DoRa: Weight-Decomposed Low-Rank Adaptation

TELLEZ Alba

July 12, 2024

1 Introduction

A pretrained matrix $W \in \mathbb{R}^{d \times d}$ from a model like LLaMA 2, trained by Meta on extensive internet data, represents broad global knowledge. However, it lacks specific details for fine-grained tasks. To adapt it for a particular purpose, such as a specialized chatbot or a classification task, this matrix needs to be fine-tuned, extracting relevant information to enhance performance in the desired area.

1.1 Attention mechanism

The attention mechanism allows a model to focus on specific parts of the input sequence that are more relevant for making predictions. This is particularly useful in tasks like natural language processing where the relationship between words in a sentence can be complex.

The attention mechanism involves the following key components (see Fig.1):

- **Input Matrix (X)**: The input data, represented as a matrix of shape $s \times d$, where s is the number of tokens (e.g., words) and d is the dimensionality of each token.
- **Queries (Q)**: These are the transformed representations of the input data used to query the relevant information.
- **Keys (K)**: These represent the input data and are used to match against the queries.
- **Values (V)**: These are the actual data that we want to focus on and are weighted according to the relevance determined by the queries and keys.

By multiplying the input matrix X with the respective weight matrices $W_Q, W_K, W_V \in \mathbb{R}^{d \times d}$associated to X, K, V $(\in \mathbb{R}^{s \times d})$, we get:

$$Q = X \cdot W_Q \tag{1}$$

$$K = X \cdot W_K \tag{2}$$

$$V = X \cdot W_V \tag{3}$$

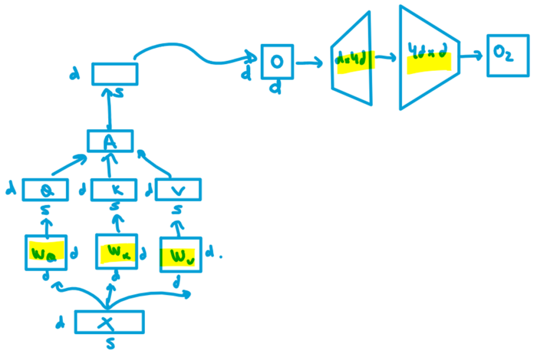


Figure 1: Attention Mechanisim

1.1.1 Calculating Attention

The attention mechanism involves computing a score to determine the importance of each value in the sequence. This is typically done using the dot product of the queries and keys. These scores are then normalized (usually with a softmax function) to get attention weights:

$$\text{Attention Weights} = \text{softmax}(Q \cdot K^T) \tag{4}$$

The final step is to use these weights to get a weighted sum of the values:

$$\text{Output} = \text{Attention Weights} \cdot V \tag{5}$$

The resulting output matrix has the shape $s \times d$.

1.1.2 Output Matrix (O)

After computing the attention, the output is passed through another weight matrix W_O to transform it back to the original dimensionality:

$$\text{Final Output} = \text{Output} \cdot W_O \tag{6}$$

where W_O has a shape of $d \times d$.

1.1.3 Feedforward Layers

The output of the attention mechanism is then passed through feedforward layers to further process the information. Typically, this involves two transformations:

- A linear transformation with a weight matrix of shape $d \times 4d$.
- Another linear transformation with a weight matrix of shape $4d \times d$.

These transformations help in refining the output and making it suitable for the next layer in the model.

2 LoRa: Low-Rank Adaptation of Large Language Models

Direct fine-tuning of large models demands significant memory. For example, if using the Adam optimizer $d \times d$ requires storing three copies of this matrix W, which is not efficient. For a model like LLaMA 7B (with 7 billion parameters), this necessitates around 21 billion parameters, translating to approximately 21 GB of RAM.

LoRA (Low-Rank Adaptation) is a technique designed to reduce the memory and computational requirements by approximating those weight matrices with lower-rank matrices. This makes it easier to fine-tune the model with limited resources, making fine-tuning feasible on consumer-grade hardware. For solving these memory problem, the authors of LoRa propose decomposing the weight matrix into two low-rank matrix $A \in \mathbb{R}^{d \times r}$ and $B \in \mathbb{R}^{r \times d}$.

LoRA is applied to the feedforward layers, which operate token-wise. The process can be visualized with the diagram Fig.2.

