**Report: UnoLoRA and UnoLoRA⋆ for Efficient Multi-Task Learning**

This report details **UnoLoRA** and its enhanced version, **UnoLoRA⋆**, which are novel techniques for efficiently adapting large language models to perform well on multiple tasks simultaneously. These methods build upon the established **Low-Rank Adaptation (LoRA)** approach. We will explore how UnoLoRA and UnoLoRA⋆ differ from LoRA, the core techniques they utilize, and their performance in comparison to LoRA.

### **1. How UnoLoRA Differs from LoRA**

The fundamental distinction lies in how LoRA adapters are used in multi-task learning scenarios.

* **Traditional LoRA:** Typically employs **separate, task-specific LoRA adapters** for each individual task. This means if you want a model to perform 10 different tasks, you would train 10 separate sets of small, low-rank matrices (A and B).
* **UnoLoRA:** Introduces a significant change by utilizing **a single LoRA module that is shared across all the tasks**. Instead of having different sets of A and B matrices for each task, UnoLoRA uses just one set that is learned while the model trains on all the tasks together. The goal of this shared adapter is to capture both **general knowledge applicable to all tasks** and **specific nuances required for each individual task**.

### **2. Technique of UnoLoRA**

UnoLoRA's technique centers on the idea that the single shared LoRA module can learn adaptations that benefit multiple tasks due to LoRA's **implicit regularization properties**. **without adding extra constraints**. LoRA does this because it limits the changes to the model (since A and B are small and low-rank), which naturally keeps the model from overfitting. The paper's analysis reveals a natural division of labor within the shared LoRA adapter:

* **A Matrices tend to learn generalizable features** that are useful across different tasks. These matrices exhibit broader transformations in the model's parameter space.
* **B matrices tend to specialize in capturing task-specific representations**. They show more concentrated patterns, indicating they focus on particular adaptations needed for individual tasks.

By sharing a single LoRA module, UnoLoRA achieves **parameter efficiency** as only one set of low-rank matrices needs to be trained for all tasks, significantly reducing memory requirements. This approach also aims to mitigate **negative transfer** between tasks by learning task-agnostic adaptations.

**Parameter Efficiency**: Normally, if we train a separate LoRA module for each task, we end up using a lot of extra parameters. Instead, UnoLoRA shares a single LoRA module across all tasks, meaning we train fewer parameters overall. This reduces memory usage while still adapting to different tasks.  
**Negative Transfer Mitigation**: In multi-task learning, sometimes learning one task can hurt performance on another (like if learning Spanish grammar messes up your English grammar). UnoLoRA avoids this by ensuring that the shared LoRA module mostly learns **task-agnostic (general) adaptations**—patterns that help across tasks rather than conflicting ones.

A **task-agnostic adaptation** means learning **general skills** that are useful for multiple tasks, instead of learning something specific to only one task.

### **How This Applies to LoRA**

* Instead of learning something overly specific for each task, the **LoRA module focuses on patterns that help across different tasks**.
* This **prevents conflicts** between tasks (negative transfer) because the model doesn’t get overly specialized in one way that could harm another task.

### **3. UnoLoRA⋆: An Enhanced Variant**

**UnoLoRA⋆** builds upon the foundation of UnoLoRA to further improve task adaptation and training efficiency. It introduces a **shared hypernetwork**:

* **Shared Hypernetwork:** This component generates **task-specific embeddings** by considering the **task ID**, **sample encodings**, and **position information** within the network. Think of these embeddings as unique signatures for each task and each piece of data within a task.
  + What is the Shared Hypernetwork?
    - The shared hypernetwork in UnoLoRA⋆ is like a smart coordinator that helps the model understand how each task and data sample is different.The **hypernetwork** is a separate small neural network that sits **alongside the main model** and generates embeddings for different tasks and data points.The embeddings are **generated dynamically** during inference (when the model is making predictions) and training.
    - It does this by generating task-specific embeddings—which are like unique signatures for:

1. Which task is being performed (e.g., translating English to French vs. summarizing a document).
2. What kind of data is being processed (e.g., a short tweet vs. a long article).
3. Where the information is being used in the model (e.g., at the beginning vs. deeper in the network).

* **Modulating the A Matrix:** These task-specific embeddings are then used to create **scaling factors** that are applied to the **A matrix** of the shared LoRA module. The A matrix, as mentioned earlier, is responsible for capturing generalizable features. By modulating it with task-specific scaling, UnoLoRA⋆ allows the model to leverage these general features in a way that is tailored to each specific task. The adapted weight matrix in UnoLoRA⋆ is computed as: **W ′ = W + α ·B(A · diag(st))**, where *st* represents the task-specific scaling vector.

