

Improving LoRA: Implementing Weight-Decomposed Low-Rank Adaptation (DoRA) from Scratch



SEBASTIAN RASCHKA, PHD

FEB 19, 2024

183

46

17

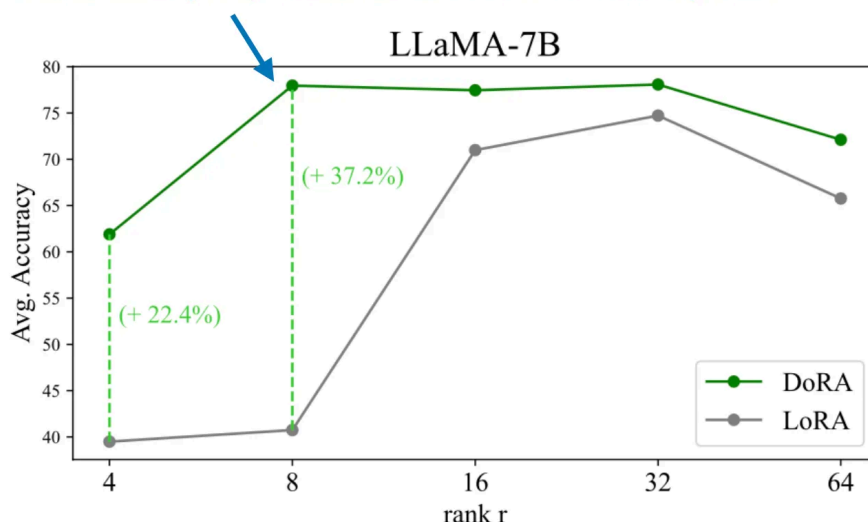
SI

Low-rank adaptation (LoRA) is a machine learning technique that modifies a pretrained model (for example, an LLM or vision transformer) to better suit a specific, often smaller, dataset by adjusting only a small, low-rank subset of model's parameters.

This approach is important because it allows for efficient finetuning of large models on task-specific data, significantly reducing the computational cost and time required for finetuning.

Last week, researchers proposed [DoRA: Weight-Decomposed Low-Rank Adaptation](#), a new alternative to LoRA, which may outperform LoRA by a small margin.

DoRA achieves good performance even if the rank is relatively small



DoRA is a promising alternative to standard LoRA (annotated figure from the DoRA paper: <https://arxiv.org/abs/2402.09353>)

To understand how these methods work, we will implement both LoRA and DoRA in PyTorch from scratch in this article!


LoRA Recap

Before we dive into DoRA, here's a brief recap of how [LoRA](#) works.

Since LLMs are large, updating all model weights during training can be expensive due to GPU memory limitations. Suppose we have a large weight matrix \mathbf{W} for a given layer. During backpropagation, we learn a $\Delta\mathbf{W}$ matrix, which contains information on how much we want to update the original weights to minimize the loss function during training.

In regular training and finetuning, the weight update is defined as follows:

$$\mathbf{W}_{updated} = \mathbf{W} + \Delta\mathbf{W}$$

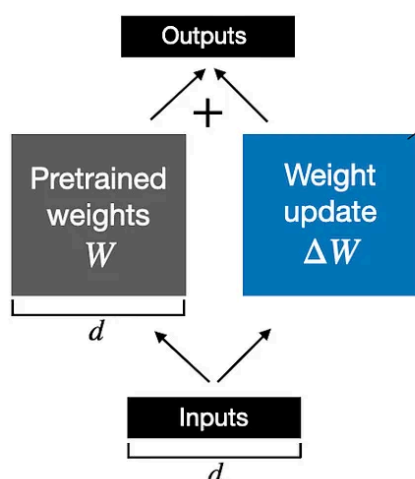
The LoRA method proposed by [Hu et al.](#) offers a more efficient alternative to computing the weight updates $\Delta\mathbf{W}$ by learning an approximation of it,  $\mathbf{A}\mathbf{B}$. In other words, in LoRA, we have the following, where \mathbf{A} and \mathbf{B} are two small weight matrices:

$$\mathbf{W}_{updated} = \mathbf{W} + \mathbf{A}\mathbf{B}$$

(The "." in " $\mathbf{A}\mathbf{B}$ " stands for matrix multiplication.)

The figure below illustrates these formulas for full finetuning and LoRA side side.

Weight update in regular finetuning



Weight update in LoRA

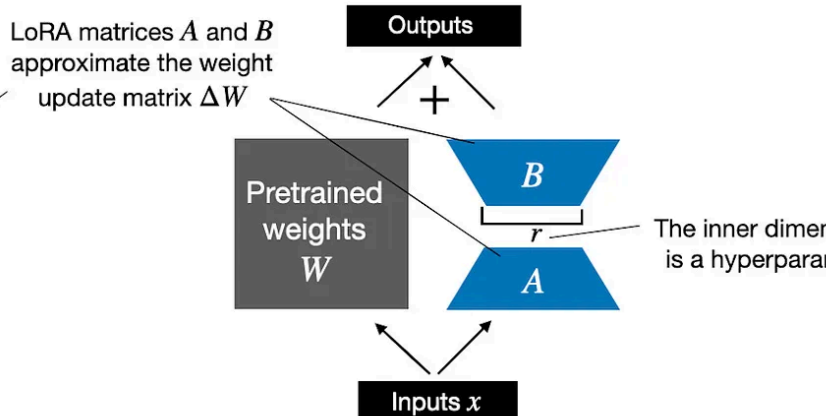


Figure: An illustration of regular finetuning (left) and LoRA finetuning (right).