The output O is computed as follows:

$$O = Wx + ABx \tag{7}$$

Here, W is the original weight matrix, representing the pre-trained model's knowledge. A A is initialized from a normal distribution with zero mean and some standard deviation, while B is initialized as a zero matrix. Remember the goal is to fine-tune our model and we basically add these both A and B adapters like in Fig. to every weight Matrix we have, you add it to all your feed forward layers. The idea involves training layers A and B while keeping matrix X frozen. The memory requirement for this setup is $d^2 + 2dr$ parameters. Although this is more than the original model, storing moments in the original model requires $3 \cdot d^2$, as two moments of the weights need to be stored. In this approach, you store one weight matrix and three moments, resulting in $3 \cdot 2dr$. This

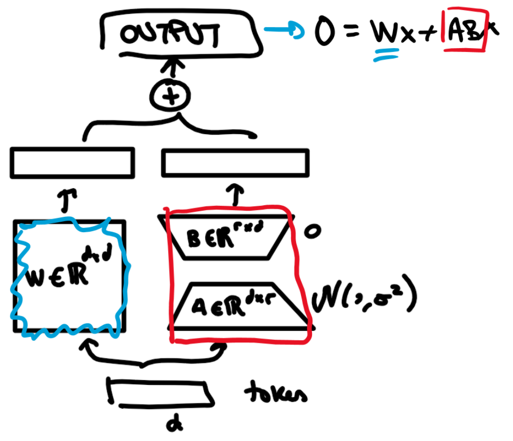

OUTPUT
$O = Wx + \boxed{AB}x$
$W \in \mathbb{R}^{d \times d}$
$B \in \mathbb{R}^{r \times d}$
0
$A \in \mathbb{R}^{d \times r}$
$\mathcal{N}(0, \sigma^2)$
token
d

Figure 2: LoRa Scheme: The input token, represented as a d-dimensional vector, is passed through both low-rank matrices A and B, producing two output vectors that are then summed to give the final output vector.

is significantly smaller if r is small. You can choose r to be 1, 4, or 32, which is much smaller than d (often 1024). Thus, the overall memory footprint is reduced.

The number of trainable parameters is much smaller, typically around 1% of the total parameters. This means only small weights are stored and trained, leading to a small memory footprint and allowing efficient training. The total parameter count will be around 7 billion plus a few more, potentially reaching up to 8 billion in the worst case with a large r. However, these matrices are generally very small.

Once you're done fine-tuning your A and B matrices, you can combine them with W to form:

$$O = Wx + ABx = [W + AB]\, X = W'X \tag{8}$$

This combined matrix can be stored in memory or on disk with the same memory footprint as before, facilitating efficient inference.

3 Weight Decomposition Analysis

The key idea extends this by decomposing the original weight matrix into direction and magnitude (vectors are defined by these). The weight matrix W can be expressed as:

$$W = \|W\|_c \cdot \frac{W}{\|W\|_c} \tag{9}$$

where M is the magnitude matrix and the second is the matrix of unit/normalized vectors (direction).

As you fine-tune the model with low-rank adaptation (LoRA), the updates in magnitude and direction can be tracked. In summary, the process involves:

- Fine-tuning A and B while keeping X frozen.
- Combining A and B with W to form $W' = W + ABX$.
- Decomposing W into magnitude and direction: $W = M \cdot U$.

This decomposition allows efficient fine-tuning and inference with a reduced memory footprint, maintaining the original model's efficiency.

Now, we can try to understand the impact of fine-tuning on M and U. The average change in magnitude so your weight Matrix is defined by a set of vectors:

$$\Delta M_{\text{FT}}^{t} = \frac{\sum_{n=1}^{k} |m_{\text{FT}}^{n,t} - m_0^n|}{k}$$

$$\Delta D_{\text{FT}}^{t} = \frac{\sum_{n=1}^{k} (1 - \mathbf{cos}(V_{\text{FT}}^{n,t}, W_0^n))}{k}$$

The process involves tracking changes in the magnitude and direction of weight vectors during fine-tuning. The average magnitude of the weight matrix at time t is compared to the original magnitude to measure deviation. Directional changes are quantified using cosine similarity. Initially, vectors are aligned (cosine similarity = 1), but as fine-tuning progresses, vectors may diverge, indicated by a decrease in cosine similarity. This method captures the evolving distance and orientation of weight vectors relative to their initial states, offering insights into the fine-tuning dynamics and their effects on the model's parameters.