Each term in the formula has a role:

* **W** → The original model weights (before adaptation).
* **A** → The general transformation matrix that captures broad patterns.
* **B** → The task-specific transformation matrix that refines details.
* **st** → The **task-specific scaling vector** (this comes from the **shared hypernetwork**).
* **diag(st)** → Converts the vector **st** into a diagonal matrix, so each element can scale different parts of **A** independently.
* **α** → A scaling factor that controls how much influence the adaptation has.

The inclusion of the shared hypernetwork in UnoLoRA⋆ leads to **improved convergence** during training and better adaptation to individual tasks.

### **4. Performance Comparison with LoRA**

UnoLoRA and UnoLoRA⋆ demonstrate compelling performance, especially in terms of **parameter efficiency**, compared to traditional LoRA in multi-task settings.

* **Parameter Efficiency:** Both UnoLoRA and UnoLoRA⋆ achieve high parameter efficiency by training only about **0.05% of the total parameters per task**. In contrast, while single-task LoRA is also efficient (0.4% trainable parameters in the experiments), using separate LoRA adapters for each task in a multi-task scenario would lead to a linear increase in the total number of trainable parameters. UnoLoRA's shared approach significantly reduces this overhead.

* **GLUE Benchmark Results:**
  + In **single-task training**, standard LoRA achieved an average score of **84.40%** on the GLUE benchmark, performing competitively with full fine-tuning (84.67%) while using significantly fewer parameters. This establishes LoRA as an effective parameter-efficient fine-tuning method.
  + In **multi-task training** on the GLUE benchmark:
    - **UnoLoRA⋆ achieved an average score of 84.95%**, outperforming the base **UnoLoRA (84.40%)**.
    - While **HyperFormer++** (another parameter-efficient multi-task learning technique) achieved a slightly higher average of **86.48%**, UnoLoRA and UnoLoRA⋆ did so with considerably fewer trainable parameters (around 0.05% compared to 0.290% for HyperFormer++).
    - Notably, UnoLoRA⋆ showed specific performance gains on tasks like CoLA, MRPC, and MNLI compared to UnoLoRA.
    - UnoLoRA achieved the best performance on the RTE task among the multi-task approaches evaluated.
  + **Convergence Speed:** **UnoLoRA⋆ demonstrated significantly faster convergence** than the base UnoLoRA across various GLUE tasks. This means UnoLoRA⋆ can reach a higher level of performance in fewer training steps, making it particularly advantageous in resource-limited situations.

### **5. Intuitive Explanation**

Think of adapting a large language model for different tasks like having a set of tools in a workshop.

* **LoRA:** Would be like giving each specific job (task) its own set of specialized small tools to adjust the main machinery.
* **UnoLoRA:** Is like having one single set of general-purpose small tools that you try to use for all the different jobs. This is more efficient because you have fewer tools to manage. It works because some basic adjustments are helpful for many tasks.
* **UnoLoRA⋆:** Is like having that same single set of general tools, but you also have a clever assistant (the hypernetwork) who figures out small, task-specific attachments or ways to use those general tools slightly differently for each job. This allows you to be efficient with your main tools while still being precise and effective for each task, and you might even get the job done faster.

## 

## 

## 

## **Report: MoRA: High-Rank Updating for Efficient Fine-Tuning**

This report explains **MoRA (High-Rank Updating)** and how it differs from the popular **LoRA (Low-Rank Adaptation)** method for efficiently fine-tuning large language models. We will also discuss the core techniques of MoRA and compare its performance to LoRA based on the provided source.

### **1. How MoRA Differs from LoRA**

The main difference between MoRA and LoRA lies in the **rank of the weight update** during fine-tuning.

* **LoRA utilizes low-rank matrices** to update the model's weights. This means the changes LoRA makes to the model are limited in their complexity, as they are formed by multiplying two smaller matrices. The **rank of this update is significantly smaller** than the full rank of the original weight matrices. For example, if LoRA uses a rank of 8, the update's complexity is capped at this low level.
* MoRA, on the other hand, employs a square matrix to achieve high-rank updating. By using a square matrix, MoRA aims to maximize the rank of the weight update while keeping the number of trainable parameters the same as LoRA. For instance, with the same number of parameters as a LoRA configuration with rank 8 and a hidden size of 4096, MoRA uses a 256x256 square matrix, allowing for a much higher rank update (up to 256). This enables MoRA to introduce more complex changes to the model compared to LoRA with the same parameter budget.  
  **What Does “Higher Rank Update” Mean?**

Rank in matrices represents how much independent information the matrix can encode.  
Higher rank = More flexibility to introduce complex transformations.