How does LoRA save GPU memory? If a pretrained weight matrix W is a $1,000 \times 1,000$ matrix, then the weight update matrix ΔW in regular finetuning is a $1,000 \times 1,000$ matrix as well. In this case, ΔW has 1,000,000 parameters. If we consider a LoRA rank of 2, then A is a 1000×2 matrix, and B is a 2×1000 matrix, and we only have $2 \times 2 \times 1,000 = 4,000$ parameters that we need to update when using LoRA. In the previous example, with a rank of 2, that's 250 times fewer parameters.



Of course, A and B can't capture all the information that ΔW could capture, this is by design. When using LoRA, we hypothesize that the model requires to be a large matrix with full rank to capture all the knowledge in the pretraining dataset. However, when we finetune an LLM, we don't need to update all the weights and capture the core information for the adaptation with a smaller number of weights than ΔW would; hence, we have the low-rank updates via AB .

If you paid close attention, the full finetuning and LoRA depictions in the figure above look slightly different from the formulas I have shown earlier. That's due to the distributive law of matrix multiplication: we don't have to add the weights with the updated weights but can keep them separate. For instance, if x is the input data, then we can write the following for regular finetuning:

$$\mathbf{x} \cdot (\mathbf{W} + \Delta \mathbf{W}) = \mathbf{x} \cdot \mathbf{W} + \mathbf{x} \cdot \Delta \mathbf{W}$$

Similarly, we can write the following for LoRA:

$$\mathbf{x} \cdot (\mathbf{W} + \mathbf{A} \cdot \mathbf{B}) = \mathbf{x} \cdot \mathbf{W} + \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{B}$$

The fact that we can keep the LoRA weight matrices separate makes LoRA especially attractive. In practice, this means that we don't have to modify the weights of the pretrained model at all, as we can apply the LoRA matrices on the fly. This is especially useful if you are considering hosting a model for multiple customers. Instead of having to save the large updated models for each customer, you only have to save a small set of LoRA weights alongside the original pretrained model.

To make this less abstract and to provide additional intuition, we will implement LoRA in code from scratch in the next section.

A LoRA Layer Code Implementation

We begin by initializing a `LoRALayer` that creates the matrices \mathbf{A} and \mathbf{B} , along with the alpha scaling hyperparameter and the rank hyperparameters. This layer can accept an input and compute the corresponding output, as illustrated in the figure below.

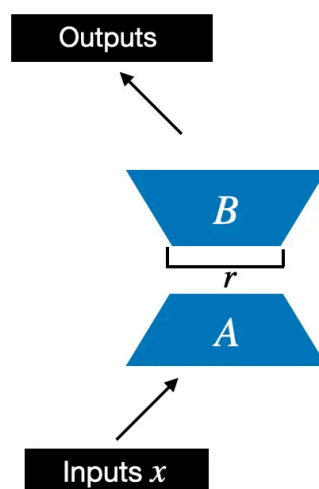


Illustration of the LoRA matrices \mathbf{A} and \mathbf{B} with rank r .

In code, this LoRA layer depicted in the figure above looks like as follows:

```
import torch.nn as nn

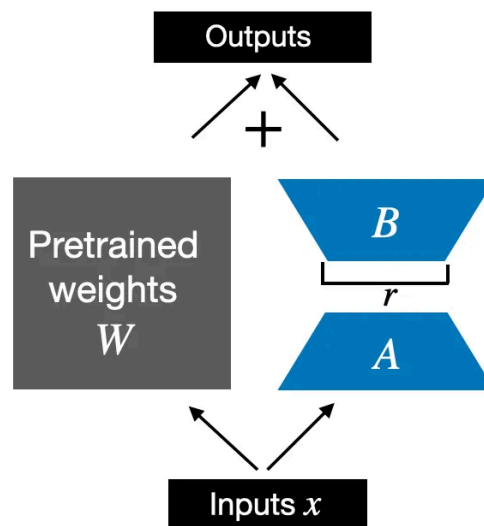
class LoRALayer(nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        super().__init__()
        std_dev = 1 / torch.sqrt(torch.tensor(rank).float())
        self.A = nn.Parameter(torch.randn(in_dim, rank) * std_dev)
        self.B = nn.Parameter(torch.zeros(rank, out_dim))
        self.alpha = alpha

    def forward(self, x):
        x = self.alpha * (x @ self.A @ self.B)
        return x
```

In the code above, `rank` is a hyperparameter that controls the inner dimension of the matrices A and B . In other words, this parameter controls the number of additional parameters introduced by LoRA and is a key factor in determining the balance between model adaptability and parameter efficiency.

The second hyperparameter, `alpha`, is a scaling hyperparameter applied to the output of the low-rank adaptation. It essentially controls the extent to which the adapted layer's output is allowed to influence the original output of the layer being adapted. This can be seen as a way to regulate the impact of the low-rank adaptation on the layer's output.

So far, the `LoRALayer` class we implemented above allows us to transform the layer inputs x . However, in LoRA, we are usually interested in replacing existing Linear layers so that the weight update is applied to the existing pretrained weights, as shown in the figure below:



LoRA applied to an existing linear layer

To incorporate the original Linear layer weights as shown in the figure above we will implement a `LinearWithLoRA` layer that uses the previously implemented `LoRALayer` and can be used to replace existing `Linear` layers in a neural network, for example, the self-attention module or feed forward modules in an LLM:

```
class LinearWithLoRA(nn.Module):

    def __init__(self, linear, rank, alpha):
        super().__init__()
        self.linear = linear
        self.lora = LoRALayer(
            linear.in_features, linear.out_features, rank, alpha
        )

    def forward(self, x):
        return self.linear(x) + self.lora(x)
```



Note that since we initialize the weight matrix B (`self.B` in `LoraLayer`) with zero values in the LoRA layer, the matrix multiplication between A and B results in a matrix consisting of 0's and doesn't affect the original weights (since adding to the original weights does not modify them).

