explanation: the opposite of the above task.
- Code Completion and Correction: Which similar to the above task.

2.3 Preprocessing

According to the previous analysis, I will make the following preprocessing steps:

1. Since my compute resource is limited, I will only use the first 500 samples of the dataset for the fine-tuning. (500 is the maximum number of samples I can use for the fine-tuning in my colab environment.)

2. Adjust input and labels for autoregressive models training, which means combining the "instruction", "input", and "output" columns into a single input sequence. In addition, I will add additional prompt before the text to indicate the task type.

```
prompt_template = """Here is an instruction for python code generation, Instruction:
{instruction}\n The response is as follows,  Response:"""
answer_template = """{input}\n{output}"""
```

3. Configure the input and labels for the model training and tokenization. As state in the last step, the labels of this training task is exactly the input of the model which obey the autoregressive model training.

```
def _preprocess_batch(batch: Dict[str, List]):
    model_inputs = tokenizer(batch["text"], max_length=MAX_LENGTH, truncation=True,
padding='max_length')
    model_inputs["labels"] = copy.deepcopy(model_inputs['input_ids'])
    return model_inputs
```

where The tokenizer is the Roberta tokenizer from the huggingface library.

4. Split the dataset into training, validation, and test sets which the ratio is 8:1:1.

3 Model Fine-Tuning with LoRA

3.1 Model Selection and Architecture Exploratory

According to the (Edward Hu et al., 2021), I will choose the Roberta base model from Facebook AI. The Roberta is an optimized BERT Pretraining method for pretraining self-supervised NLP systems which is also similar to autoregressive models training and suitable for code-related QA tasks.

The `flytech/python-codes-25k` dataset has a 'text' column which contains the instruction, input, and output of the task which is exactly used for autoregressive and self supervised training. Therefore, the Roberta model is suitable for this task. The expected performance should be good.

Here is key part of the architecture of the Roberta model, for simpycity, please refer to my ipynb file.

```
(attention): RobertaAttention(
            (self): RobertaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): RobertaSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-05, elementwise_affine=True)
```

```
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
```

As we can see the attention mechanism related matrices are the “query”, “key”, and “value”, so next I will inject trainable low-rank matrices into these matrices.

3.2 Integration of LoRA

According to the PEFT library, I will use the following code to integrate LoRA into the Roberta model:

```
from peft import LoraConfig, get_peft_model, prepare_model_for_int8_training
LORA_R = 256
LORA_ALPHA = 256
LORA_DROPOUT = 0.05
lora_config = LoraConfig(
                  r = LORA_R,
                  lora_alpha = LORA_ALPHA,
                  lora_dropout = LORA_DROPOUT,
                  bias="none",
                  task_type="CAUSAL_LM",
                  target_modules=["query", "key", "value"])

model = get_peft_model(model, lora_config)
model.print_trainable_parameters()
```

As we can see, I set the rank of the low-rank matrices as 256, lora alpha as 256, and the dropout rate as 0.05.

Since this is a text generation task, I use the “CAUSAL_LM” as the task type. As we can see I specify the target modules as “query”, “key”, and “value” which are the self attention mechanism related matrices of transformer.

Then after the integration, the architecture of the model will be like this (here I just snippet the modified part)

```
(attention): RobertaAttention(
                (self): RobertaSelfAttention(
                  (query): lora.Linear(
                    (base_layer): Linear(in_features=768, out_features=768, bias=True)
                    (lora_dropout): ModuleDict(
                      (default): Dropout(p=0.05, inplace=False)
                    )
                    (lora_A): ModuleDict(
                      (default): Linear(in_features=768, out_features=256, bias=False)
                    )
                    (lora_B): ModuleDict(
                      (default): Linear(in_features=256, out_features=768, bias=False)
                    )
                    (lora_embedding_A): ParameterDict()
                    (lora_embedding_B): ParameterDict()
                  )
                  (key): lora.Linear(
                    (base_layer): Linear(in_features=768, out_features=768, bias=True)
                    (lora_dropout): ModuleDict(
                      (default): Dropout(p=0.05, inplace=False)
                    )
                    (lora_A): ModuleDict(
                      (default): Linear(in_features=768, out_features=256, bias=False)
```

```
      )
      (lora_B): ModuleDict(
        (default): Linear(in_features=256, out_features=768, bias=False)
      )
      (lora_embedding_A): ParameterDict()
      (lora_embedding_B): ParameterDict()
    )
    ...
```

As we can see the lora.Linear of A and B are injected into the original query, key, and value matrices.

3.3 Metrics and Training Configuration

I use BLEU score as the evaluation metric for this text generation task. In addition, I use `evaluate.load("bleu")` to load the BLEU score, `TrainingArguments` to configure the training process, and `Trainer` to train the model. For more details, please refer to my ipynb file.

And here is the hyperparameters I used for the training:

Hyperparameters	Values
Learning Rate	5e-4
Epochs	10
Batch Size	16

4 Training, Evaluation, and Testing

4.1 Training and Evaluation Process and Results

4.1.1 Training Configurations

As mentioned above, the configurations are shown as follows:

Hyperparameters	Values
Task Type	CAUSAL_LM
Target Modules	query, key, value
LoRA Rank	256
LoRA Alpha	256
LoRA Dropout	0.05
Learning Rate	5e-4
Epochs	10
Batch Size	1
Evaluation Metric	BLEU Score

4.1.2 Training and Evaluation Results

The training results are shown as follows:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 49626 entries, 0 to 49625
Data columns (total 4 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   output       49626 non-null  object
 1   input        49626 non-null  object
 2   instruction  49626 non-null  object
 3   text         49626 non-null  object
dtypes: object(4)
memory usage: 1.5+ MB
```

Figure 4: Training and validation loss

4.2 Testing Results

The testing results are shown as follows:

```
Test Evaluation Results:
{'eval_loss': 1.555329442024231,
'eval_bleu': 0.35416743245504645,
'eval_runtime': 4.552,
'eval_samples_per_second': 10.984,
'eval_steps_per_second': 10.984,
'epoch': 3.0}
```

As we can see, the BLEU score of the fine-tuned model is 0.35.

4.3 Custom Testing

I also test the model with some custom inputs and the results are as follows:

As we can see, the model can generate the correct python code for the given task.

4.4 Analysis of Results

The BLEU score (0.35) of the fine-tuned model is not too high, but the custom testing results show that the model can generate the correct python code for the given task. This may be due to the small size of the dataset and the limited compute resources. Then, the LoRA method is efficient and effictive for fine-tuning the pretrained language model for code-related tasks.

5 References

[1] Edward Hu et al. (2021). Low-Rank Adaptation of Large Language Models. arXiv preprint arXiv:2102.13167.
[2] https://huggingface.co/datasets/flytech/python-codes-25k