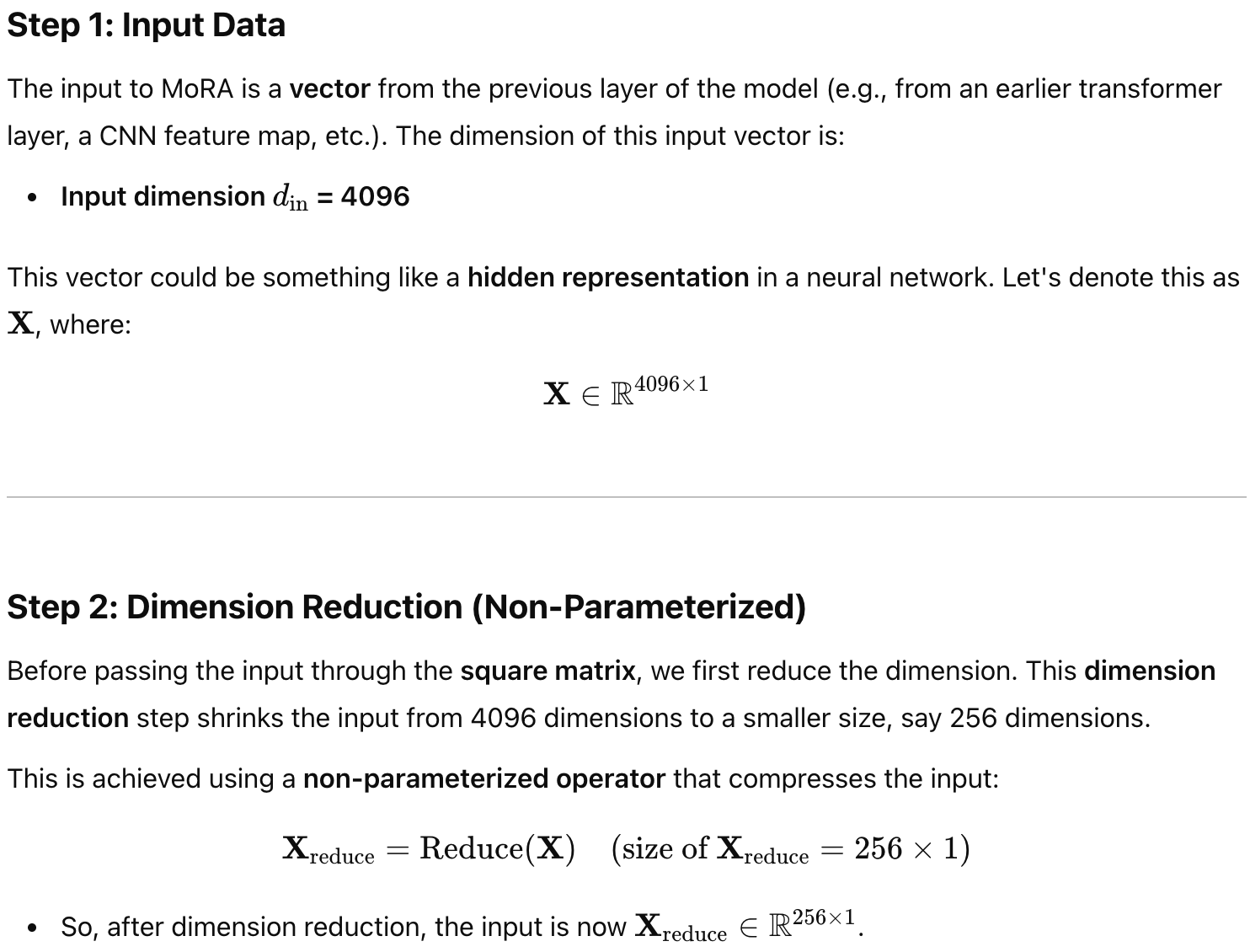
Essentially, **LoRA makes simpler, lower-dimensional adjustments to the model, while MoRA allows for more complex, higher-dimensional changes** using a similar number of trainable parameters.

