Fine-Tuning, Deployment & Business Integration

03-Hours Seminar @ NUST, ISB
PART 2

Dr. Basharat Hussain
Assistant Professor, NUCES
Islamabad, Pakistan

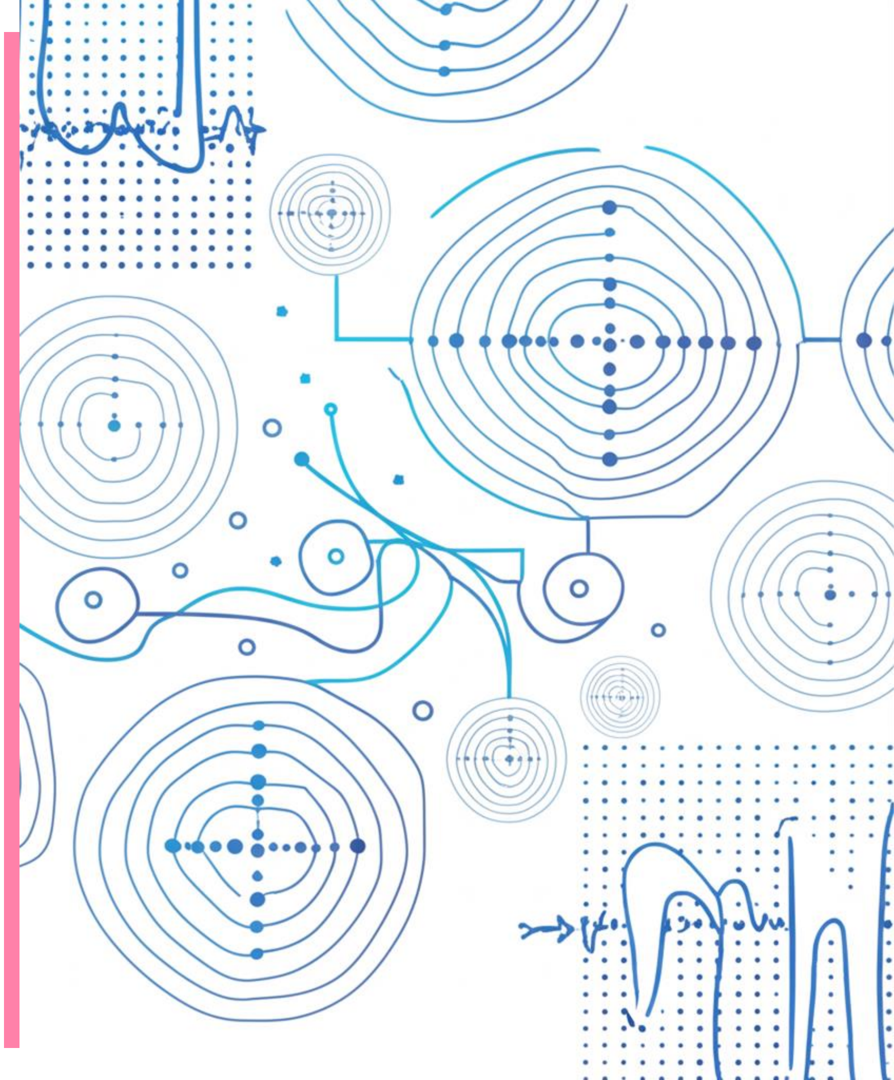
28th Aug 2025

Fine-Tuning Using LoRA

FINE-TUNING DEPLOYMENT BUSINESS INTEGRATION

Complete Guide to LLM Fine Tuning for Beginners

- The code step by step guide to replicate fine-tuning process.



LoRA

It is too expensive to fine-tune all parameters in a large model.

- During fine-tuning we initialized with pre-trained params $\Phi_0$and $\Phi_0 + \Delta\Phi$ updated to by following the objective: $\max_{\Phi} \sum \sum \log(p_{\Phi}(y_t | x, y_{<t}))$
- We can hypothesize that the update matrices in LM adaptation have a low "intrinsic rank", leading to Low-Rank Adaptation (LoRA)
- For each downstream task, we do not need to store/deploy a different set of $\Delta\Phi$ where $|\Phi_0| = |\Delta\Phi|$

Can we find a param-efficient approach by low intrinsic rank?

$$\Phi' = \Phi_0 + \Delta\Phi$$

LoRA in Training and Inference

Previous study shows that

- Pre-trained LLMs have a “low intrinsic dimension”
- LLMs can still learn efficiently despite a low-dim reparametrization

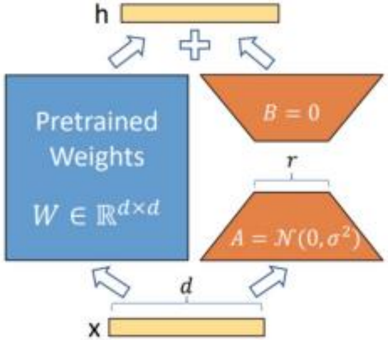


Figure 1: Our reparametrization. We only train A and B.

