MUSIC: Make Unification Simple In Image Classification

Muhammad Shahbaz*, Hamza Waheed*, Abdullah Asghar*, Abdul Wahab*
Masters in Data Science
FAST National University of Computer and Emerging Sciences
Emails: i248029@isb.nu.edu.pk, i247628@isb.nu.edu.pk, i248000@isb.nu.edu.pk, wahab24@example.com

***Abstract*—Training a model is the backbone of Machine learning because all the decisions will be done on its basis. Once it is trained we further improve it on the basis of feedback and our requirements. To perform this we use the process of fine-tuning, it is process to further retrain the pre-trained model on small target dataset, we combine multiple tasks and then retrain it on already trained model to improve the performance on these specific tasks. The accuracy achieved is very low after fine-tuning.We observed the flaw in the process that when we merge our tasks and then do fine-tuning, our performance decreases. To solve this issue the concept of MUSIC (Make Utility Simple In Image Classification) takes place, we simply fine-tune the model on each task individually instead of merging them as one and as a result we achieved accuracy of 88% from 70% which is a drastic difference.**

***Index Terms*—**

Fine-tuning, Editng models, Artificial Intelligence

I. Introduction

In the last few years AI (Artificial Intelligence) has taken the world. AI models are trained on data to perform specific tasks. Common type of AI models include Supervised learning models which are trained in labeled data to predict outcomes such as regression and classification , Unsupervised learning models which are trained on unlabeled data to identify patterns such as K means classification, Deep learning models which uses neural networks like humans and process complex data like images, voice notes etc. AI models are used commonly as core of AI, we often edit them after training to achieve desired results. We train them to change the behavior, if we want to increase the response of specific class in case of classification then we need to retrain the model accordingly and vice versa. This process is called fine-tuning and it is different from training a model. In the context of training the model has not learned anything we assign it parameters and weights to operate while fine-tuning is process after training model where we take the pre trained model's behavior as a starting point and then tune it on specific tasks.

While performing fine tuning for experimentation on MNIST dataset we observed that overall our accuracy is decreasing, before fine tuning our accuracy was 91.70% and after fine tuning our accuracy decreased to 70% which is a huge difference.

A. Methamatical Equations

The fine-tuning process can be mathematically represented as follows:

1. Start with the pre-trained model:

$$\mathcal{M}_0(\theta_0) \tag{1}$$

2. Fine-tune on the dataset of class 1 ($\mathcal{D}_1$):

$$\mathcal{M}_1(\theta_1) = \arg\min_{\theta} \mathcal{L}(\mathcal{M}(\theta), \mathcal{D}_1) \tag{2}$$

3. Fine-tune the updated model on the dataset of class 2 ($\mathcal{D}_2$):

$$\mathcal{M}_2(\theta_2) = \arg\min_{\theta} \mathcal{L}(\mathcal{M}_1(\theta), \mathcal{D}_2) \tag{3}$$

4. Fine-tune the updated model further on the dataset of class 3 ($\mathcal{D}_3$):

$$\mathcal{M}_3(\theta_3) = \arg\min_{\theta} \mathcal{L}(\mathcal{M}_2(\theta), \mathcal{D}_3) \tag{4}$$

Here:

- $\mathcal{L}$ represents the loss function (e.g., cross-entropy loss).
- $\mathcal{M}_i(\theta_i)$ denotes the model fine-tuned after training on class i.

Summary of the Fine-Tuning Process

In this approach, the pre-trained model $\mathcal{M}$ is fine-tuned incrementally on datasets corresponding to individual classes. Instead of merging datasets, we fine-tune the model sequentially for better class-specific learning.

1) We begin by fine-tuning the pre-trained model on data from class 1, updating its parameters to θ_1.
2) Next, the model fine-tuned on class 1 is further fine-tuned on data from class 2, resulting in updated parameters θ_2.
3) Finally, the model fine-tuned on class 2 is fine-tuned on class 3, producing the final model with parameters θ_3.