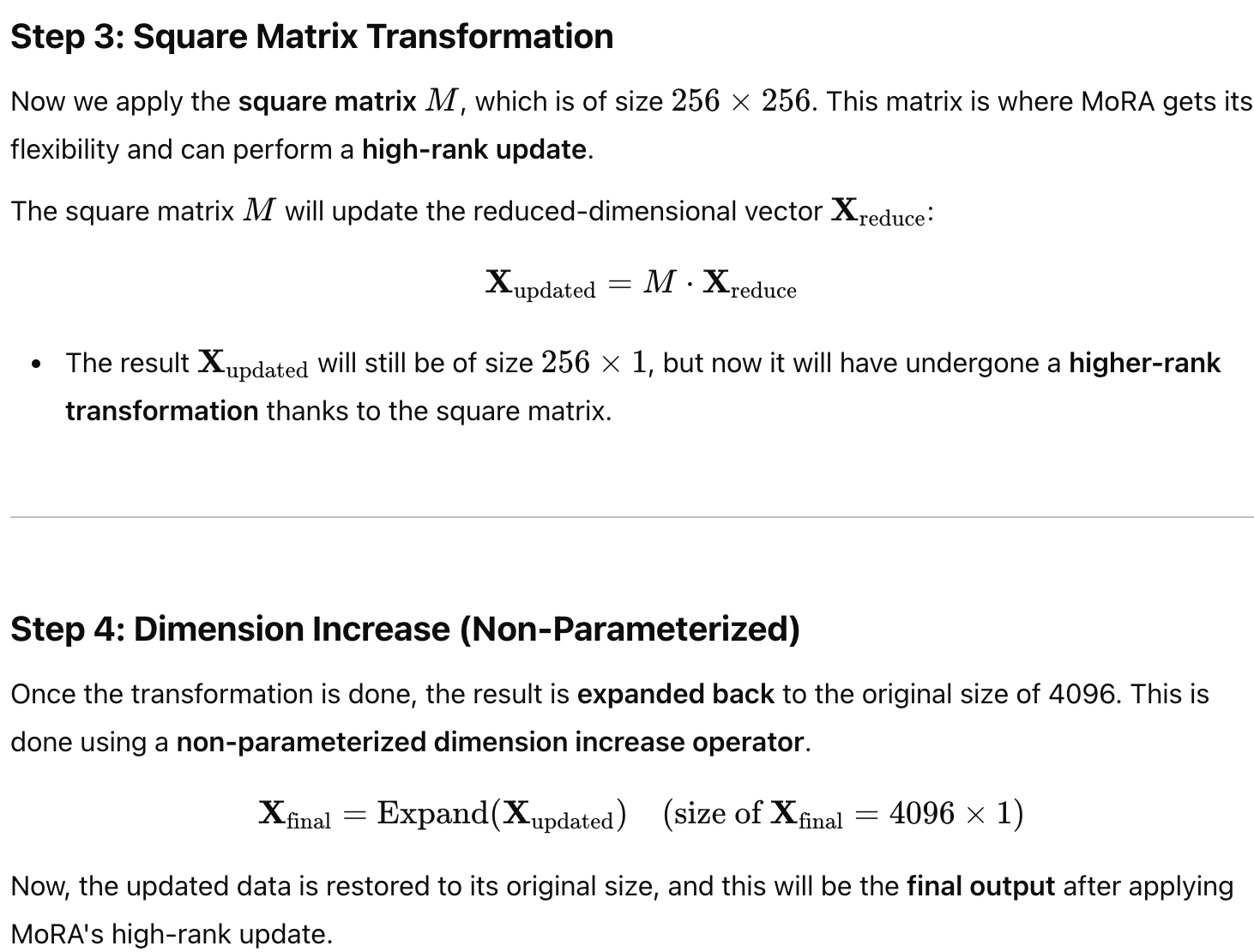
### **2. Technique of MoRA**

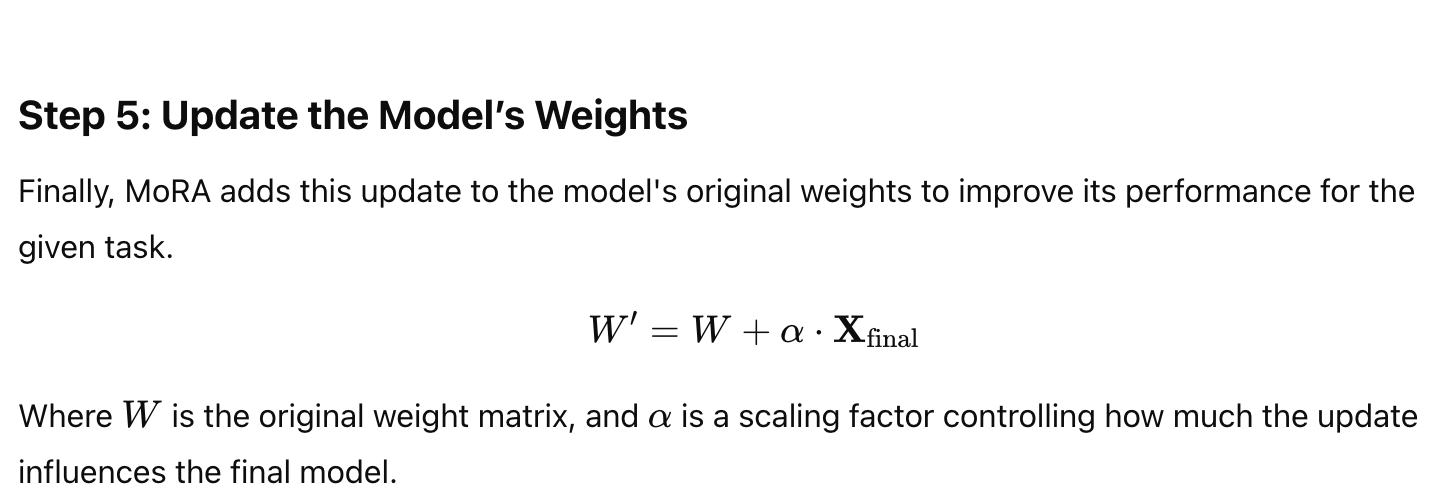
MoRA's technique revolves around achieving high-rank updates using a square matrix without increasing the number of trainable parameters compared to LoRA. To do this, MoRA introduces non-parameterized operators.  
**What are Non-Parameterized Operators?**