In their analysis, the authors observe that fine-tuning models exhibit a decreasing slope when plotting change in direction versus change in magnitude. They note an inverse relationship: as the direction change increases, the magnitude decreases. They also find high variance across direction changes and magnitudes. Specifically, with fine-tuning, an increase in direction change correlates with a decrease in magnitude. However, LoRA weights show an increase in magnitude with increased direction change, exhibiting low variance (see Fig.3).

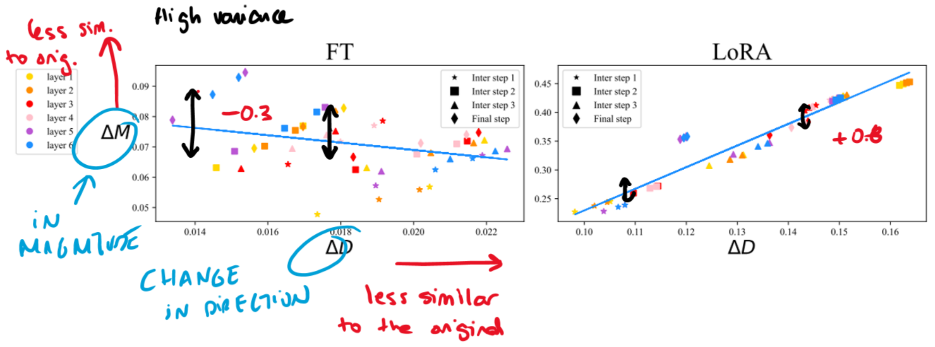


Figure 3: Changes in Magnitude Vs. Direction in FT (High variance, negative slope) versus LoRa (Low variance, positive slope)

This discrepancy suggests that LoRA does not mimic fine-tuning accurately. To address this, they introduce Dora, which aims to better replicate fine-tuning by independently manipulating direction and magnitude.

4 DoRa:Weight-Decomposed Low-Rank Adaptation

By decomposing the weight matrix into magnitude and direction, Dora trains these components separately: the magnitude is directly updated, while the direction is adjusted using LoRA. This approach maintains efficiency and requires minimal additional parameters.

The weight follows now the following equation: Where the magnitude is multiplyied my the LoRa

$$W' = \underline{m}\frac{V + \Delta V}{||V + \Delta V||_c} = \underline{m}\frac{W_0 + \underline{BA}}{||W_0 + \underline{BA}||_c}$$

and the normalization. This is the gradient and in order to avoid saving those parameters, we let it as a constant value (SG).

The DoRa Method SCheme:

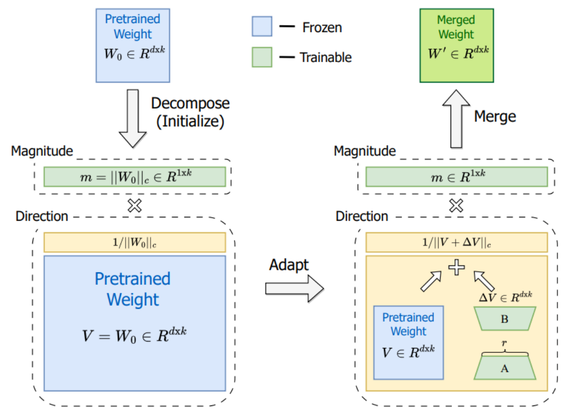


Figure 4: DoRa Scheme: An overview of DoRA, which decomposes the pre-trained weight into magnitude and direction components for fine-tuning, especially with LoRA to efficiently update the direction component.

Empirical results demonstrate that Dora achieves a decreasing slope with higher variance, similar to fine-tuning (see Fig.4), and outperforms LoRA across various benchmarks, particularly with smaller rank r. This method effectively captures the desired inverse relationship between direction and magnitude changes, leading to improved performance with a memory-efficient model.

