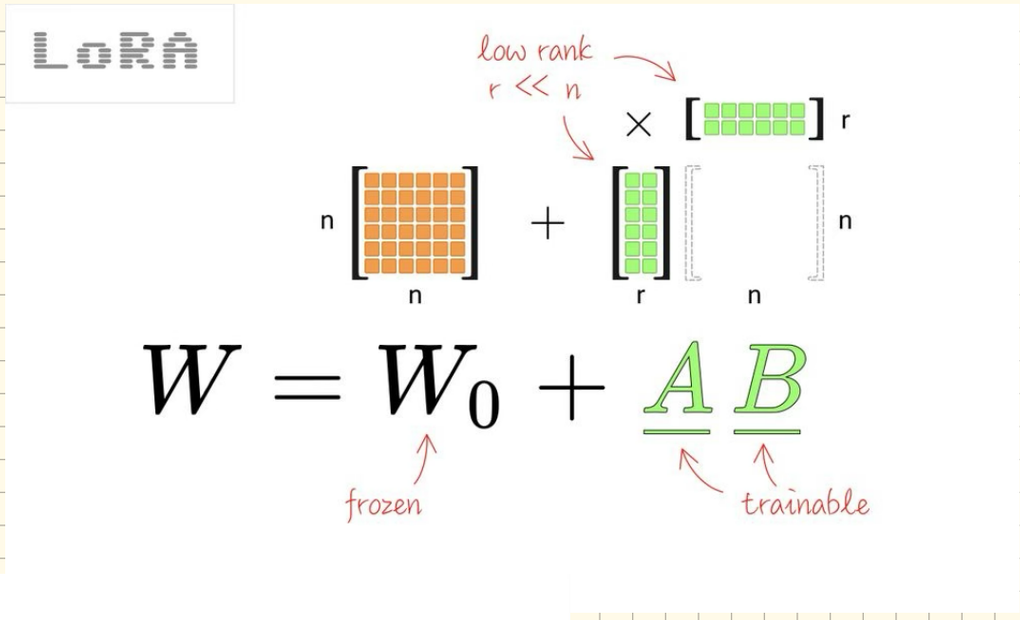
LoRA



🔍 What is LoRA?

LoRA is a technique used to fine-tune large pre-trained models (especially transformers) by **injecting trainable low-rank matrices** into the weights, instead of updating the full weight matrices. This drastically reduces the number of trainable parameters and computation.

📐 Mathematics Behind LoRA

In a transformer, weight matrices (like in self-attention or feedforward layers) are often large, say $W \in \mathbb{R}^{d \times k}$. Instead of updating W, LoRA freezes it and adds a low-rank decomposition:

$$W' = W + \Delta W \quad \text{where} \quad \Delta W = AB$$

- $A \in \mathbb{R}^{d \times r}$ and $B \in \mathbb{R}^{r \times k}$
- $r \ll \min(d, k)$: the rank is small

So during training:

- Only A and B are updated (much smaller)
- W is kept frozen
- At inference: either keep them separate or merge $W + AB$

✅ Benefits

- **Parameter efficiency**: Only $O(r \cdot (d + k))$ parameters are trained vs $O(dk)$
- **Computational efficiency**: FLOPs reduced
- **Storage**: Much smaller fine-tuned model files

🎻 Analogy: Customizing a Violin Performance

Imagine you're a **violinist** playing a complex piece of music. You're already highly skilled and have memorized a standard version of the piece (the **pretrained model**).

Now, suppose a **composer** wants you to slightly **adapt** your performance for a new audience (fine-tuning), but:

- They **don't want to retrain you completely**.
- They **only want minimal changes** to match the new style.
- And they don't want you to forget the original version.

🧠 **Standard Fine-Tuning:**

This would be like retraining the violinist from scratch — changing their entire muscle memory. It's expensive, time-consuming, and risks forgetting the original piece.