These are operations that do not have any learnable parameters. They are used to reshape or transform data in a way that allows MoRA to handle higher-rank updates while still keeping the number of parameters constant.

* **Dimension Reduction and Increase:** These operators are designed to **reduce the input dimension** for the square matrix and then **increase the output dimension** back to the original size. Think of it like squeezing the information into a higher-capacity square "bottleneck" and then expanding it back out. These operators themselves do not have any learnable parameters.
* **Mergeability:** Importantly, these operators are designed so that the resulting weight update can be **merged back into the original large language model**, just like LoRA. This means that during inference (using the fine-tuned model), there is **no additional computational cost** associated with MoRA compared to the original model.







The paper explores **four kinds of these non-parameter operators** for reducing and increasing dimensions:

* **Truncation:** Simply taking a portion of the input and padding with zeros later. This method is less effective due to information loss.
* **Sharing Rows and Columns:** Grouping dimensions together and summing them during compression, and then repeating or interleaving during decompression. This is more efficient for larger ranks.
* **Reshaping (Decouple):** Reshaping the input into a matrix and then concatenating it after multiplication with the square matrix.
* **Rotation:** Incorporating rotation operators inspired by RoPE to help the square matrix differentiate between different parts of the input. This method was found to be more efficient, especially for smaller ranks.

In essence, MoRA uses these clever, parameter-free tricks to allow a square matrix to perform a more complex update to the model's weights than LoRA, while maintaining parameter efficiency and deployability.

### **3. Performance Comparison with LoRA**

The paper presents a comprehensive evaluation of MoRA and LoRA across various tasks. Here's a summary of the performance comparison:

* **Memory-Intensive Tasks (Memorization and Continual Pretraining): MoRA significantly outperforms LoRA**.  
  + In a **memorization task** involving UUID pairs, MoRA required **fewer training steps** to memorize the information compared to LoRA, even with a large rank of 256 for LoRA. While increasing LoRA's rank helped, a gap still existed with FFT (Full Fine-Tuning). MoRA with rank 256 could achieve **similar performance to FFT** in memorizing all UUID pairs.
  + In **continual pretraining** on biomedical and financial domains, MoRA showed **better performance** than LoRA with the same number of trainable parameters. This highlights MoRA's advantage in tasks requiring the acquisition of new knowledge.
* **Instruction Tuning and Mathematical Reasoning: MoRA achieves comparable performance to LoRA**.  
  + 5, MoRA's performance was **on par with LoRA**. This aligns with the observation that LoRA can effectively leverage existing knowledge for format adaptation in instruction tuning.
  + For **mathematical reasoning** tasks (like GSM8K and MATH), MoRA also demonstrated **similar performance to LoRA**. Interestingly, while LoRA with a low rank (8) underperformed FFT on these tasks, increasing LoRA's rank to 256 could close this gap. MoRA also achieved comparable results at both ranks.
* **Pre Training from Scratch:** In pretraining experiments on the C4 dataset, **MoRA showed better performance (lower perplexity)** compared to LoRA with the same amount of trainable parameters. Furthermore, combining MoRA with a merge-and-reinitialize strategy (ReMoRA) yielded even greater improvements than doing the same with LoRA (ReLoRA). This again emphasizes the benefits of high-rank updating in learning new information during pretraining.

