Course Project: Part 2

Generative AI

Saarland University – Winter Semester 2024/25

Martínez
7057573
`cama00005@stud.uni-saarland.de`

Part 2.a: Fine-tuning Baseline Models

Table 1: Program repair quality metrics, *RPass* and *REdit*, for the finetuned models on Repair and Hint tasks. In parentheses, the change in the metric compared to the Repair model is shown, where green means an improving change and red a worsening change. For **(I.11)**.

Model	***RPass***	***REdit***
Phi-3-SFT-Hints_r16_alpha32	72.0 (-4.0)	21.22 ($+11.33$)
Phi-3-SFT-Repair_r16_alpha32	76.0 (–)	9.89 (–)

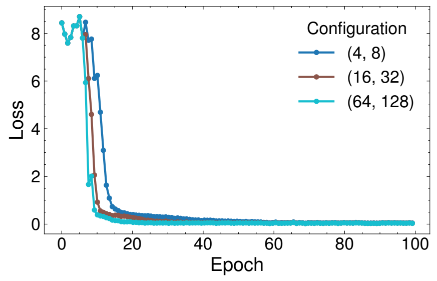
Part 2.b: Fine-tuning for Program Repair with Varying LoRA Parameters

Table 2 shows a comparison between different finetuned versions of the Phi-3-mini model on the Repair task using varying LoRA parameters. Recall that in Project Part#1 (see Table 4 in **Appendix**), the base Phi-3-mini scored 36.0 in *RPass* and 18.11 in *REdit*, which is significantly worse than any of the finetuned models. This means that the fine-tuning process with LoRA was effective and greatly improved the model's performance on the Repair task.

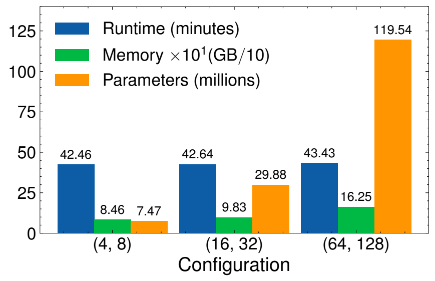
Table 2: Program repair quality metrics, *RPass* and *REdit*, for the finetuned models on Repair task using different LoRA parameter configurations: $(r, \alpha) = (4, 8), (16, 32), (64, 128)$. In parentheses, the change in the metric compared to a base configuration, chosen to be $(16, 32)$, is shown, where green means an improving change and red a worsening change. For **(I.13)**.

(r, α)	***RPass***	***REdit***
$(4, 8)$	60.0 (-16.0)	11.47 ($+1.58$)
$(16, 32)$	76.0 (–)	9.89 (–)
$(64, 128)$	84.0 ($+8.0$)	17.14 ($+7.25$)

(I.14) Figure 1 shows that the *TrainingTime* of all three LoRA configurations are very similar. The main difference thus lies in the *TrainingMemory* and the number of trained parameters. Naturally, the $(64, 128)$ configuration has the highest *TrainingMemory* and number of trained parameters, which is expected given the higher values of r and α. More precisely, the number of trained parameters increases approximately by a factor of 4 when going from $(4, 8)$ to $(16, 32)$, and by a factor of 4 again when going from $(16, 32)$ to $(64, 128)$. This is consistent with the theoretical analysis of LoRA, which predicts that the number of parameters scales quadratically with the values of r and α [1].



(a) Loss over training epochs.



(b) *TrainingTime*, *TrainingMemory* and number of trained parameters comparison.

Figure 1: Trade-offs between LoRA parameter configurations in terms of loss evolution, *TrainingTime*, *TrainingMemory*, and number of trained parameters. These experiments were conducted with a GPU T4 instance on Google Colab. For **(I.13)**.

On the other hand, the increase of *TrainingMemory* between configurations is not as extreme. From $(4, 8)$ to $(16, 32)$, there is a 16% increase, and from $(16, 32)$ to $(64, 128)$, there is a 65% increase. This is likely due to the fact that the memory requirements of the model are not only determined by the number of parameters, but also by the size of the intermediate activations and the gradients during training.

Finally, in terms of program repair quality, the $(64, 128)$ configuration outperforms the other two configurations in terms of *RPass*, but has the highest *REdit*, suggesting that it might be overfitting to the training data. The $(4, 8)$ configuration has the lowest *RPass* and the second highest *REdit*, indicating that it is underfitting. The $(16, 32)$ configuration is a good balance between the two, achieving a value of *RPass* only 8 points lower than the best, but at the same time the lowest *REdit*.

The choice between the three configurations will depend on the specific requirements of the application. Personally, in the feedback generation domain, the $(16, 32)$ configuration would be the most suitable, as it can generate accurate repairs (high *RPass*) with the lowest number of changes to the buggy code compared to the other models (low *REdit*). This allows a student to learn from the generated feedback without being overwhelmed by too many changes or dissapointed that the complete idea of the code was changed.