This method ensures that each class contributes to the model's performance individually, preventing performance degradation that can arise from merging tasks. As a result, we achieved a significant accuracy improvement compared to merging all classes before fine-tuning.

II. Related Work

a) Editing Models with Task Arithmetic: Training a model is the backbone of Machine learning because all the decisions will be done on its basis [1]. Once training is done we retrain it according to our needs and feedback. This process is very costly due to following factors: Time and capital is effected most on retraining as we need to perform that process again so more time is utilized due to which more money is exhausted. We only have trained model instead of training data so we need to collect it again so need resources to do it. Fine tuning was introduced to address this problem but it requires labeled data so we need more efficient solution, here comes the solution of editing models with task arithmetic operations using task vectors. Task vector is weights in direction (Positive / Negative) which help the model to perform better. For example if we want to mitigate undesirable behaviors then we can change behaviors of model by negating as task vector similarly if we can to make model to learn new thing then we can do it by adding task vector. This performs well target tasks or even also improves performance and also efficient to compute as compared to fine tuning.

b) LANGUAGE MODELS ARE HOMER SIMPSON! Safety Re-Alignment of Fine-tuned Language Models through Task Arithmetic: The introduction tells about the advances and growth of LLMs [2], which have demonstrated satisfaction in different tasks. However, the fine-tuning process, which is meant to improve the model functionality in defined areas, usually results in compromise of safety. The authors cite earlier works that explain how fine-tuning can argue indirectly make the models unsafe. Here comes the concept of RESTA, whose aim is to reinstate safety without paying a heavy price in performance appeal. The RESTA Method can be defined in either terms – simplicity and effectiveness. The primary activity consists of a safety vector being added in an elemental manner, to the model parameters. In addition to the authors also came up with a method called Drop and REscale (DARE) which assists in eliminating the extra parameters that were captured during the fine-tuning to increase the efficiency of the safety vectors. The authors made evaluations of RESTA on two ways of fine-tuning parameters called the parameter efficient fine-tuning PEFT and the full fine-tuning FullFT. It was observed that these two approaches compromised the safety of the models when used on several tasks, including those with harmless datasets. In order to evaluate the performance of RESTA, the authors created a safety evaluation benchmark known as CATQA, which contains 550 dangerous questions that have been divided into 11 groups, each of 5 subcategories. Such a benchmark was purposefully developed in order to include all the abusive cases specified by OpenAI and Meta's usage regulations. As it turned out, the results of the evaluations were promising. The authors noted that the fine-tuned models had a great decrease in the unsafety scores after applying RESTA. When for instance, the Llama-2 model was tested on CATQA, the unsafety score reduced from 33.57% to 12.17% in PEFT while in FullFT, the figure reduced from 22.16% to 4.34%. These results shows that RESTA enhances safety without compromising the performance of the model in a variety of tasks.

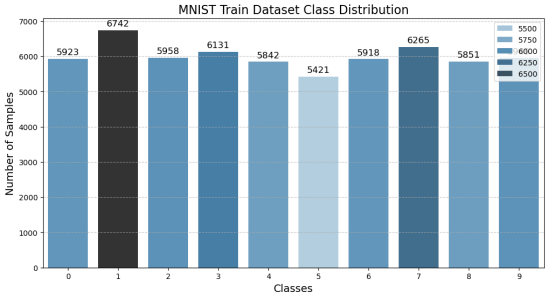
c) Task Arithmetic in the Tangent Space: Improved Editing of Pre-Trained Models: Task arithmetic refers to the ability to perform arithmetic operations on the weights of a model to get desired outcomes for multiple tasks [3]. This is cost-effective and scalable approach to edit pre-trained models directly in weight space. By changing the weights associated with different tasks, researchers can enhance a model's performance on those tasks or even negate certain tasks, leading to a phenomenon known as task forgetting. The authors highlight that traditional model editing methods often involve costly joint fine-tuning across multiple tasks, which can degrade the model's pre-training performance or zero-shot accuracy. Task arithmetic offers a promising alternative by allowing for more efficient adjustments to the model's weights. One of the most important contributions of the paper is a proposal to linearizing models in order to enhance weight disentanglement. The authors show that fine-tuning a model in its tangent space can amplify the disentanglement of weights and thus effectively improve performances across benchmark arithmetic tasks. Finally, the authors presented empirical results that linearizing models can bring 5.8 points of accuracy in task addition and 13.1 points less in task negation on various vision-language benchmarks.