Let's try out LoRA on a small neural network layer represented by a single Linear layer:

In:

```
torch.manual_seed(123)
layer = nn.Linear(10, 2)
x = torch.randn((1, 10))

print("Original output:", layer(x))
```

Out:

```
Original output: tensor([[0.6639, 0.4487]], grad_fn=<AddmmBackward0>)
```

Now, applying LoRA to the Linear layer, we see that the results are the same since we haven't trained the LoRA weights yet. In other words, everything was as expected:

In:



```
layer_lora_1 = LinearWithLoRA(layer, rank=2, alpha=4)
print("LoRA output:", layer_lora_1(x))
```

Out:

```
LoRA output: tensor([[0.6639, 0.4487]], grad_fn=<AddmmBackward0>)
```

Earlier, I mentioned the distributive law of matrix multiplication:

$$\mathbf{x} \cdot (\mathbf{W} + \mathbf{A} \cdot \mathbf{B}) = \mathbf{x} \cdot \mathbf{W} + \mathbf{x} \cdot \mathbf{A} \cdot \mathbf{B}.$$

Here, this means that we can also combine or merge the LoRA matrices and original weights, which should result in an equivalent implementation. In code, this alternative implementation to the `LinearWithLoRA` layer looks as follows:

```
class LinearWithLoRAMerged(nn.Module):
    def __init__(self, linear, rank, alpha):
        super().__init__()
        self.linear = linear
        self.lora = LoRALayer(
            linear.in_features, linear.out_features, rank, alpha
        )

    def forward(self, x):
        lora = self.lora.A @ self.lora.B # Combine LoRA matrices
        # Then combine LoRA with orig. weights
        combined_weight = self.linear.weight + self.lora.alpha*lora.T
        return F.linear(x, combined_weight, self.linear.bias)
```

In short, `LinearWithLoRAMerged` computes the left side of the equation $\mathbf{x} \cdot (\mathbf{W} + \mathbf{A} \cdot \mathbf{B})$ whereas `LinearWithLoRA` computes the right side -- both are equivalent.

We can verify that this results in the same outputs as before via the following code:



In:

```
layer_lora_2 = LinearWithLoRAMerged(layer, rank=2, alpha=4)
print("LoRA output:", layer_lora_2(x))
```

Out:

```
LoRA output: tensor([[0.6639, 0.4487]], grad_fn=<AddmmBackward0>)
```


Now that we have a working LoRA implementation let's see how we can apply it to a neural network in the next section.

Ahead of AI is a reader-supported publication. To receive new posts and support my work, consider becoming a free or paid subscriber.

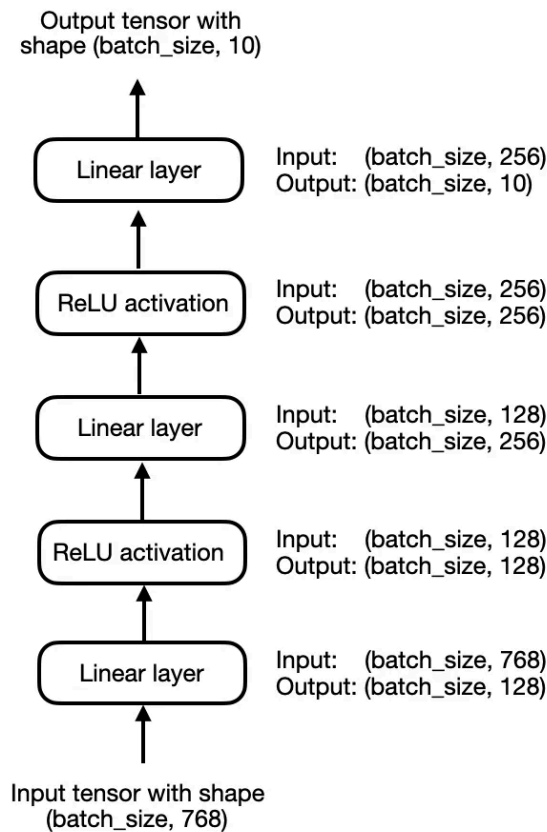
[Subscribe](#)

Applying LoRA Layers

Why did we implement LoRA in the manner described above using PyTorch modules? This approach enables us to easily replace a `Linear` layer in an existing neural network (for example, the feed forward or attention modules of a Large Language Model) with our new `LinearWithLoRA` (or `LinearWithLoRAMerged`) layer.

For simplicity, let's focus on a small 3-layer multilayer perceptron instead of LLM for now, which is illustrated in the figure below:





A simple 3-layer multilayer perceptron

In code, we can implement the multilayer perceptron, shown above, as follows:

In:



```
class MultilayerPerceptron(nn.Module):
    def __init__(self, num_features,
                  num_hidden_1, num_hidden_2, num_classes):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(num_features, num_hidden_1),
            nn.ReLU(),
            nn.Linear(num_hidden_1, num_hidden_2),
            nn.ReLU(),

            nn.Linear(num_hidden_2, num_classes)
        )

    def forward(self, x):
        x = self.layers(x)
        return x
```

```

model = MultilayerPerceptron(
    num_features=num_features,
    num_hidden_1=num_hidden_1,
    num_hidden_2=num_hidden_2,
    num_classes=num_classes
)

print(model)

```

Out:

```

MultilayerPerceptron(
  (layers): Sequential(
    (0): Linear(in_features=784, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=256, bias=True)
    (3): ReLU()
    (4): Linear(in_features=256, out_features=10, bias=True)
  )
)

```

Using `LinearWithLora`, we can then add the LoRA layers by replacing the `Linear` layers in the multilayer perceptron model:



In:

```

model.layers[0] = LinearWithLoRA(model.layers[0], rank=4, alpha=8)
model.layers[2] = LinearWithLoRA(model.layers[2], rank=4, alpha=8)
model.layers[4] = LinearWithLoRA(model.layers[4], rank=4, alpha=8)

print(model)