**In summary, MoRA demonstrates a clear advantage over LoRA in tasks that require memorizing new knowledge and enhancing domain-specific capabilities, while performing comparably in tasks that primarily involve leveraging existing knowledge and adapting to formats.** This suggests that the **high-rank updating mechanism in MoRA is more effective for knowledge acquisition** during fine-tuning and pretraining.

## **Report: ComLoRA: Enhancing LoRA through Competition**

This report explains **ComLoRA (Competitive Low-Rank Adaptation)** and how it differs from the standard **LoRA (Low-Rank Adaptation)** method for efficiently fine-tuning large language models. We will also discuss the core techniques of ComLoRA and compare its performance to LoRA based on the provided source.

### **1. How ComLoRA Differs from LoRA**

The main difference between ComLoRA and LoRA is that **ComLoRA introduces a competitive learning mechanism among multiple LoRA components**, while standard LoRA typically trains a single LoRA module.

* **Standard LoRA** involves adding a pair of low-rank matrices to certain layers of a large language model and only training these small matrices. The rank of these matrices determines the capacity of the adaptation. A low rank might not be expressive enough for complex tasks, while simply increasing the rank can lead to more parameters and a risk of overfitting.
* **ComLoRA, on the other hand, initializes multiple (K) distinct LoRA components**, each with a rank *r*. These different LoRA components are trained simultaneously and **compete with each other** during the fine-tuning process. This competition encourages each LoRA component to become better at the task. After training, the **best-performing LoRA component is selected** based on its performance on a validation set. Crucially, **only this winning LoRA component is used during inference**, meaning there's no extra computational cost compared to standard LoRA.

Think of it like having multiple specialists (LoRA components) trying to solve the same problem. They learn and improve by seeing how well they each perform relative to the others. In the end, you pick the best specialist to do the job.

### **2. Technique of ComLoRA**

ComLoRA works by incorporating **competitive learning** into the LoRA fine-tuning process. Here's a breakdown of the key steps:

* **Multiple LoRA Components:** ComLoRA starts by creating several independent LoRA modules (e.g., 2, 4, or more), each with a chosen low rank.
* **LoRA Selector:** A small, lightweight neural network called the **LoRA selector** is introduced. This selector looks at the input sequence and calculates a **similarity score** for each LoRA component. These scores indicate how suitable each LoRA component might be for the given input.
* **Different LoRA Components for Different Tasks**

Each **LoRA component** in **ComLoRA** is designed to **specialize** in certain aspects or features of the model. These components can be trained to perform **different tasks** or to handle **specific features** of the input data. For example:

* One LoRA component might specialize in **grammar correction**.
* Another might focus on **sentiment analysis**.
* Another could be trained to handle tasks like **named entity recognition (NER)**.

This specialization makes it possible to tailor each component to specific types of input or specific tasks, such as generating text, performing sentiment analysis, or answering questions.

* **The LoRA Selector Chooses the Best Component for the Task**

In **ComLoRA**, the **LoRA selector** is a key component that plays an important role in deciding which LoRA component to use for a given input:

* The **LoRA selector** calculates a **similarity score** for each LoRA component based on how well each component performs for the current task or input.
* Based on these similarity scores, the **LoRA selector** chooses the most suitable **LoRA component** for the task at hand.

This means that for different types of inputs (or tasks), **ComLoRA** can select **different LoRA components** that have been trained to handle different aspects of language or tasks. For example:

* If the input involves a lot of **sentiment-related information**, the LoRA component specialized in sentiment analysis may get a higher score and be selected.
* If the input involves **complex sentence structures**, the LoRA component specialized in grammar correction might be chosen.

### **Shared Model, Task-Specific Components**

While the **main model** remains **shared** across tasks (i.e., the base language model), the **LoRA components** are the ones that specialize in handling different aspects of the task. Therefore, **ComLoRA** allows for **task-specific specialization** through **LoRA components**, but the base model is the same across all tasks.

* **Training in ComLoRA**
  + **Each LoRA component** learns to handle a specific task or aspect of the data.
  + The **LoRA selector** is trained to give higher scores to the components that perform best for the current task or input. This encourages the **selector** to prefer the LoRA component that is more effective for the specific task at hand.
  + During training, **competition** arises between the components as the model **chooses the best LoRA component** for each task, guided by the **language modeling loss**, **pairwise loss**, and **alignment loss**.
* **Determining the Winner:** After the training is complete, the LoRA selector is used on a separate validation dataset to calculate the total similarity score for each LoRA component across all validation examples. The **LoRA component with the highest total similarity score is declared the "winner"**.
* **Inference:** When using the fine-tuned model for new tasks (inference), **only the winning LoRA component is used**. The other LoRA components and the selector are discarded. This is why ComLoRA doesn't add any extra computational overhead during inference compared to standard LoRA.