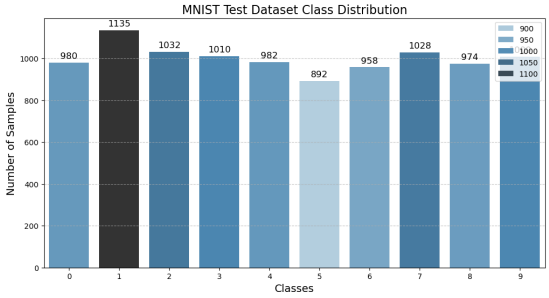
Experiment indicated that fine-tuning in the tangent space significantly improved the arithmetic benchmark for most tasks compared to the pre-trained models.

III. Design and Implementation

A. Data Set

MNIST (Modified national institute of standards and technology) digits dataset is a large dataset used in computer vision field for classification of numbers. MNIST dataset contain 60000 images for training and validation as well as 10000 images for training purpose. Size of each images is 28 * 28. By using Pytorch's'torchvision' library download and load the handwritten digits dataset of MNIST. Also performing the pre-processing step on dataset by converting each image to tensor also normalize each tensor using transform.Compose. There are 60000 samples in the training dataset and dimension of each image in dataset in [1, 28, 28]. MNIST Digits dataset contain 0 to 9 unique classes.





Here is the visualization of all classes using bar plot to show the count of each class in the train and test dataset.

Now we extracted 1000 samples of each class 3, 4 and 7 from the training dataset and saved in the separate files for later use. Now 57000samples left in the training dataset that we saved in another file train_rem.pkl. Now load the training dataset from train_rem.pklfile also load the testing dataset. Following is the EDA of training and testing datasets.

Training data:

- Total Samples: 57000
- Number of Classes: 10
- Classes: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
- Train Data Shape: torch.Size([32, 1, 28, 28])
- Train Labels Shape: torch.Size([32])

Visualizing random samples of train dataset

Train data classes distribution using boxplot Testing Dataset:

- Total Samples: 10000
- Number of Classes: 10
- Classes: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
- Test Data Shape: torch.Size([32, 1, 28, 28])
- Train Test Labels Shape: torch.Size([32])

Visualizing random samples of test dataset

B. Enviornment Specifications

It is necessary to stress that all factors of the experimental setting were chosen deliberately. The implementation was done in Python 3.12 and the PyTorch Lib 2.0.1 for purposes of model designing and training. Data cleaning and initial data exploration were done with NumPy and Pandas while the metric calculations were done with the use of scikit-learn 1.3.1. For data visualization, Matplotlib and Seaborn libraries were used because they provide easy to understand plots and charts. The code execution was carried out from within Jupyter Notebook, which is operated from the Anaconda platform. Since, it was essential to maintain homogeneity and compartmentalization of all application dependent elements, a virtual environment was built out using pip and conda. The experiments were carried out on the CPU given the unavailability of a GPU, while computational threading was used to enhance concurrency.