🎼 **LoRA Fine-Tuning:**

Instead, LoRA gives the violinist **a few small notecards** (low-rank matrices A and B) with subtle style changes:

- **"Add a soft vibrato here."**
- **"Play this part slightly faster."**
- **"Emphasize this note."**

These cards don't rewrite the entire performance. They're **small adjustments** on top of the original. The violinist still plays the same core music (frozen weights), but now **adapts** it with minimal, efficient tweaks.

These tweaks are:

- **Small in size** (low-rank)
- **Fast to read** (low FLOPs)
- **Easy to swap out** for other tasks or audiences

Let's say:

- $W \in \mathbb{R}^{512 \times 512}$ (full-rank)
- LoRA rank $r = 8$
- Input vector $x \in \mathbb{R}^{512}$

Original FLOPs:

- Multiply Wx: $512 \times 512 = 262,144$ multiply-adds

LoRA FLOPs:

- $A \in \mathbb{R}^{512 \times 8}$, $B \in \mathbb{R}^{8 \times 512}$
- $A^T x$: $8 \times 512 = 4096$ FLOPs
- Then $B(A^T x)$: $512 \times 8 = 4096$ FLOPs
- Total: **8,192 FLOPs**

That's **~32x less** computation than full-rank (not counting the frozen Wx part if reused).

🧠 Python Implementation (Simple Version)

python Copy

```python
import numpy as np

# Full weight (frozen)
W = np.random.randn(512, 512)

# LoRA parameters
r = 8  # rank
A = np.random.randn(512, r)
B = np.random.randn(r, 512)

# Input vector
x = np.random.randn(512)

# Forward pass with LoRA
# Normal: y = Wx
# LoRA: y = Wx + B(A^T x)
Wx = W @ x  # frozen weight
delta_Wx = B @ (A.T @ x)  # low-rank update

y = Wx + delta_Wx

print("Output shape:", y.shape)
```

📉 Summary of Optimization

Method	Trainable Params	FLOPs	Memory	Accuracy
Full Fine-Tune	$512^2 = 262k$	High	High	High
LoRA (r=8)	$8 * (512 + 512) = 8.2k$	Low	Low	High (≈ full)

state dicts

🔎 What's in a state_dict?

A typical model's `state_dict` might look like:

python Copy

```
{
    'linear1.weight': tensor([[...], [...], ...]),
    'linear1.bias': tensor([...]),
    'linear2.weight': tensor([[...], [...], ...]),
    'linear2.bias': tensor([...]),
}
```

Each entry corresponds to:

- A **layer name** (including submodule paths)
- A **parameter tensor**

> ⚠️ It does **not** include the module structure or logic—only **trained parameters**.

🧠 Example from nn.Sequential

python

```
import torch.nn as nn

model = nn.Sequential(
    nn.Linear(4, 3),
    nn.ReLU(),
    nn.Linear(3, 2),
)

print(model.state_dict())
```

🔽 Output (simplified):

python

```
OrderedDict([
  ('0.weight', tensor([...4x3...])),
  ('0.bias', tensor([...])),
  ('2.weight', tensor([...3x2...])),
  ('2.bias', tensor([...]))
])
```

- `0.` refers to the first layer (`Linear`)
- `2.` refers to the third layer (another `Linear`)
- The ReLU layer has no learnable parameters, so it's not in the dict.

🧠 Example 1: Simple Self-Attention Layer

python

```
import torch
import torch.nn as nn

class SimpleSelfAttention(nn.Module):
    def __init__(self, embed_dim):
        super().__init__()
        self.q_proj = nn.Linear(embed_dim, embed_dim)
        self.k_proj = nn.Linear(embed_dim, embed_dim)
        self.v_proj = nn.Linear(embed_dim, embed_dim)
        self.out_proj = nn.Linear(embed_dim, embed_dim)

    def forward(self, x):
        Q = self.q_proj(x)
        K = self.k_proj(x)
        V = self.v_proj(x)
        attn_weights = torch.softmax(Q @ K.transpose(-2, -1) / (
        out = attn_weights @ V
        return self.out_proj(out)

model = SimpleSelfAttention(embed_dim=4)
```

