Fine-tuning a small part of model parameters

Adapter, Prefix-tuning, LoRA & others

Mark Litvinov, AMI HSE

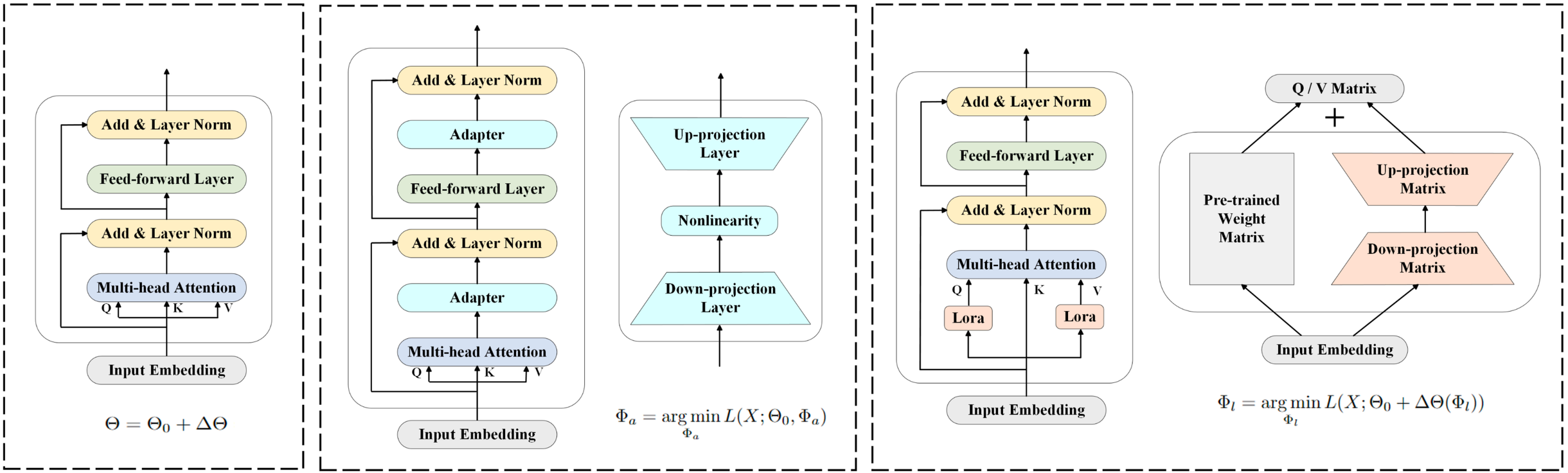
The Plan

1. Reasons For Fine-Tuning
2. FMFT, PEFT, Adapter, LoRA
3. Comparison of Approaches
4. Prefix-tuning

Reasons for Partial / Parameter Efficient FT

- Computational Efficiency and Resource Constraints
- Mitigation of Catastrophic Forgetting
- Adaptability and Generalization to Specific Tasks

FMFT, PEFT, Adapter, LoRA



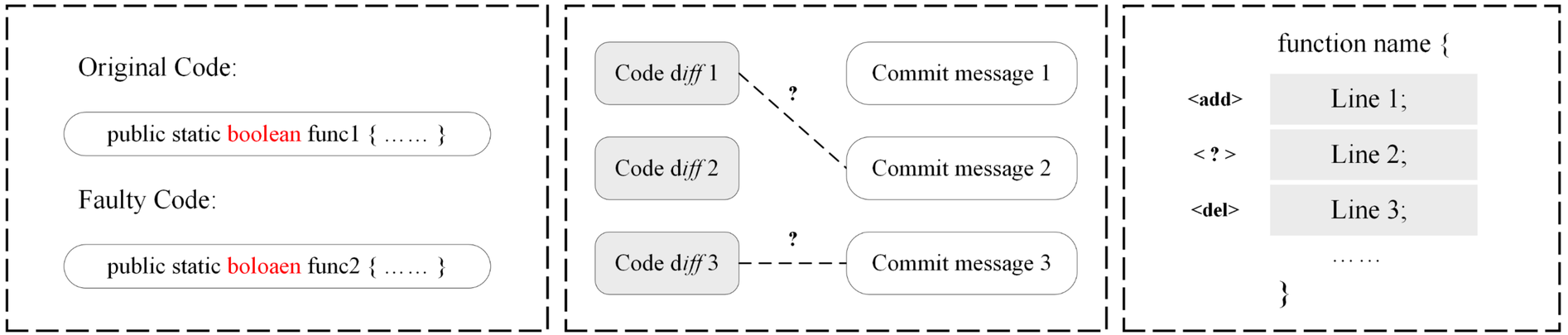
(a) FMFT (b) Adapter Tuning (c) LoRA

Tasks

Just-In-Time Defect Prediction - JIT-DP aims to identify defective code changes when they are just committed, and return judgements

Commit Message Generation - Commit message sum- marizes the intent and content of a code change, which is helpful for developers to understand programs quickly in soft- ware maintenance

Probing Tasks - three probing tasks that are related to code properties from different aspects: Invalid Type Detection (TYP), Code Change Match (CCM), Line Type Prediction (LTP)



(a) Invalid Type Detection (b) Code Change Match (c) Line Type Prediction

Datasets

TABLE I
DATASET STATISTICS.