During training: for pre-trained weight $W_0 \in \mathbb{R}^{d\times k}$, W_0 is fixed

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

$$B \in \mathbb{R}^{d\times r},\ A \in \mathbb{R}^{r\times k},\ r \ll \min(d, k)$$

During inference:

$$W = W_0 + BA$$

Example of LoRA

- Suppose you want to fine-tune a **7B parameter LLM** for **sentiment analysis**.
- **Without LoRA**:
  - You'd need to update **all 7 billion parameters**, which is expensive in compute, memory, and storage.
- **With LoRA**:
  - Imagine one weight matrix in the model is W_0 with shape 4096×4096 (that's ~16 million parameters).
  - Instead of updating all 16M parameters, you freeze W_0.
  - Choose rank $r=8$.
  - Train two small matrices:
    - A with shape 8×4096 (~32k params)
    - B with shape 4096×8 (~32k params)
  - Total trainable parameters ≈ **64k**, compared to 16M.
- **Training step**:
  - Model computes $h = W_0 x + BAx$.
  - Only A and B change during training.
- **Inference step**:
  - The effective weight is $W = W_0 + BA$.
  - The model behaves as if it had a new weight matrix specialized for sentiment analysis, without ever altering W_0.

LoRA Toy Example — Step-by-step

This document shows the toy example used to explain LoRA's low-rank adaptation.

Dimensions used: $d = 3$, $k = 3$, $r = 2$.

1. Pretrained weight (frozen)

Pretrained weight matrix W_0 (shape 3×3):

```
[[ 2. ,  0.5, -1. ],
 [ 0. ,  1. ,  0.5],
 [-0.5,  0.5,  1.5]]
```

2. LoRA factors (trainable)

LoRA matrices:

- B (shape 3×2):

```
[[ 0.6, -0.2],
 [-0.1,  0.3],
 [ 0.4,  0.5]]
```

- A (shape 2×3):

```
[[ 0.2, -0.3,  0.1],
 [-0.4,  0.2,  0.5]]
```

3. Low-rank update

Compute BA (shape 3×3):

```
[[ 0.2 , -0.22, -0.04],
 [-0.14,  0.09,  0.14],
 [-0.12, -0.02,  0.29]]
```

4. Adapted weight

Adapted weight used at inference: $W = W_0 + BA$:

```
[[ 2.2 ,  0.28, -1.04],
 [-0.14,  1.09,  0.64],
 [-0.62,  0.48,  1.79]]
```

5. Example input vector

Input vector x:

```
[ 1. , -2. ,  0.5]
```

6. Forward computations

- Base output (no LoRA): $h_{base} = W_0 x$:

```
[ 0.5 , -1.75, -0.75]
```

- Training-time view (apply low-rank update separately): $h = W_0 x + BAx$:

```
[ 1.12 , -2.   , -0.685]
```

- Inference-time view (merged weights): $h = Wx$:

```
[ 1.12 , -2.   , -0.685]
```