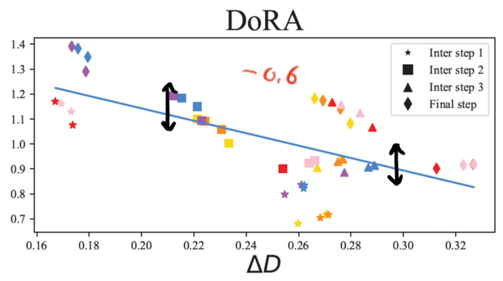


Figure 5: Changes in Magnitude Vs. Direction in DoRa (High variance, negative slope)

Table 1 shows a comparison of the average accuracy between LoRA and DoRA method across various rank settings for commonsense reasoning tasks. DoRA consistently outperforms LoRA at all rank settings, with the performance gap widening as the rank decreases. This suggests that our method effectively enhances the learning capacity of LoRA, enabling it to achieve better accuracy with fewer trainable parameters. Similar or better performance respect to LoRa with almost same number of

PEFT Method	rank r	# Params (%)	BoolQ	PIQA	SIQA	HellaSwag	WinoGrande	ARC-e	ARC-c	OBQA	Avg.
LoRA	4	0.10	$\boxed{2.3}$	46.1	18.3	19.7	55.2	65.4	51.9	57	39.5
	8	0.21	31.3	57.0	44.0	11.8	43.3	45.7	39.2	53.8	40.7
	16	0.42	69.9	77.8	75.1	72.1	55.8	77.1	62.2	78.0	70.9
	32	0.83	68.9	80.7	77.4	78.1	78.8	77.8	61.3	74.8	74.7
	64	1.64	66.7	79.1	75.7	17.6	78.8	73.3	59.6	75.2	65.8
DoRA (Ours)	4	0.11	$\boxed{51.3}$	42.2	77.8	25.4	78.8	78.7	62.5	78.6	61.9
	8	0.22	69.9	81.8	79.7	85.2	80.1	81.5	65.7	79.8	77.9
	16	0.43	70.0	82.6	79.7	83.2	80.6	80.6	65.4	77.6	77.5
	32	0.84	68.5	82.9	79.6	84.8	80.8	81.4	65.8	81.0	78.1
	64	1.65	69.9	81.4	79.1	40.7	80.0	80.9	65.5	79.4	72.1

Figure 6: Accuracy comparison of LoRA and DoRA with varying ranks for LLaMA-7B on the commonsense reasoning tasks.

parameters (DoRa a bit more due to saving magnitude). Only in the yellow colors, LoRa outperforms DoRa. Also, see the performance for low r, DoRa is better.

Robustness of DoRA towards different rank settings (DoRa is more robust to different raks (specially lower ones):

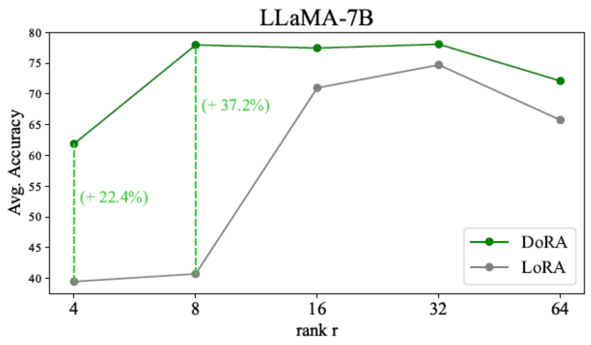
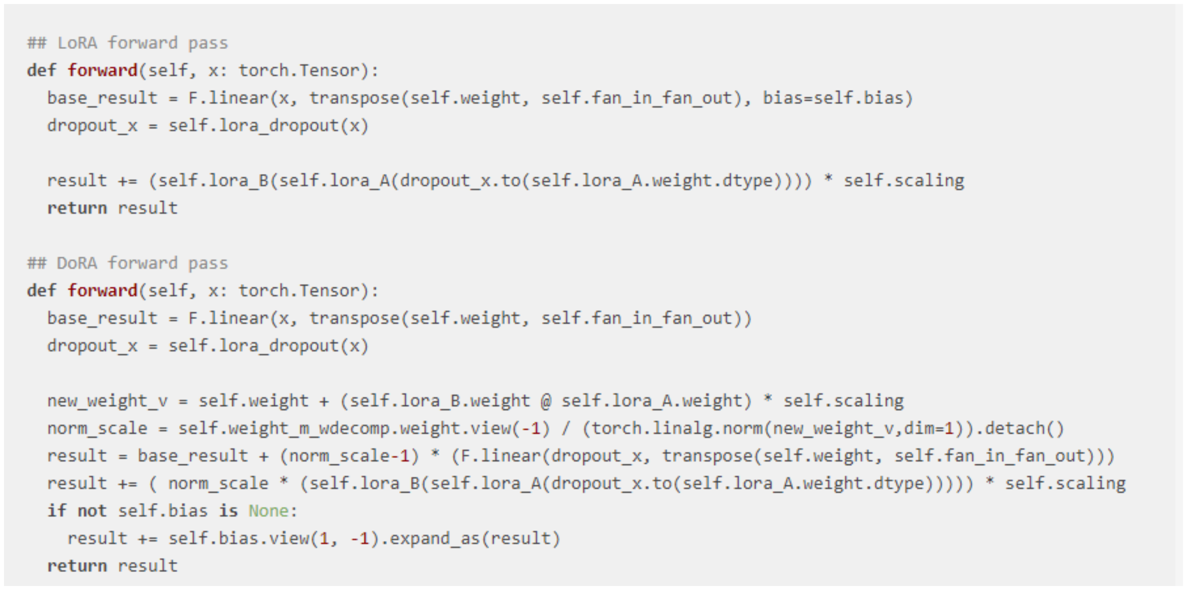