Part 2.c: Fine-tuning on Merged Datasets

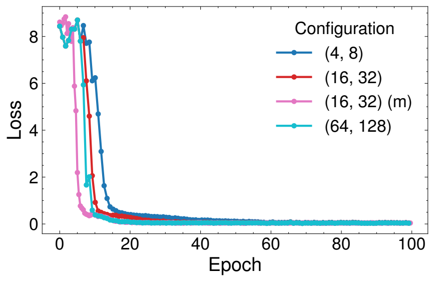
Table 3: Program repair quality metrics, *RPass* and *REdit*, for the finetuned models on Repair, Hint and both tasks combined. In parentheses, the change in the metrics compared to the multi-task model (Phi-3-SFT-Merged_r16_alpha32) is shown, where green means an improving change and red a worsening change. The results of **(I.11)** are included for ease of comparison. For **(I.15)**.

Model	***RPass***	***REdit***
Phi-3-SFT-Hints_r16_alpha32	72.0 (-8.0)	21.22 ($+6.62$)
Phi-3-SFT-Repair_r16_alpha32	76.0 (-4.0)	9.89 (-4.71)
Phi-3-SFT-Merged_r16_alpha32	80.0 (–)	14.6 (–)

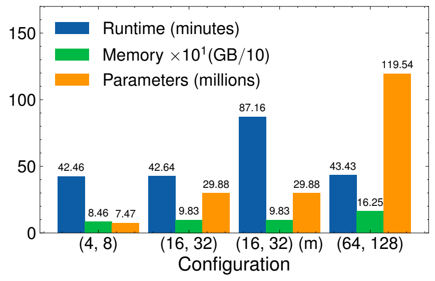
(I.17) In order to create the merged dataset to train the multi-task model (Phi-3-SFT-Merged_r16_alpha32), I added a simple concatenation of both datasets (Repair + Hint) in `project_part2_assemble_dataset.py` and a shuffling of the result afterwards.

Figure 2 shows the trade-offs between the specialized models and the multi-task model in terms of loss evolution, *TrainingTime*, *TrainingMemory*, and number of trained parameters. The multi-task model with $(r, \alpha) = (16, 32)$ has a higher *TrainingTime*, but suprisingly has a better loss evolution compared to the specialized models. That is, the loss decreases rapidly in the first few epochs, indicating rapid learning, and then plateaus similar to the other models.

On the other hand, the *TrainingMemory* between the specialized model for the Repair task and the multi-task model is the same.



(a) Loss over training epochs.

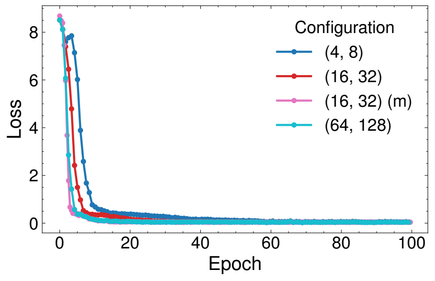


(b) *TrainingTime*, *TrainingMemory* and number of trained parameters comparison.

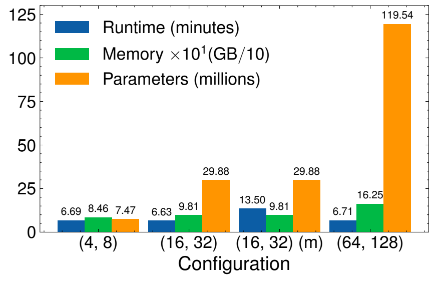
Figure 2: Trade-offs between the specialized models and the multi-task model in terms of loss evolution, *TrainingTime*, *TrainingMemory*, and number of trained parameters. The (m) indicates the multi-task model. These experiments were conducted with a GPU T4 instance on Google Colab. For **(I.17)**.

Figure 3 shows the same concepts, as Figure 2, but for a GPU A100 instance. Using this GPU significantly reduced the *TrainingTime* for all models, especially for the multi-task model. Nevertheless, if we take into account relative performance, the multi-task model still requires $2\times$ the *TrainingTime* of the specialized model for the Repair task, i.e., 6.63 minutes vs. 13.50 minutes with an A100 and 42.64 minutes vs 87.16 minutes with a T4, respectively.

On the other hand, the loss over training epochs showcase the *scaling laws*, as we learned in Week 4. All models showcase a stepper and smoother decrease in loss as the number of epochs increases. That is, a higher computing power effectively reduced the time to reach a certain loss value, eventhough the hyperparameters, model architecture and datasets were the same [2].



(a) Loss over training epochs.



(b) *TrainingTime*, *TrainingMemory* and number of trained parameters comparison.

Figure 3: Trade-offs between the specialized models and the multi-task model in terms of loss evolution, *TrainingTime*, *TrainingMemory*, and number of trained parameters. The (m) indicates the multi-task model. These experiments were conducted with a GPU A100 instance on Google Colab. For **(I.17)**.

Acknowledgements

Week 10's slides and listed references.

References

[1] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[2] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020.

Appendix: (I.3) Results of Project Part#1

Table 4: Overall results for GPT-4o-mini and Phi-3-mini with the basic prompt to generate program repairs. For **(I.1)** and **(I.3)**.

Model	***RPass***	***REdit***
GPT-4o-mini	88.0	23.77
Phi-3-mini	36.0	18.11