Tasks	Datasets	Training	Validation	Test
JIT-DP	JIT-Defects4J	16,374	5,465	5,480
CMG	Java	160,018	19,825	20,159
	C#	149,907	18,688	18,702
	C++	160,948	20,000	20,141
	Python	206,777	25,912	25,837
	JavaScript	197,529	24,899	24,773
Probing Tasks	TYP	600	200	200
	CCM	600	200	200
	LTP	600	200	200

Models and baselines for AT and LoRA

1. CodeBERT: It is an encoder-only PLM with 125M parameters, pre-trained on CodeSearchNet [22] which contains 2M bimodal data
2. GraphCodeBERT: It has the same architecture and parameter size as CodeBERT, while pre-trained with edge prediction and node alignment to learn from data flow
3. PLBART: It is pre-trained on Java and Python datasets, as well as natural language descriptions. It is an encoder-decoder architecture that has 140M parameters
4. UniXcoder: It is a unified encoder-decoder PLM with 125M parameters, and can be flexibly modified to encoder-only or decoder-only architecture
5. CodeT5: It contains 220M parameters, and is a representative PLM of encoder-decoder architecture for its well performance in generation tasks

Baselines - LLM’s

1. JITLine: It is a baseline of JIT-DP, which represents code changes by bag-of-tokens features and constructs classifiers like SVM to predict defective commits.

2. JITFine: It uses CodeBERT as an encoder, and in- corporates the encoded code change representations with extra expert features, achieving a substantial improvement in the JIT-DP task.

3. CC2Vec: It is a baseline of JIT-DP, which utilizes a hierarchical attention network and emphasizes modeling the correlation between removed codes and added codes.

4. CodeReviewer: It is proposed for code review ac- tivities, but can be extended to the CMG task. It proposes four pre-training tasks to improve CodeT5’s [59] capacity for understanding code changes.

5. CCT5: It is the state-of-the-art baseline for both JIT-DP and CMG. It also pre-trains CodeT5 [59] with code-change-related corpus and objectives.

Metrics

Just-In-Time Defect Prediction - F1 + AUC ROC

Commit Message Generation: For the CMG task, we evaluate the generated commit messages using three metrics: BLEU [42], Meteor [2], and Rouge-L [35]. BLEU calculates the n-gram precision between the generated text and ground truth text. Meteor takes into account the precision, recall, and fluency of the generated text. Rouge-L focuses on the longest common subsequence so that it evaluates more about the word order

Probing Tasks: Due to probing tasks being classifica- tion tasks, we use accuracy as the evaluation metric

JIT-DP Results

TABLE VI
EVALUATION RESULTS OF PROBING TASKS FOR JIT-DP. THE BEST AVERAGE PERFORMANCE IS HIGHLIGHTED IN BOLDFACE.

	Models	TYP	CCM	LTP
FMFT	CodeBERT	54.0	50.0	62.5
	GraphCodeBERT	78.5	58.0	71.5
	PLBART	80.0	62.0	73.5
	UniXcoder	89.0	52.5	65.5
	CodeT5	91.5	70.0	79.5
	Avg.	78.6	58.5	70.5
LoRA	CodeBERT	91.0	50.5	72.5
	GraphCodeBERT	97.5	61.5	71.5
	PLBART	87.5	55.5	73.5
	UniXcoder	95.5	58.5	67.5
	CodeT5	92.5	64.5	77.0
	Avg.	**92.8**	58.1	72.4
Adapter Tuning	CodeBERT	91.0	56.0	65.0
	GraphCodeBERT	94.0	61.0	71.0
	PLBART	84.5	63.0	76.5
	UniXcoder	93.5	55.5	70.5
	CodeT5	95.5	72.0	80.5
	Avg.	91.7	**61.5**	**72.7**

CMG Results

TABLE VII
EVALUATION RESULTS OF PROBING TASKS FOR CMG. THE BEST AVERAGE PERFORMANCE IS HIGHLIGHTED IN BOLDFACE.

Models		TYP	CCM	LTP
FMFT	Java	86.0	55.5	77.5
	C#	88.0	57.0	78.0
	C++	85.5	57.5	79.0
	Python	86.0	51.0	75.5
	JavaScript	88.5	54.0	79.0
	Avg.	86.8	55.0	77.8
LoRA	Java	84.5	65.0	80.0
	C#	91.0	54.5	77.5
	C++	90.0	58.0	77.5
	Python	88.5	58.5	76.5
	JavaScript	86.5	60.5	73.5
	Avg.	**88.1**	59.3	77.0
Adapter Tuning	Java	80.5	59.0	79.0
	C#	83.5	62.0	83.5
	C++	84.0	68.5	81.0
	Python	81.5	65.0	83.5
	JavaScript	92.0	63.0	84.0
	Avg.	84.3	**63.5**	**82.2**

Takeaways

Finding 1: Compared with FMFT, Adapter tuning and LoRA show their effectiveness in the JIT-DP task, especially when incorporating extra expert features. They obtain state-of-the-art results that achieve improvements of 8.39% and 9.87% in terms of F1 sore compared with the SOTA approach, respectively.