```

Out:

```

MultilayerPerceptron(
  (layers): Sequential(

```

```

(0): LinearWithLoRA(
  (linear): Linear(in_features=784, out_features=128, bias=True)
  (lora): LoRALayer()
)
(1): ReLU()
(2): LinearWithLoRA(
  (linear): Linear(in_features=128, out_features=256, bias=True)
  (lora): LoRALayer()
)
(3): ReLU()
(4): LinearWithLoRA(
  (linear): Linear(in_features=256, out_features=10, bias=True)
  (lora): LoRALayer()
)
)
)

```

Then, we can freeze the original `Linear` layers and only make the `LoRALayer` layers trainable, as follows:

In:

```

def freeze_linear_layers(model):
    for child in model.children():
        if isinstance(child, nn.Linear):
            for param in child.parameters():
                param.requires_grad = False
        else:
            # Recursively freeze linear layers in children modules
            freeze_linear_layers(child)

freeze_linear_layers(model)
for name, param in model.named_parameters():
    print(f"{name}: {param.requires_grad}")

```



Out:

```

layers.0.linear.weight: False
layers.0.linear.bias: False
layers.0.lora.A: True

```

```
layers.0.lora.B: True
layers.2.linear.weight: False
layers.2.linear.bias: False
layers.2.lora.A: True
layers.2.lora.B: True
layers.4.linear.weight: False
layers.4.linear.bias: False
layers.4.lora.A: True
layers.4.lora.B: True
```

Based on the `True` and `False` values above, we can visually confirm that only LoRA layers are trainable now (`True` means trainable, `False` means frozen). In practice, we would then train the network with this LoRA configuration on a new dataset or task.

To avoid making this a very long article, I am skipping over the boilerplate code to train this model. But if you are interested in the full code, you can find a standalone code notebook here: <https://github.com/rasbt/dora-from-scratch>

Furthermore, if you are interested in a LoRA from scratch explanation and application to an LLM, also check out my Lightning Studio [LoRA From Scratch: Implement Low-Rank Adaptation for LLMs in PyTorch](#).



Understanding Weight-Decomposed Low-Rank Adaptation (DoRA)

You may have noticed that we spent a lot of time implementing and talking about LoRA. That's because DoRA ([Weight-Decomposed Low-Rank Adaptation](#)) can be seen as an improvement or extension of LoRA that is built on top of it, and we can now easily adapt some of our previous code to implement DoRA.

DoRA can be described in two steps, where the first step is to decompose a pretrained weight matrix into a magnitude vector (\mathbf{m}) and a directional matrix (\mathbf{V}). The second step is applying LoRA to the directional matrix \mathbf{V} and training the magnitude vector \mathbf{m} separately.

The decomposition into magnitude and directional components is inspired by the mathematical principle that any vector can be represented as the product of its magnitude (a scalar value indicating its length) and its direction (a unit vector indicating its orientation in space).

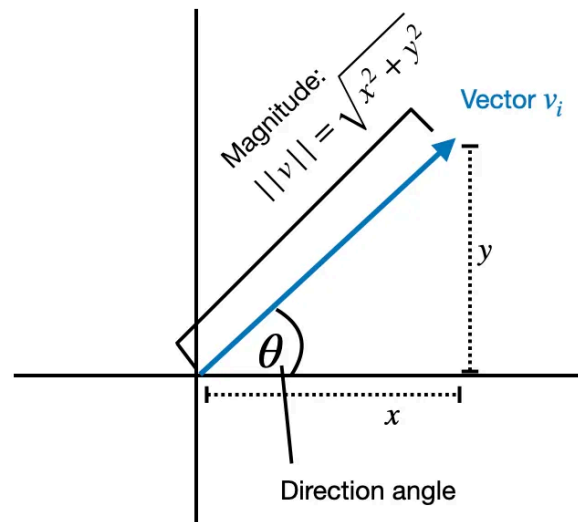


Illustration of the direction and magnitude of a single vector. For example, if we have a 2D vector $[1, 2]$, we can decompose it into a magnitude 2.24 and a directional vector $[0.447, 0.894]$. Then $2.24 * [0.447, 0.894] = [1, 2]$.

In DoRA, we apply the decomposition into magnitude and directional components to a whole pretrained weight matrix \mathbf{W} instead of a vector. Each column (vector) of the weight matrix corresponds to the weights connecting all inputs to a particular output neuron.

So, the result of decomposing \mathbf{W} is a magnitude vector \mathbf{m} that represents the scale or length of each column vector in the weight matrix, as illustrated in the figure below.