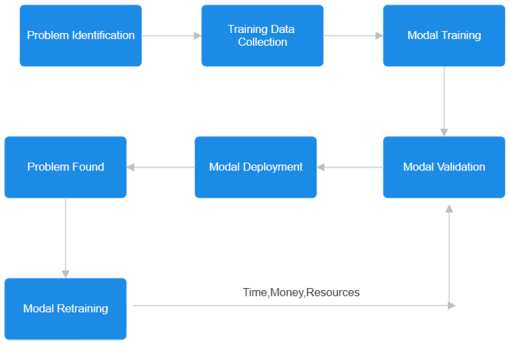
The hardware specifications of the system used were as follows: A MacBook Pro using kernel release 24.1.0, architecture: x86_64. The system was run with an OS of 2.8 GHz Quad-Core Intel Core i7 processor featuring 4 physical cores, 4 spares and thus 8 CPUs. For the computational needs in the experiments, it had 16 GB of 2133 MHz LPDDR3 RAM for sufficient resources.

C. Methodology

Let's suppose you are a student and your professor assigned you a task to learn 30 topics till tomorrow, Will you be able to learn it? Yes, you can but your performance will be compromised. Now if you are asked to learn 10 topics in a day and total 30 topics in 3 days. Now your response will be much better as compared to first case. Here we used the same concept in our training. Instead of fine tuning our model on all three classes on a single run, we separately fine-tuned the model on each class. As a result, our model performed better when we separately fine-tuned model. We extracted 1000 samples of each class 3, 4 and 7 from the training dataset and saved in the separate files for later use. Now 57000 samples left in the training dataset that we saved in another file train_rem.pkl. Now load the training dataset from train_rem.pkl file also load the testing dataset. As we know train data contain 57000 samples and 10000 samples are in test data. After performing initial EDA, we define the Simple Neural Network (SimpleNN) model which we'll use for training the model. Dataset is already splitted in training and testing. Then we use the Cross Entropy Loss function and Adam optimizer to minimize the loss function. Here is the mathematical representation of cross entropy loss.

$$H(p, q) = -\sum\_x \in \mathcal{X} p(x) \log q(x). \tag{5}$$

We perform training on 57000 samples using 50 epochs with a batch size of 32 using SimpleNN model with 109386 parameters. We achieved 91.7% accuracy on testing data when we test the dataset. Now after evaluation performance metrices we load those files that we have saved earlier. After performing its EDA, we further train these classes over the pretrained model step by step. First, we train class 3 samples on the pretrained model using 10 epochs and evaluate its performance. Similarly, we train and evaluate class 4 sample and class 7 samples reactively. Basically we are performing finetuning After this we merge the all three classes samples together and then perform training using 10 epochs over the predefined model and then evaluate the performance.



IV. Evaluation

a) Positioning Figures and Tables: Place figures and tables at the top and bottom of columns. Avoid placing them in the middle of columns. Large figures and tables may span across both columns. Figure captions should be below the figures; table heads should appear above the tables. Insert figures and tables after they are cited in the text. Use the abbreviation

TABLE I
Performance Evaluation Table

Data	Accuracy (%)	Loss
Pre-trained model	92	0.38
Fine-tuned on combined tasks	70	2.1
Fine-tuned on individual tasks	88	0.60

V. Conclusion

In this paper we introduced the best practice to fine-tune the models. For Images dataset the model performed well following out technique. Fine-tune the model individually on each task results in better accuracy as compared to fine-tune on combined tasks. This technique is simple but has a huge impact in terms of transfer learning. We will further test it on other datasets for generalization and then move to next step which is to edit models through task arithmetic (Addition or Subtraction).

References

[1] Gabriel Ilharco, Marco Tulio Ribeiro, Mitchell Wortsman, Suchin Gururangan, Ludwig Schmidt, Hannaneh Hajishirzi, Ali Farhadi, EDITING MODELS WITH TASK ARITHMETIC, 2024.
[2] Rishabh Bhardwaj, Do Duc Anh, and Soujanya Poria, Language Models are Homer Simpson! Safety Re-Alignment of Fine-tuned Language Models through Task Arithmetic, 2024.
[3] Guillermo Ortiz-Jimenez and Alessandro Favero,Task Arithmetic in the Tangent Space:Improved Editing of Pre-Trained Models, 2024.