7. Parameter counts

- Full matrix parameters: 9
- LoRA parameters (B and A): 12

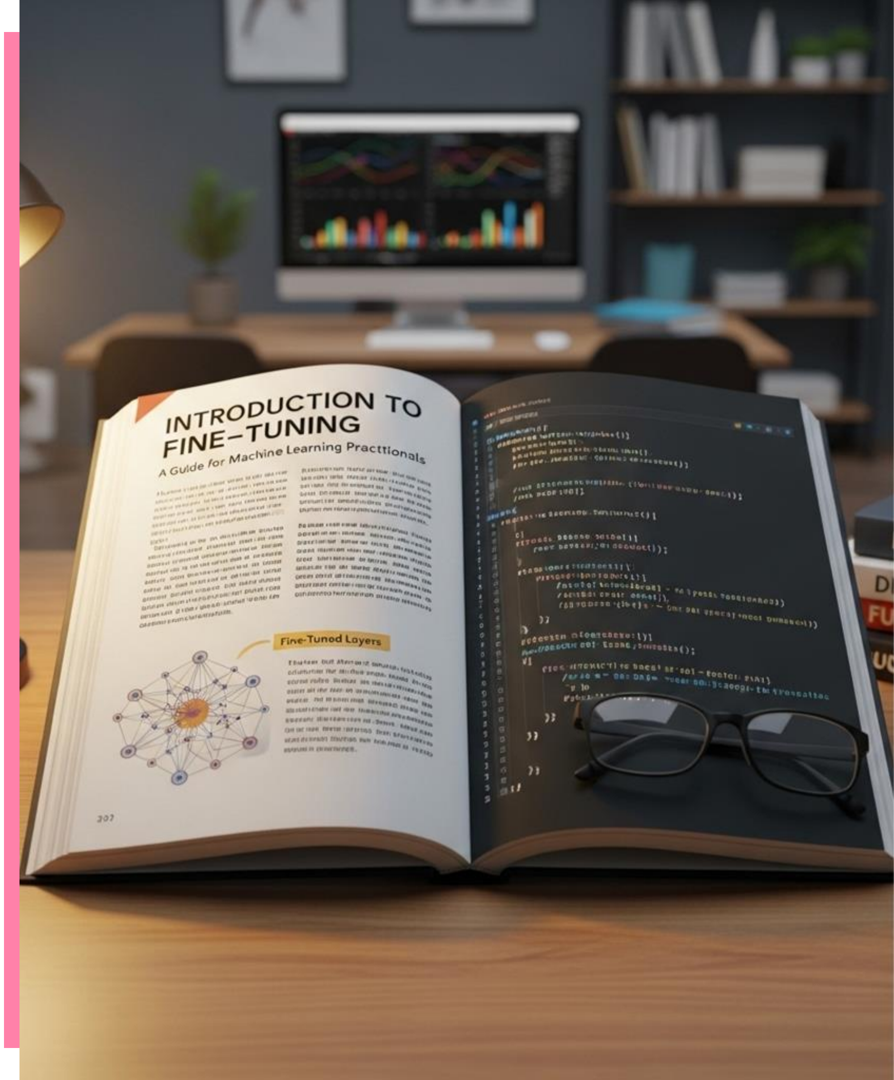
8. Notes

Only A and B are trained; W_0 remains frozen. The example demonstrates that computing W0 x + BA x (training view) yields the same result as using the merged W = W0 + BA (inference view).

Code Understanding

- **Huggingface transformers library -** allows you to download, train and fine tune pre-trained models
- **Dataset -** Library will allow you to load a dataset in JSON, CSV, Parquet, text and other formats

- from datasets import load_dataset
- from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig, TrainingArguments, Trainer



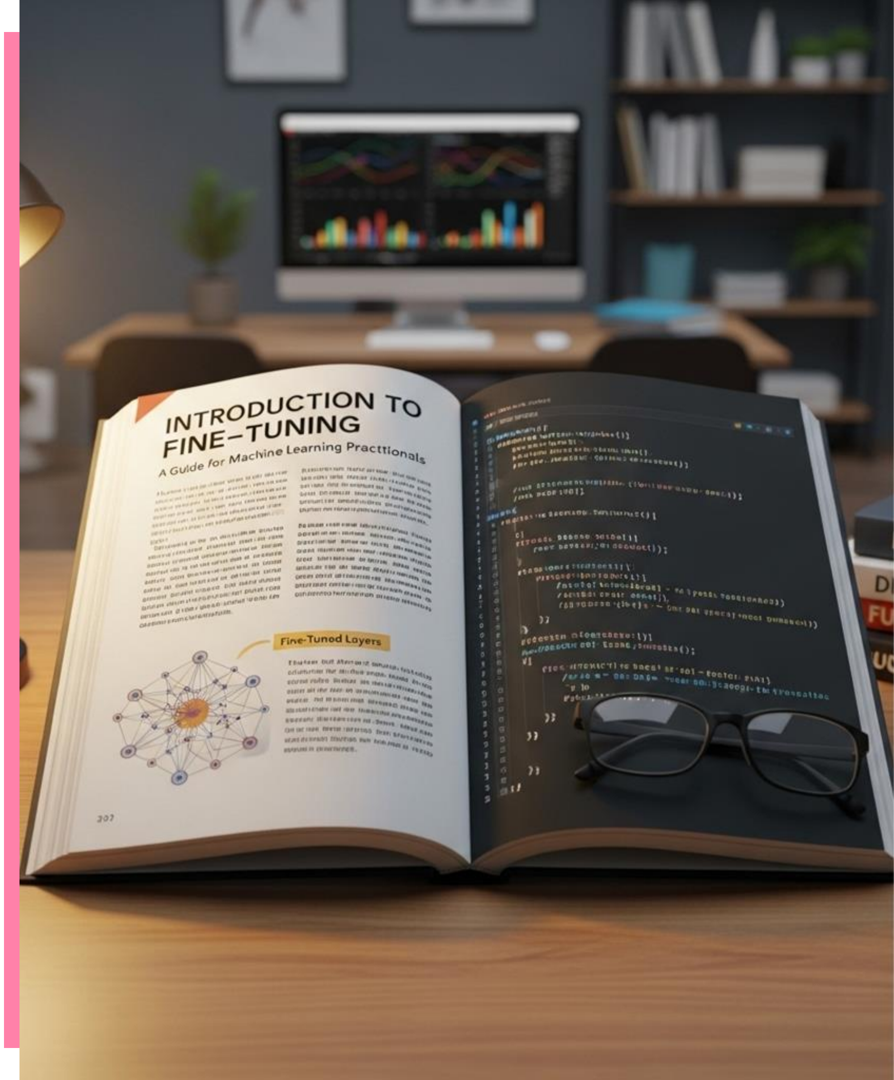
Code Understanding

- ✓ **PEFT** - Parameter-Efficient Fine-tuning library that provides a unified interface for loading and managing PEFT methods, including LoRA.