### **3. Performance Comparison with LoRA**

The experiments in the source demonstrate that **ComLoRA generally outperforms standard LoRA** on various natural language understanding tasks.

* **Commonsense Reasoning:** ComLoRA configurations often achieve higher average accuracy than various LoRA ranks, sometimes even surpassing a single LoRA with a significantly higher rank. For example, ComLoRA with lower rank components can outperform a LoRA with a higher rank, showing that the competitive learning is more effective than simply increasing the rank in standard LoRA.
* **MMLU (Massive Multitask Language Understanding):** ComLoRA consistently shows superior performance compared to different LoRA baseline configurations on this diverse set of tasks. The best-performing ComLoRA setup achieves a higher accuracy than the best LoRA configuration.
* **Personalized Conversation:** On the CONVAI2 dataset, ComLoRA outperforms LoRA across multiple evaluation metrics, indicating that it's better at capturing diverse conversational patterns and maintaining personalized dialogue.

**Key Performance Advantages of ComLoRA:**

* **Improved Performance:** ComLoRA can achieve better results than standard LoRA without increasing inference costs.
* **Enhanced Adaptability:** The competitive learning process allows ComLoRA to adapt more effectively to diverse tasks.
* **Parameter Efficiency:** ComLoRA can achieve strong performance with fewer parameters per component compared to a single high-rank LoRA.
* **No Inference Overhead:** Since only the winning LoRA is used during inference, ComLoRA maintains the efficiency of standard LoRA.

## **Report: Bayesian-LoRA: Efficient Fine-Tuning with Optimal Quantization and Rank**

This report explains **Bayesian-LoRA (B-LoRA)** and how it differs from the standard **LoRA (Low-Rank Adaptation)** method for efficiently fine-tuning large language models. We will also discuss the core techniques of B-LoRA and compare its performance to LoRA based on the provided source.

### **1. How Bayesian-LoRA Differs from LoRA**

The main difference between Bayesian-LoRA and standard LoRA lies in its ability to **automatically determine the best level of precision (quantization) and the right size (rank) for the added low-rank matrices**, whereas standard LoRA typically uses a fixed rank and does not incorporate quantization.

* **Standard LoRA** works by taking the weight changes in a pre-trained model during fine-tuning and approximating them with two smaller (low-rank) matrices. You choose a rank, and the same rank is usually applied to all these added matrices. Finding the best rank often requires trying different values and retraining, which can be time-consuming. Also, standard LoRA doesn't focus on reducing the number of bits used to represent the model's weights (quantization).
* **Bayesian-LoRA (B-LoRA)** takes a different approach by thinking about both the precision of the weights and the size of the low-rank matrices from a **Bayesian perspective**. This means it puts a "prior belief" on the possible quantization levels and rank values. During fine-tuning, B-LoRA **learns the optimal precision (how many bits to use) and the optimal rank for each individual low-rank matrix**. It does this using special "gates" that can decide whether to use more or fewer bits and a larger or smaller rank. This allows different parts of the model to have different levels of precision and importance (rank), which can be more efficient than using the same settings everywhere. B-LoRA also aims to reduce the energy consumption of the model by using quantization techniques.

Think of standard LoRA as using the same set of tools (fixed rank) for every task. Bayesian-LoRA, on the other hand, can choose the best tool (optimal rank and precision) for each specific part of the task, potentially leading to better results and efficiency.

### **2. Technique of Bayesian-LoRA**

Bayesian-LoRA works by using **Bayesian principles to control both quantization and rank adaptation** of the low-rank matrices added by LoRA. Here's a simplified breakdown of the key techniques:

* **Learnable Quantization:** B-LoRA uses a technique inspired by "BayesianBits". It starts with the full-precision weights and can gradually reduce the number of bits used to represent them (e.g., from 32 bits down to 16, 8, 4, or even 2 bits). It introduces "gating variables" that control the precision level. By putting a prior distribution on these gates and learning their optimal values during training, B-LoRA determines the best precision for each weight. This helps in reducing computational cost and energy consumption.The prior distribution can be seen as a kind of **soft constraint** that helps the model decide which weights should be represented in higher precision and which can be approximated with lower precision.
* **Prior distribution**: This prior is a statistical concept that encodes the belief or assumption about the precision before training begins.
* **Bayesian Rank Adaptation:** B-LoRA also optimizes the rank of each low-rank matrix. Similar to quantization, it uses "gating variables" and prior distributions to control the rank. During training, B-LoRA learns which ranks are most important for different parts of the model. This allows the model to allocate more capacity (higher rank) to the more crucial weight updates and less to others, making the fine-tuning more efficient.
* **Joint Optimization:** The key innovation of B-LoRA is that it **simultaneously learns both the optimal quantization levels and the optimal rank values** for each low-rank matrix. This joint optimization allows for a more holistic approach to parameter-efficient fine-tuning, where the precision and importance of each weight update are considered together.
* **Training Process:** B-LoRA is trained using a loss function that encourages good performance on the downstream task while also pushing the gating variables towards more efficient quantization and rank values. It uses a technique called the "straight-through estimator" to handle the discrete nature of quantization and rank decisions during the training process.