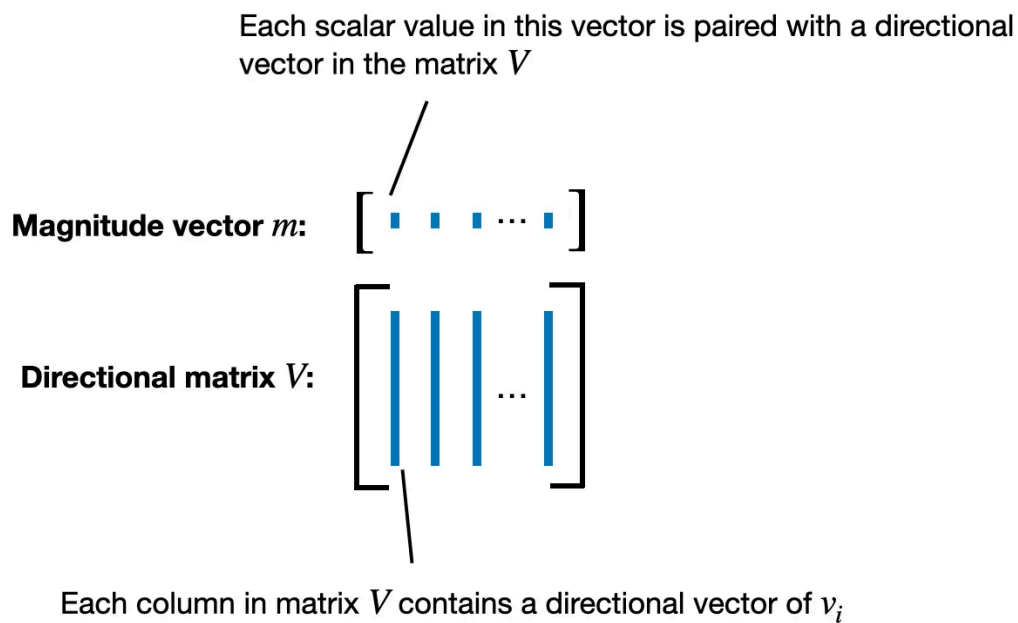


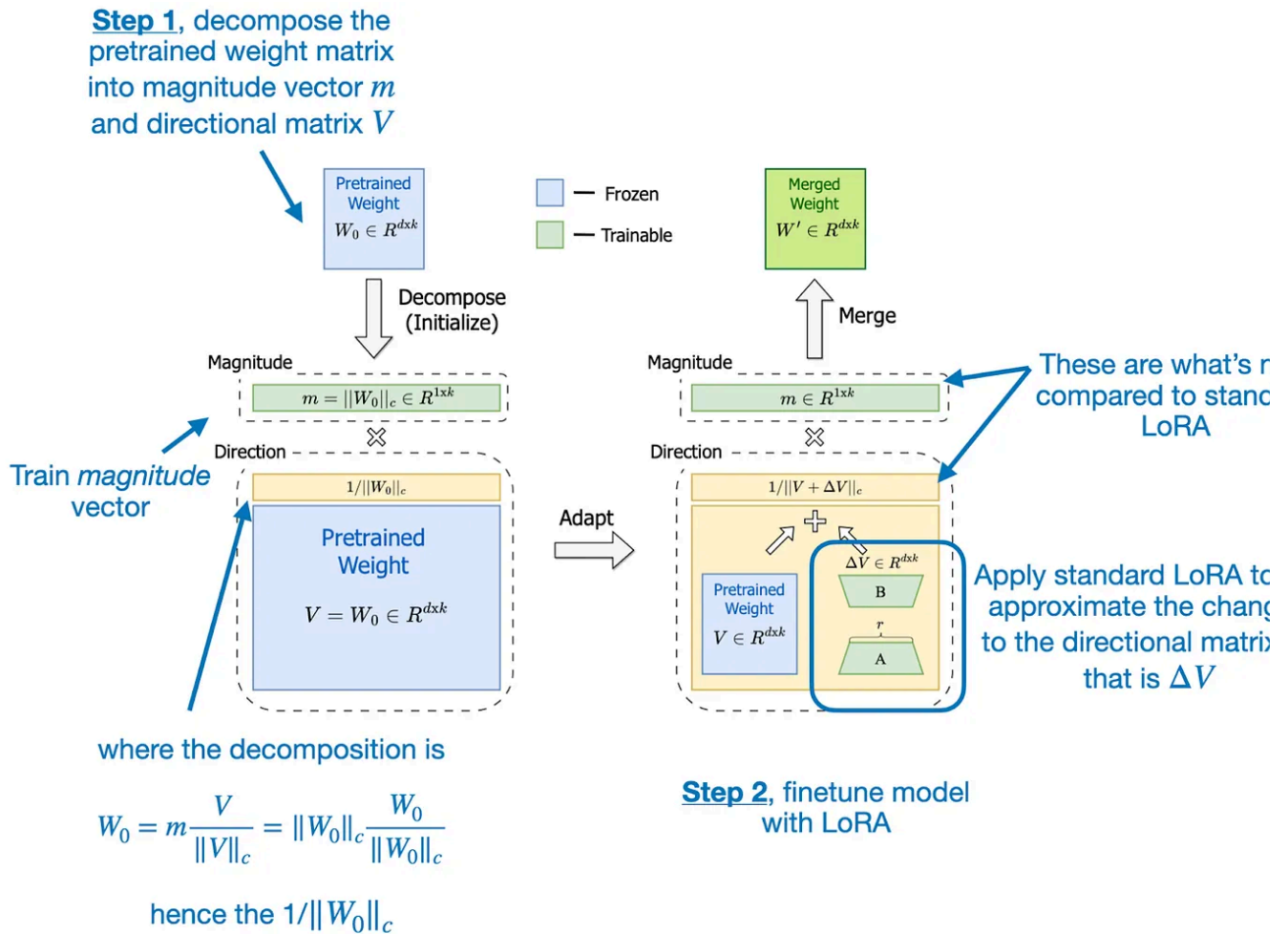
Illustration of the weight matrix decomposition in DoRA

Then, DoRA takes the directional matrix V and applies standard LoRA, for instance:

$$W' = m (V + \Delta V) / \text{norm} = m (W + AB) / \text{norm}$$

The normalization, which I abbreviated as "norm" to not further complicate things in this overview, is based on the weight normalization method proposed in Saliman's and Kingma's 2016 [Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks](https://arxiv.org/abs/1609.08248) paper.

The DoRA two-step process (decomposing a pretrained weight matrix and applying LoRA to the directional matrix) is further illustrated in the figure from the DoRA paper below.



Annotated illustration from the DoRA paper
(<https://arxiv.org/abs/2402.09353>)



The motivation for developing DoRA is based on analyzing and comparing LoRA and full finetuning learning patterns. The DoRA authors found that Lo either increases or decreases magnitude and direction updates proportionally but seems to lack the capability to make only subtle directional changes as found in full finetuning. Hence, the researchers propose the decoupling of magnitude and directional components.

In other words, their DoRA method aims to apply LoRA only to the directional component, V , while also allowing the magnitude component, m , to be trained separately.