- ✓ fine-tunes a small number of (extra) model parameters or weights while freezing most parameters of the pre trained LLMs.

- ✓ Fine tuning entire LLM would require incredible hardware but with PEFT, you can fine tune a giant LLM on a regular consumer GPU.

- ✓ from peft import PeftModel, PeftConfig



Code 1: Guide to LLM Fine-tuning

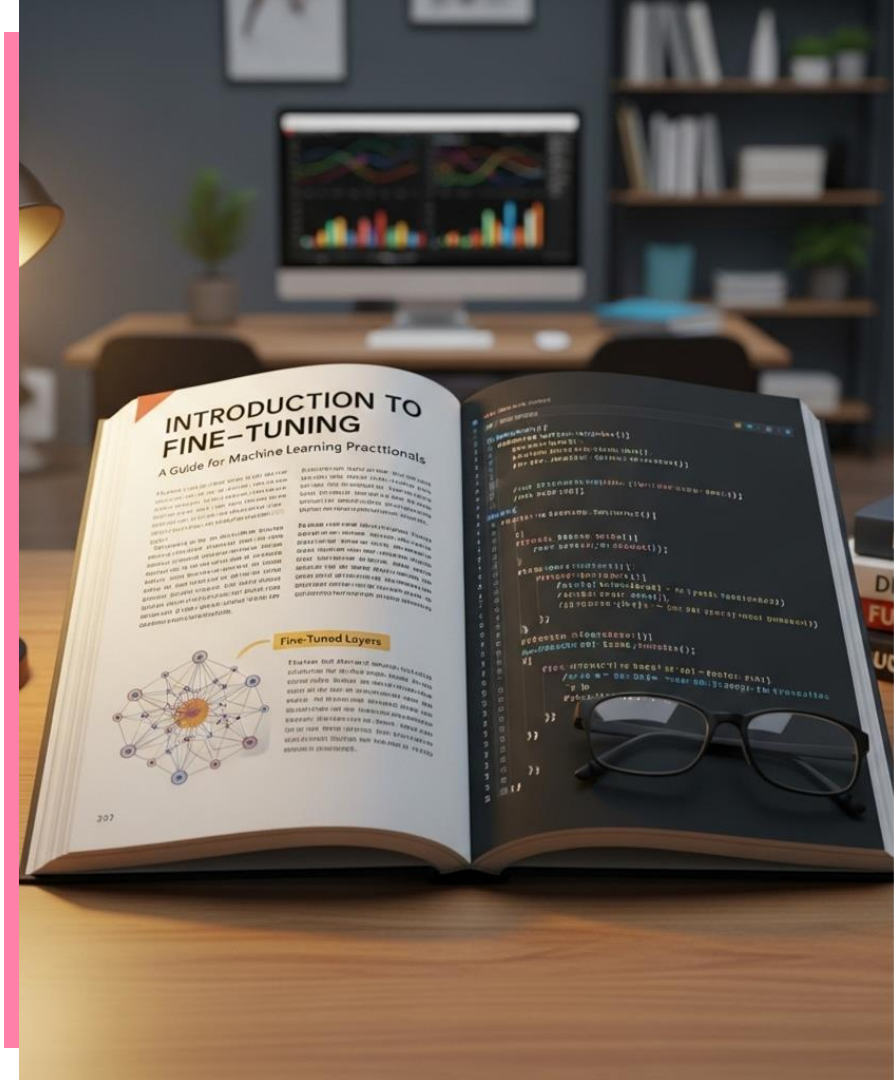
- **Lora (Low-Rank Adaptation of Large Language Models)** - is a specific category of PEFT techniques. It focuses on freezing the pre-trained model weights

- The SFTTrainer from trl provides integration with LoRA adapters through the PEFT library.

- use the LoRAConfig class from. The setup requires just a few configuration steps:

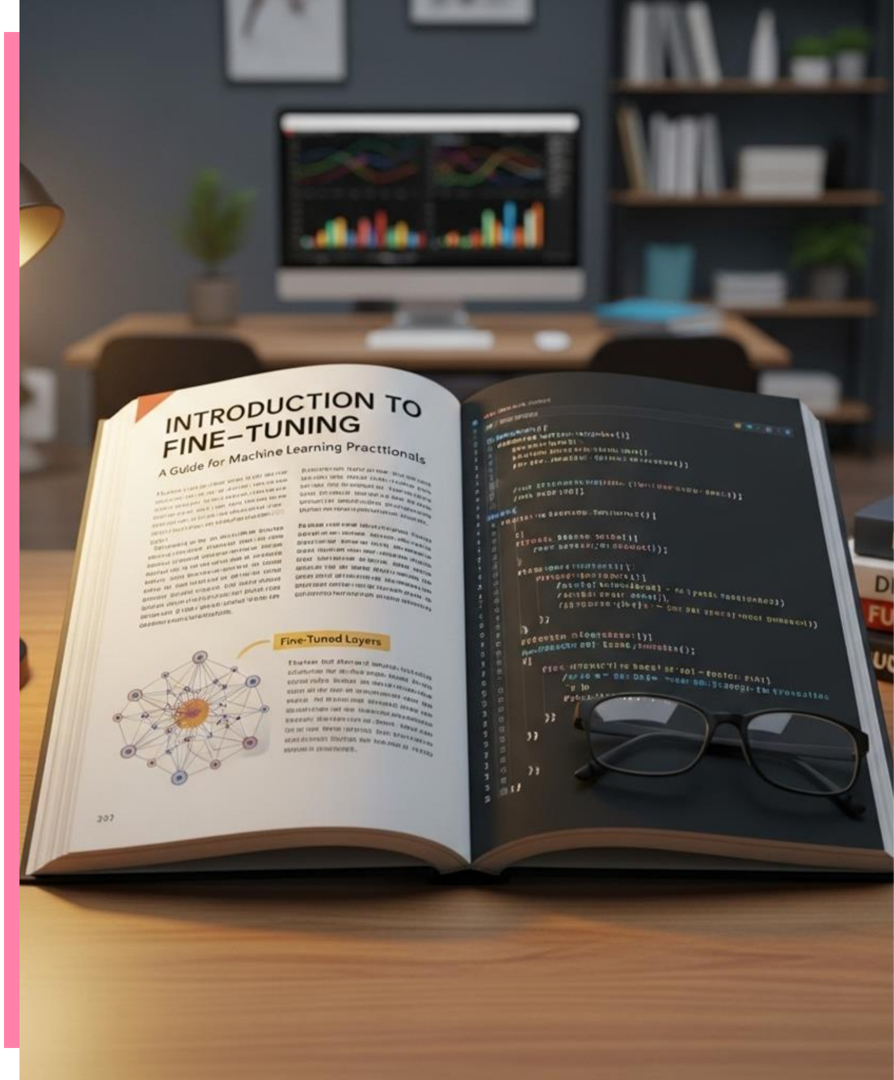
- Define the LoRA configuration (rank, alpha, dropout)
- Create the SFTTrainer with PEFT config
- Train and save the adapter weights

- from peft import PeftModel, PeftConfig, LoraConfig



Code 1: Guide to LLM Fine-tuning

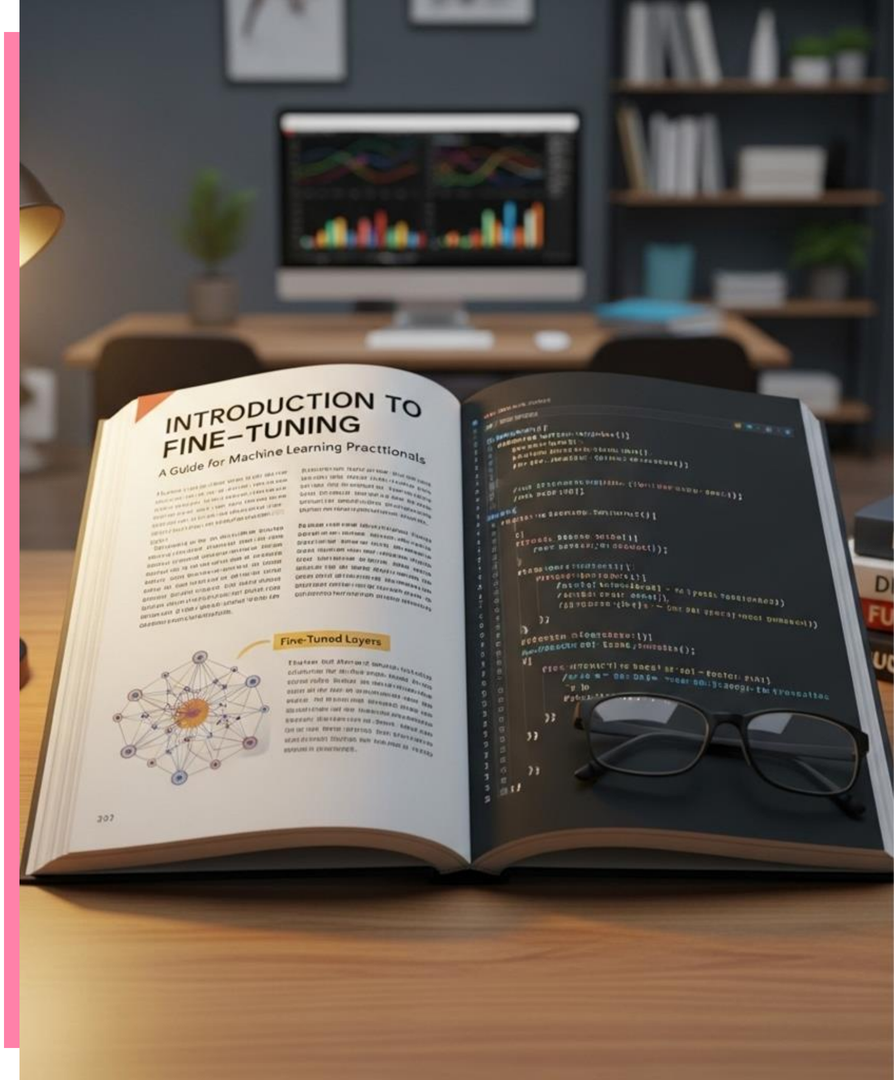
- ✓ **bitsandbytes** and **accelerate** - libraries are going to be used for quantizing a model