Finding 2: In the CMG task, adapter tuning and LoRA can achieve similar performances compared with fine-tuning and state-of-the-art baselines, while they consume less training time and memory size.

Finding 4: Adapter tuning and LoRA show effectiveness in the low-resource scenario. Practitioners are recommended to use PEFT techniques in the low-resource setting as their performance is outperforming or at least comparable to FMFT.

Prefix-Tuning

Prefix-tuning is a fine-tuning method for neural models that involves adding trainable prefixes to each attention layer in the model. These prefixes act as additional inputs, allowing the model to adapt to new tasks with minimal changes to its original weights

Mechanism: In every attention layer, trainable prefixes ($s_{l1}, \ldots, s_{lnS}$) are introduced, added to the layer's input ($x_{l1}, \ldots, x_{lnX}$), forming a new input ($s_{l1}, \ldots, s_{lnS}, x_{l1}, \ldots, x_{lnX}$).

Theory

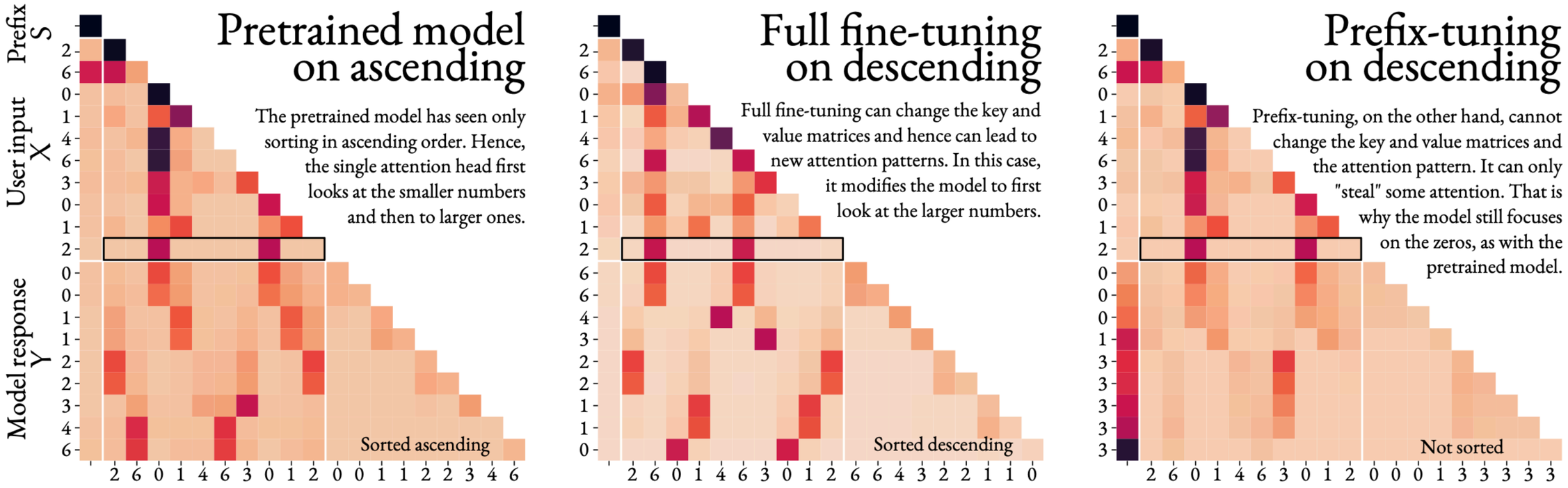
Virtual Token Explanation: A virtual token is an engineered input that represents a point in the embedding space, distinct from the natural language tokens found in the model's training data.

Theorem 1: a model can generate any sequence from all possible combinations using just one virtual token, significantly expanding the model's expressive capabilities beyond traditional token use.

Theorem 2: a model can control the output in response to any user input using just one virtual token, making prefix-tuning more expressive than simple token addition.

Significance: These theorems confirm that prefix-tuning possesses a high degree of adaptability and expressiveness, allowing the model to efficiently use the embedding space to tackle a variety of tasks.

Example With Sorting Model



Pros & Cons

Adaptation to New Tasks: with pre-trained hidden skills

Preservation of Knowledge: Minimizes the forgetting of previous knowledge, maintaining the original structure and weights of the model

Expressiveness: Offers greater expressive power compared to traditional tuning, allowing the model to generate more complex and varied responses

Restriction: While full fine-tuning can alter the attention pattern of an attention head, prefix-tuning cannot