In essence, Bayesian-LoRA intelligently compresses the LoRA adapters during fine-tuning by learning the best way to represent them (quantization) and their importance (rank) for the specific task.

### **3. Performance Comparison with LoRA**

The source indicates that **Bayesian-LoRA performs competitively with or even better than standard LoRA** and other state-of-the-art parameter-efficient fine-tuning methods on several benchmark datasets.

* **GLUE Benchmark:** On the GLUE benchmark, B-LoRA achieves **on-par performance with baselines like LoRA, DyLoRA, and AdaLoRA** across various natural language understanding tasks. Notably, it shows **better results on datasets like SST-2 and RTE** compared to some baselines. The results also show that **optimizing both quantization and rank leads to better performance** on some datasets compared to using only adaptive quantization.
* **MMLU Benchmark:** When fine-tuning larger models like Phi-2 and Qwen2 on the MMLU benchmark, B-LoRA also shows promising results. B-LoRA with rank adaptation performs **on par with QLoRA**. Interestingly, on these larger models and this benchmark, rank adaptation without quantization performs better than with quantization, a trend not observed on the GLUE benchmark.
* **Efficiency Gains:** A significant advantage of B-LoRA is its **reduction in the total number of bit operations (BOPs) by roughly 70%** compared to baseline methods. BOPs are a measure of computational complexity and can be related to energy consumption. This means B-LoRA can achieve comparable or better performance with significantly less computational cost.
* **Qualitative Analysis:** The analysis of quantization levels and rank values learned by B-LoRA shows that it can **adapt these parameters based on the task and the specific weight matrices**. For example, attention values (Wv) tend to retain more information with larger rank values, while keys (Wk) and queries (Wq) can often use lower ranks. The precision levels also seem to correlate with the size of the training data.

In conclusion, Bayesian-LoRA offers a way to **achieve strong performance similar to or better than LoRA while being significantly more computationally efficient** by automatically learning optimal quantization levels and rank values for each part of the low-rank adaptation. This makes it a promising approach for energy-efficient and scalable model fine-tuning.

| **Variant** | **Key Idea** | **Rank Update Type** | **Sharing / Modulation** | **Task Adaptation** | **Computational Efficiency** | **Unique Features** |
| --- | --- | --- | --- | --- | --- | --- |
| **LoRA** | Fine-tune LLMs with low-rank updates (A·B matrices) | Low-rank | Per-task (one LoRA per task) | Task-specific | High (0.1%–0.4% params) | Simple, effective PEFT method |
| **UnoLoRA** | One shared LoRA for all tasks | Low-rank | Shared A, B across all tasks | General + specific via A/B roles | Higher than LoRA in multi-task | Reduces neg. transfer; A learns general, B learns specific |
| **UnoLoRA★** | UnoLoRA + task-specific modulation via hypernetwork | Low-rank | Shared LoRA + hypernet scaling of A | Task-specific embeddings from task ID/sample/position | Highest among LoRA family | Fast convergence; dynamic task-aware adaptation |
| **MoRA** | High-rank updates using square matrices | High-rank | Same across tasks | General or specific | Same parameter count as LoRA, more expressive | Uses parameter-free operators (reshape, rotation) |
| **ComLoRA** | Competitive LoRA modules | Low-rank | Multiple LoRAs compete during training | Specialized per LoRA | Efficient at inference (only 1 LoRA kept) | Selector chooses best LoRA per task; better generalization |
| **Bayesian-LoRA** | Bayesian optimization of rank & precision | Adaptive rank & quantization | Per module | Per-matrix optimized | Very efficient (70% less BOPs) | Learns optimal rank and bit-width jointly |

1. **LoRA** – baseline →
2. ➜ **UnoLoRA** (single shared LoRA across tasks) →
3. ➜ **UnoLoRA★** (adds dynamic hypernetwork for modulation)

Parallel developments focusing on specialization, expressiveness, or compression:

* **MoRA** (more expressive updates via high-rank square matrices)
* **ComLoRA** (multiple competing LoRA modules)
* **Bayesian-LoRA** (adaptively tunes rank and quantization)