Code 1: Guide to LLM Fine-tuning

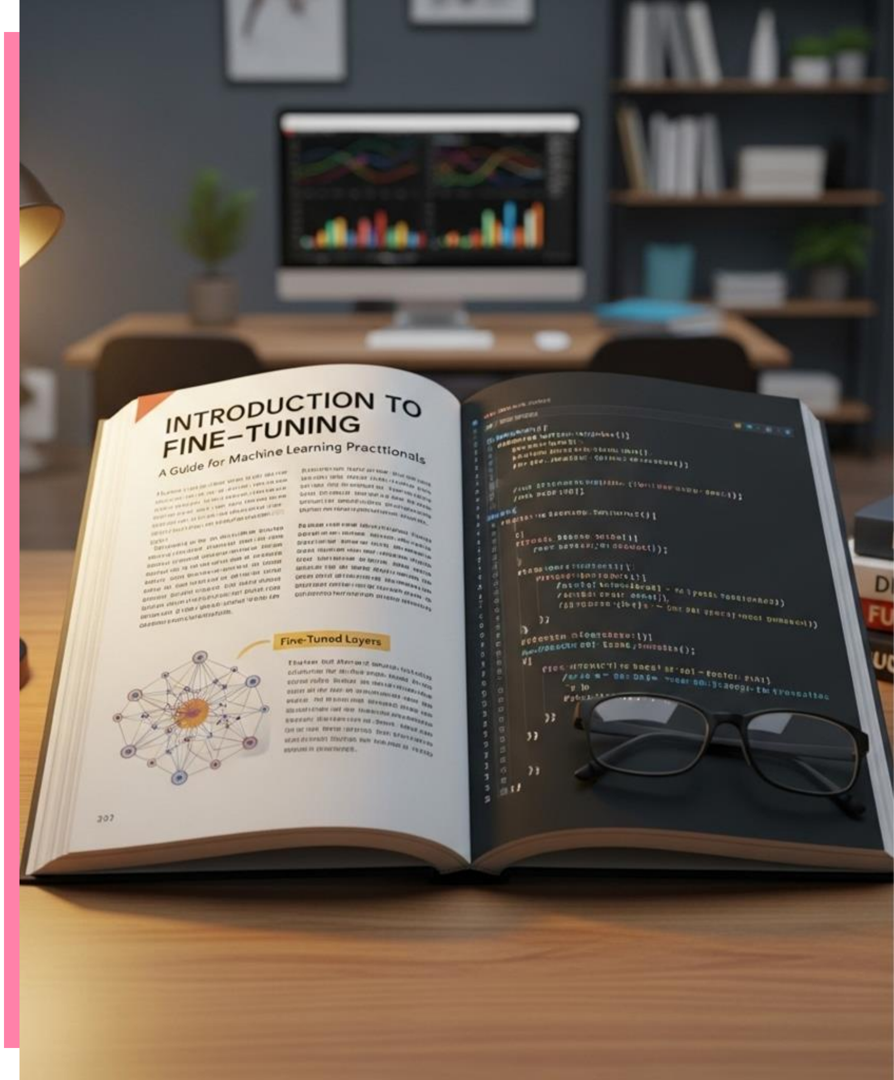
```
from peft import LoraConfig
# r: rank dimension for LoRA update matrices (smaller =
more compression)
rank_dimension = 6
# lora_alpha: scaling factor for LoRA layers (higher =
stronger adaptation)
lora_alpha = 8
# lora_dropout: dropout probability for LoRA layers (helps
prevent overfitting)
lora_dropout = 0.05

peft_config = LoraConfig(
    r=rank_dimension,  # Rank dimension - typically 4-32
    lora_alpha=lora_alpha,  # LoRA SF - typically 2x rank
    lora_dropout=lora_dropout,  # Dropout probability
    bias="none",  # Bias type for LoRA. the corresponding
biases will be updated during training
    target_modules="all-linear",  # Which modules to apply
LoRA to
    task_type="CAUSAL_LM",  # type for model architecture
)
```