🔍 state_dict() Output

python

```
print(model.state_dict().keys())
```

```
odict_keys([
    'q_proj.weight', 'q_proj.bias',
    'k_proj.weight', 'k_proj.bias',
    'v_proj.weight', 'v_proj.bias',
    'out_proj.weight', 'out_proj.bias'
])
```

🧠 So here you get 4 linear layers → each has `.weight` and `.bias`.

what is happening in LoRA code in mistral

python Copy

```python
def ignore_missing_keys(m: nn.Module, incompatible_keys: NamedTuple) -> None:
    incompatible_keys.missing_keys[:] = []  # type: ignore

self.register_load_state_dict_post_hook(ignore_missing_keys)
```

Let's **explain this step-by-step**, using a **numerical example**, so you see how it works and *why* it's needed.

🔍 Purpose (High Level)

When you call `model.load_state_dict(...)`, PyTorch checks if all expected parameters are present in the state dictionary.

- If some parameters are **missing**, PyTorch will **warn or throw errors**.
- This hook suppresses such warnings by **clearing the list of missing keys** *after loading*.

It's used here to **allow loading standard weights** (e.g. `linear.weight`) into `LoRALinear`, which also has `lora_A` and `lora_B` — even if the file doesn't include those.

🧪 Numerical Example

Step 1: Your LoRALinear has this internal structure:

python

```
LoRALinear(
    lora_A: Linear(4 → 2)
    lora_B: Linear(2 → 6)
    linear: Linear(4 → 6)
)
```

So the full list of expected weights is:

python

```
{
    'linear.weight': shape (6, 4),
    'lora_A.weight': shape (2, 4),
    'lora_B.weight': shape (6, 2),
}
```

Step 2: You load a checkpoint like this:

python

```
checkpoint = {
    'linear.weight': torch.randn(6, 4)
}
```

Now, if you do:

python

```
model.load_state_dict(checkpoint)
```

🔻 PyTorch will say:

> ❗ Missing keys: ['lora_A.weight', 'lora_B.weight']

That's because those keys are part of the model but not in the checkpoint.

🧩 3. What ignore_missing_keys() Really Does

When you register:

python Copy

```
self.register_load_state_dict_post_hook(ignore_missing_keys)
```

You tell PyTorch:

> "I *know* `lora_A` and `lora_B` aren't in the file. That's OK. Just ignore that fact."

Under the hood:

python Copy

```
incompatible_keys.missing_keys[:] = []
```

This clears the error list. So `load_state_dict()` thinks everything went fine.

➡️ **Those layers (lora_A, lora_B) will just keep their default-initialized values**, which come from `nn.Linear(...)`.

🧪 4. Concrete Example

Model Code

python Copy

```
class MyLoRALinear(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(4, 6)
        self.lora_A = nn.Linear(4, 2)
        self.lora_B = nn.Linear(2, 6)

        def ignore_missing_keys(m, incompatible_keys):
            incompatible_keys.missing_keys[:] = []

        self.register_load_state_dict_post_hook(ignore_missing_keys)
```

Training or Base Checkpoint File

python Copy

```
state_dict = {
    'linear.weight': torch.randn(6, 4)
}
```

Result of Loading

python Copy

```
model.load_state_dict(state_dict)
```

- `linear.weight` is restored from file ✅
- `lora_A.weight`, `lora_B.weight` remain at their **default random initialization** (they're learnable unless frozen)

This allows you to:

- Load **base model weights** (e.g. pre-trained GPT-2)
- Add **LoRA layers** for fine-tuning
- Without needing those LoRA weights to already exist in the file

🧠 But You Know the Truth

> Those layers (lora_A, lora_B) **didn't exist during original training.**
> They are **new layers you added for LoRA fine-tuning.**

So, it's **correct** that there are no LoRA weights in the file.

You're intentionally:

- Keeping `linear` frozen (loaded from checkpoint ✅)
- Training only `lora_A` and `lora_B` (random init ❄️)