Introducing the magnitude vector m adds 0.01% more parameters if DoRA is compared to LoRA. However, across both LLM and vision transformer benchmarks, they found that DoRA even outperforms LoRA if the DoRA ran

halved, for instance, when DoRA only uses half the parameters of regular LoRA as shown in the performance comparison below.

Higher scores are better

Model	PEFT Method	# Params (%)	BoolQ	PIQA	SIQA	HellaSwag	WinoGrande	ARC-e	ARC-c	OBQA	Avg.
ChatGPT	-	-	73.1	85.4	68.5	78.5	66.1	89.8	79.9	74.8	77.0
LLaMA-7B	Prefix	0.11	64.3	76.8	73.9	42.1	72.1	72.9	54.0	60.6	64.6
	Series	0.99	63.0	79.2	76.3	67.9	75.7	74.5	57.1	72.4	70.8
	Parallel	3.54	67.9	76.4	78.8	69.8	78.9	73.7	57.3	75.2	72.2
	LoRA	0.83	68.9	80.7	77.4	78.1	78.8	77.8	61.3	74.8	74.7
	DoRA [†] (Ours)	0.43	70.0	82.6	79.7	83.2	80.6	80.6	65.4	77.6	77.5
	DoRA (Ours)	0.84	68.5	82.9	79.6	84.8	80.8	81.4	65.8	81.0	78.1

Other parameter-efficient finetuning methods

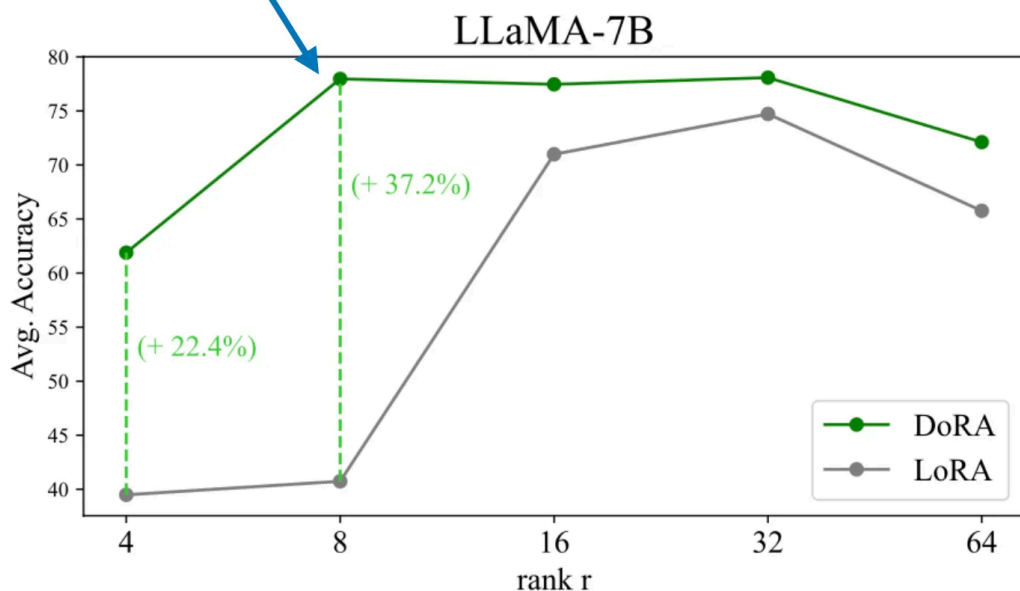
Even with parameter outperform

DoRA with half the parameters

Comparison between LoRA and DoRA from the DoRA paper
(<https://arxiv.org/abs/2402.09353>)

As I wrote in another article a few months ago, LoRA requires careful tuning the rank to optimize performance: [Practical Tips for Finetuning LLMs Using LoRA \(Low-Rank Adaptation\)](#). However, DoRA seems to be much more robust to changes in rank, as shown in the comparison below.

DoRA achieves good performance even if the rank is relatively small



DoRA is more robust to the rank hyperparameter than LoRA (annotated figure from the DoRA paper, <https://arxiv.org/abs/2402.09353>)

The possibility to successfully use DoRA with relatively small ranks makes the method even more parameter-efficient than LoRA.

Overall, I am quite impressed by the results, and it should not be too big of lift to upgrade a LoRA implementation to DoRA, which we will do in the next section.

Implementing DoRA Layers in PyTorch

In this section, we will see what DoRA looks like in code. Previously, we said we can initialize a pretrained weight \mathbf{W}_0 with magnitude \mathbf{m} and directional component \mathbf{V} . For instance, we have the following equation:

$$\mathbf{W}_0 = \mathbf{m} \frac{\mathbf{V}}{\|\mathbf{V}\|_c}$$

where $\|\mathbf{V}\|_c$ is the vector-wise norm of \mathbf{V} . Then we can write DoRA including LoRA weight update \mathbf{BA} as shown below:

$$\mathbf{W}' = \mathbf{m} \frac{\mathbf{V} + \mathbf{BA}}{\|\mathbf{V} + \mathbf{BA}\|_c}$$

Now, in the DoRA paper, the authors formulate DoRA as follows, where they use the initial pretrained weights \mathbf{W}_0 as the directional component directly and learn magnitude vector \mathbf{m} during training:

$$\mathbf{W}' = \mathbf{m} \frac{\mathbf{V} + \Delta\mathbf{V}}{\|\mathbf{V} + \Delta\mathbf{V}\|_c} = \mathbf{m} \frac{\mathbf{W}_0 + \mathbf{BA}}{\|\mathbf{W}_0 + \mathbf{BA}\|_c}$$



Here, $\Delta\mathbf{V}$ is the update to the directional component, matrix \mathbf{V} .