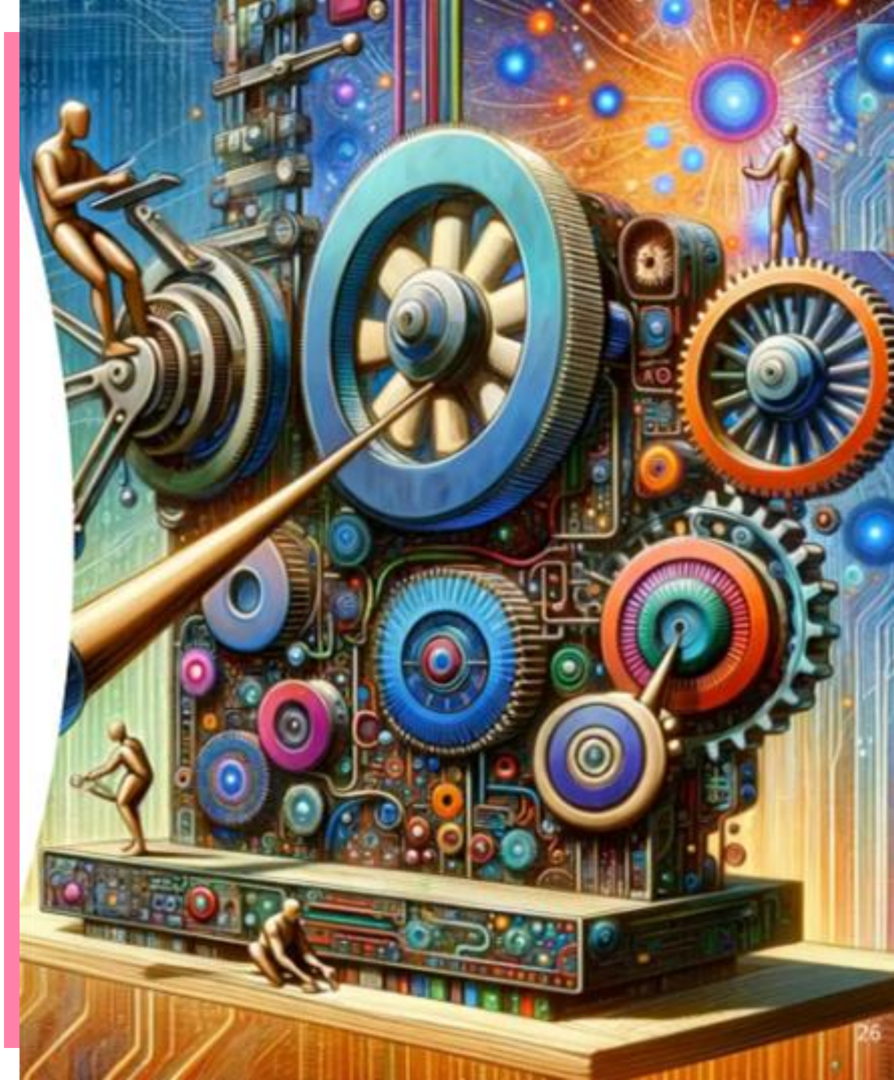
Code 1: Guide to LLM Fine-tuning

```python
# Create SFTTrainer with LoRA configuration
trainer = SFTTrainer(
    model=model,
    args=args,
    train_dataset=dataset["train"],
    peft_config=peft_config,  # LoRA configuration
    max_seq_length=max_seq_length,  # Maximum sequence length
    processing_class=tokenizer,
)
```