Figure 7: Average accuracy of LoRA and DoRA for varying ranks for LLaMA-7B on the commonsense reasoning tasks.

Implementation of DoRa compared to LoRa in PyTorch-like code:



```
## LoRA forward pass
def forward(self, x: torch.Tensor):
  base_result = F.linear(x, transpose(self.weight, self.fan_in_fan_out), bias=self.bias)
  dropout_x = self.lora_dropout(x)

  result += (self.lora_B(self.lora_A(dropout_x.to(self.lora_A.weight.dtype)))) * self.scaling
  return result

## DoRA forward pass
def forward(self, x: torch.Tensor):
  base_result = F.linear(x, transpose(self.weight, self.fan_in_fan_out))
  dropout_x = self.lora_dropout(x)

  new_weight_v = self.weight + (self.lora_B.weight @ self.lora_A.weight) * self.scaling
  norm_scale = self.weight_m_wdecomp.weight.view(-1) / (torch.linalg.norm(new_weight_v,dim=1)).detach()
  result = base_result + (norm_scale-1) * (F.linear(dropout_x, transpose(self.weight, self.fan_in_fan_out)))
  result += ( norm_scale * (self.lora_B(self.lora_A(dropout_x.to(self.lora_A.weight.dtype))))) * self.scaling
  if not self.bias is None:
    result += self.bias.view(1, -1).expand_as(result)
  return result
```

5 Conclusion

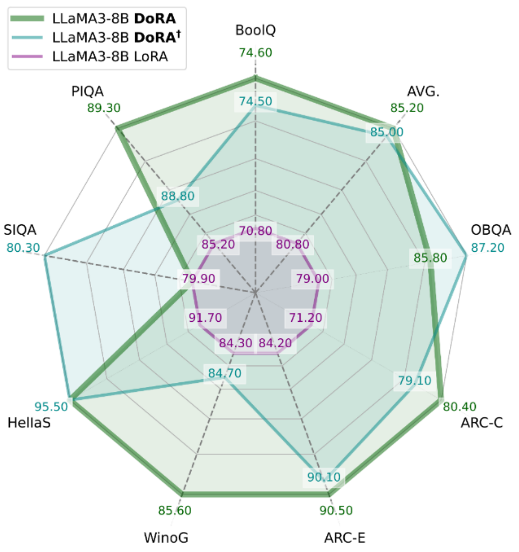


Figure 8: DoRA consistently outperforms LoRA on the LLaMA family models for commonsense reasoning tasks.

- **Adaptation Focus**
  - **LORA**: Focuses on adapting low-dimensional parameters of a pre-trained model, facilitating rapid specialization to new tasks.
  - **DORA**: Focuses on dynamic and continuous adaptation of the model based on specific context and input, optimizing its response in real-time.
- **Efficiency vs. Precision**
  - **LORA**: Prioritizes efficiency in terms of memory and computation, suitable for scenarios with limited resources.
  - **DORA**: Prioritizes precision and continuous optimization, ideal for scenarios requiring constant and precise adaptation.
- **Applications**
  - **LORA**: Suitable for quickly and efficiently adapting pre-trained models to new tasks.
  - **DORA**: Ideal for tasks with high variability and dynamism, requiring adaptive and optimized real-time responses.

References

[1] Aburass, S., [other authors' names if available] (2023). *An Ensemble Approach to Question Classification: Integrating Electra Transformer, GloVe, and LSTM.*