While the original authors haven't released the official implementation yet, you can find an independent implementation [here](#), which loosely inspired my implementation below

Taking our previous `LinearWithLoRAMerged` implementation, we can upgrade DoRA as follows:

```
class LinearWithDoRAMerged(nn.Module):

    def __init__(self, linear, rank, alpha):
        super().__init__()
```

```

self.linear = linear
self.lora = LoRALayer(
    linear.in_features, linear.out_features, rank, alpha
)
self.m = nn.Parameter(
    self.linear.weight.norm(p=2, dim=0, keepdim=True))

# Code loosely inspired by
# https://github.com/catid/dora/blob/main/dora.py

def forward(self, x):
    lora = self.lora.A @ self.lora.B
    numerator = self.linear.weight + self.lora.alpha*lora.T
    denominator = numerator.norm(p=2, dim=0, keepdim=True)
    directional_component = numerator / denominator
    new_weight = self.m * directional_component
    return F.linear(x, new_weight, self.linear.bias)

```

The `LinearWithDoRAMerged` class is different from our previous `LinearWithLoRAMerged` class in several key aspects, primarily in how it modifies and applies the weights of the Linear layer. However, both classes integrate `LoRALayer` to augment the original linear layer's weights, but DoRA adds weight normalization and adjustment.



The figure below shows a file-diff of both classes side by side:

```

17- class LinearWithLoRAMerged(nn.Module):
18-     def __init__(self, linear, rank, alpha):
19-         super().__init__()
20-         self.linear = linear
21-         self.lora = LoRALayer(
22-             linear.in_features, linear.out_features, rank, alpha
23-         )
24-
25-
26-     def forward(self, x):
27-         lora = self.lora.A @ self.lora.B
28-         combined_weight = self.linear.weight + self.lora.alpha*lora.T
29-
30-         return F.linear(x, combined_weight, self.linear.bias)

```

```

17+ class LinearWithDoRAMerged(nn.Module):
18+     def __init__(self, linear, rank, alpha):
19+         super().__init__()
20+         self.linear = linear
21+         self.lora = LoRALayer(
22+             linear.in_features, linear.out_features, rank, alpha
23+         )
24+
25+
26+         self.m = nn.Parameter(
27+             self.linear.weight.norm(p=2, dim=0, keepdim=True))
28+
29+     def forward(self, x):
30+         lora = self.lora.A @ self.lora.B
31+         combined_weight = self.linear.weight + self.lora.alpha*lora.T
32+         column_norm = combined_weight.norm(p=2, dim=0, keepdim=True)
33+         V = combined_weight / column_norm
34+         new_weight = self.m * V
35+         return F.linear(x, new_weight, self.linear.bias)
36+

```

File-diff between `LinearWithLoRAMerged` and `LinearWithDoRAMerged`

As we can see in the figure above, `LinearWithDoRAMerged` introduces an additional step involving dynamic normalization of the augmented weights.

After combining the original weights with the LoRA-adjusted weights (`self.linear.weight + self.lora.alpha*lora.T`), it calculates the norm of the combined weights across columns (`column_norm`). Then, it normalizes the combined weights by dividing them by their norms ($V = \text{combined_weight} / \text{column_norm}$). This step ensures that each column of the combined weight matrix has a unit norm, which can help stabilize the learning process by maintaining the scale of weight updates.

DoRA also introduces a learnable vector `self.m`, which represents the magnitude of each column of the normalized weight matrix. This parameter allows the model to dynamically adjust the scale of each weight vector in the combined weight matrix during training. This additional flexibility can help the model better capture the importance of different features.

In summary, `LinearWithDoRAMerged` extends the concept of `LinearWithLoRAMerged` by incorporating dynamic weight normalization and scaling to improve the training performance.



In practice, considering the multilayer perceptron from earlier, we can simply swap existing Linear layers with our `LinearWithDoRAMerged` layers as follows:

In:

```
model.layers[0] = LinearWithDoRAMerged(model.layers[0], rank=4, alpha=8)
model.layers[2] = LinearWithDoRAMerged(model.layers[2], rank=4, alpha=8)
model.layers[4] = LinearWithDoRAMerged(model.layers[4], rank=4, alpha=8)

print(model)
```

Out:

```

MultilayerPerceptron(
  (layers): Sequential(
    (0): LinearWithDoRAMerged(
      (linear): Linear(in_features=784, out_features=128, bias=True)
      (lora): LoRALayer()
    )
    (1): ReLU()
    (2): LinearWithDoRAMerged(
      (linear): Linear(in_features=128, out_features=256, bias=True)
      (lora): LoRALayer()
    )
    (3): ReLU()
    (4): LinearWithDoRAMerged(
      (linear): Linear(in_features=256, out_features=10, bias=True)
      (lora): LoRALayer()
    )
  )
)

```

Before we finetune the model, we can reuse the `freeze_linear_layers` function we implemented earlier to only make the LoRA weights and magnitude vector trainable:

In:



```

freeze_linear_layers(model)
for name, param in model.named_parameters():
    print(f"{name}: {param.requires_grad}")

```

Out:

```

layers.0.m: True
layers.0.linear.weight: False
layers.0.linear.bias: False
layers.0.lora.A: True
layers.0.lora.B: True
layers.2.m: True
layers.2.linear.weight: False
layers.2.linear.bias: False

```

```

layers.2.lora.A: True
layers.2.lora.B: True
layers.4.m: True
layers.4.linear.weight: False
layers.4.linear.bias: False
layers.4.lora.A: True
layers.4.lora.B: True

```

The full code example, including model training, is available in my GitHub repo here: <https://github.com/rasbt/dora-from-scratch>.

Train model with DoRA

```

freeze_linear_layers(model_dora)

# Check if linear layers are frozen
for name, param in model_dora.named_parameters():
    print(f"{name}: {param.requires_grad}")

```

```

layers.0.m: True
layers.0.linear.weight: False
layers.0.linear.bias: False
layers.0.lora.A: True
layers.0.lora.B: True
layers.2.m: True
layers.2.linear.weight: False
layers.2.linear.bias: False
layers.2.lora.A: True
layers.2.lora.B: True
layers.4.m: True
layers.4.linear.weight: False
layers.4.linear.bias: False
layers.4.lora.A: True
layers.4.lora.B: True

```



```

optimizer_dora = torch.optim.Adam(model_dora.parameters(), lr=learning_rate)
train(num_epochs, model_dora, optimizer_dora, train_loader, DEVICE)
print(f'Test accuracy DoRA finetune: {compute_accuracy(model_dora, test_loader, DEVICE):.2f}%')

```

```

Epoch: 001/002 | Batch 000/938 | Loss: 0.0661
Epoch: 001/002 | Batch 400/938 | Loss: 0.0871
Epoch: 001/002 | Batch 800/938 | Loss: 0.1102
Epoch: 001/002 training accuracy: 98.04%
Time elapsed: 0.13 min
Epoch: 002/002 | Batch 000/938 | Loss: 0.0332
Epoch: 002/002 | Batch 400/938 | Loss: 0.0148
Epoch: 002/002 | Batch 800/938 | Loss: 0.0471
Epoch: 002/002 training accuracy: 98.03%
Time elapsed: 0.26 min
Total Training Time: 0.26 min
Test accuracy DoRA finetune: 97.26%

```

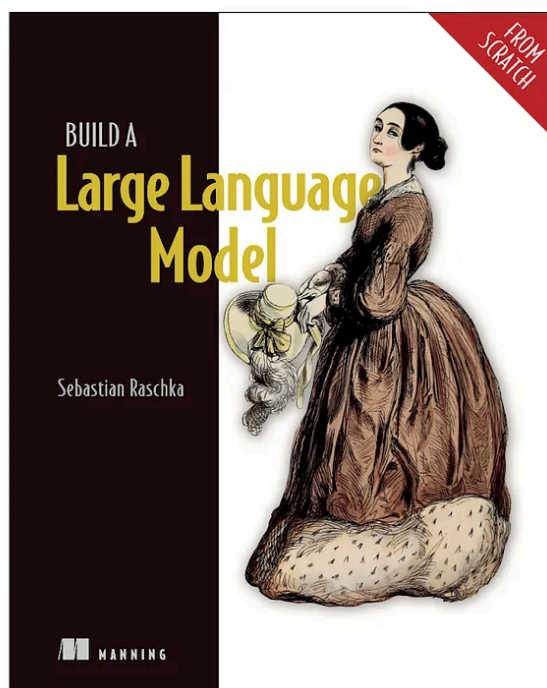
DoRA code notebook: <https://github.com/rasbt/dora-from-scratch>

Conclusion

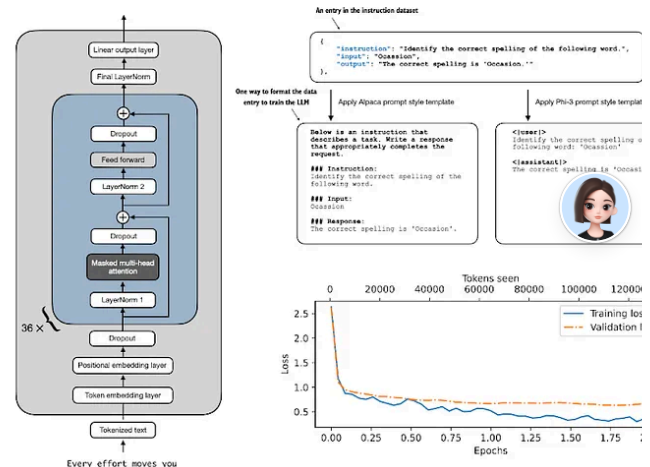
In my opinion, DoRA seems like a logical, effective, and promising extension of LoRA, and I am excited to try it in real-world LLM finetuning contexts.

In the meantime, I also added the DoRA implementation above to the [LoRA From Scratch – Implement Low-Rank Adaptation for LLMs in PyTorch](#) Lightning Studio to finetune a DistilBERT language model (see `bonus_02_finetune-with-dora.ipynb`). Even without hyperparameter tuning, I already saw a >1% prediction accuracy improvement over LoRA.

This magazine is a personal passion project. For those who wish to support please consider purchasing a copy of my [Build a Large Language Model \(From Scratch\)](#) book. (I am confident that you'll get lots out of this book as it explains how LLMs work in a level of detail that is not found anywhere else.)



Deeply understand LLMs by implementing them from the ground up



Build a Large Language Model (From Scratch) now [available on Amazon](#)

If you read the book and have a few minutes to spare, I'd really appreciate a [brief review](#). It helps us authors a lot!

Alternatively, I also recently enabled the paid subscription option on Substa to support this magazine directly.

Ahead of AI is a reader-supported publication. To receive new posts and support my work, consider becoming a free or paid subscriber.

Subscribe



183 Likes · 17 Restacks

Discussion about this post

- Comments
- Restacks



Write a comment...



Samuel Flender 2024年2月19日

♥ Liked by Sebastian Raschka, PhD



Fantastic write-up, thank you!

Small correction:

"The DoRA two-step process (decomposing a pretrained weight matrix and applying Do the directional matrix) is further illustrated in the figure from the DoRA paper below."

applying DoRA to the directional matrix --> applying LoRA to the directional matrix.

♡ LIKE (3) 💬 REPLY

1 reply by Sebastian Raschka, PhD



Ketut Artayasa 2024年2月19日

♥ Liked by Sebastian Raschka, PhD

```
layer_lora_1 = LinearWithLoRA(layer, rank=2, alpha=4)
print("LoRA output:", layer_lora_2(x))
```