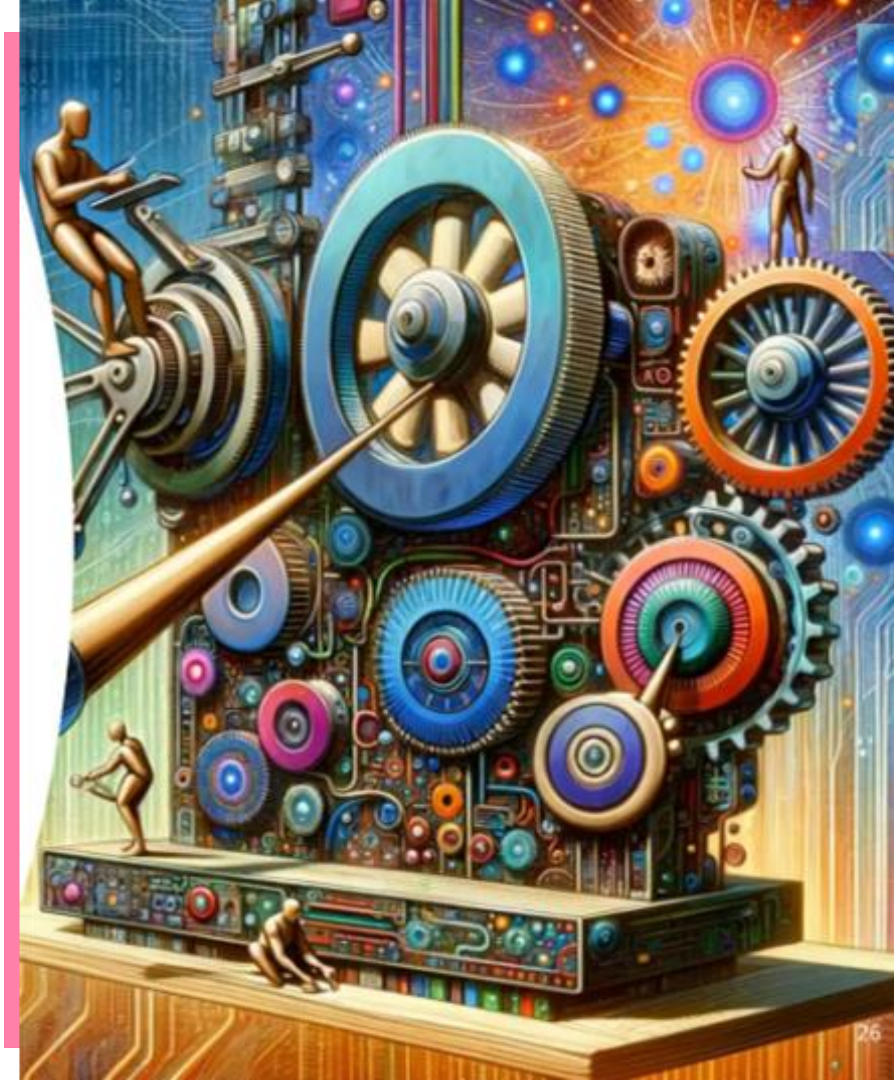
LoraConfig

🔷 1. r=rank_dimensionMeaning: Low-rank dimension rr that controls the size of LoRA matrices AA and BB.Typical range: 4 to 32.Effect:Small r → fewer trainable params, smaller memory footprint, but possibly lower accuracy.Large r → better capacity to adapt but higher cost.Examples:Small tasks (classification, adapters for small datasets): r=4–8.Complex tasks (dialog, summarization): r=16–32.Very large models (65B params): sometimes r=64 or more.
🔷 2. lora_alphaMeaning: Scaling factor applied to the low-rank update BABA.Typical rule: Usually 2 × r, but values up to 128–256 are used.Effect:Larger alpha → stronger influence of LoRA update.Smaller alpha → more conservative adjustment.Examples:If r=8, then lora_alpha=16.If r=16, then lora_alpha=32.Hugging Face examples often use lora_alpha=16, 32, 64.
🔷 3. lora_dropoutMeaning: Dropout applied to LoRA during training for regularization.Typical values: 0.0 – 0.1.Effect:0.0 if dataset is large and you don’t want to regularize.0.05–0.1 if dataset is small to avoid overfitting.



LoraConfig

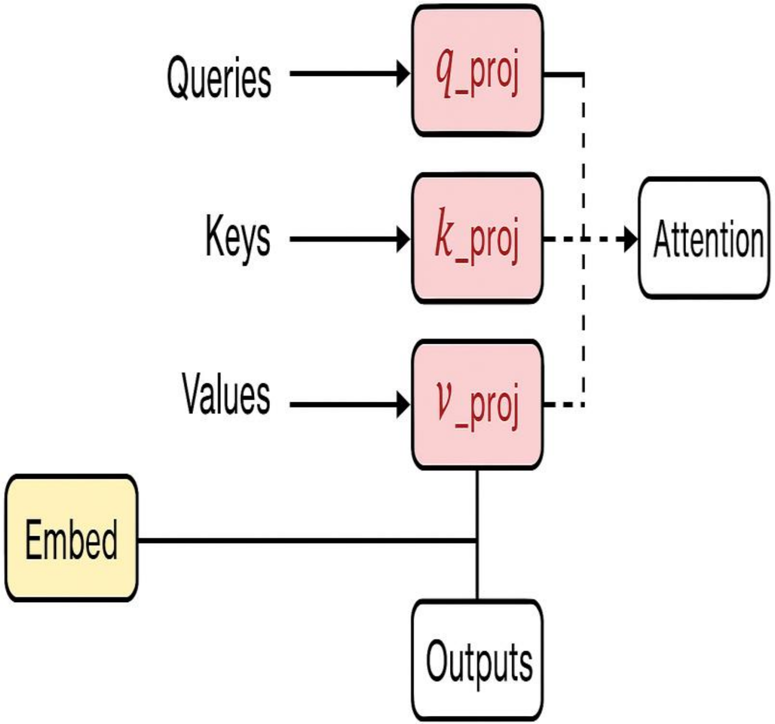
🔷 4. bias : Whether to update biases in the target modules.Options:"none" → don’t update biases (default, efficient)."all" → train all biases."lora_only" → train only biases corresponding to LoRA modules.Most common: "none" for efficiency.
🔷 5. target_modules : Which modules (layers) to inject LoRA adapters into.Values:"all-linear" (default for Transformers) → applies to all linear layers.You can specify names like ["q_proj", "v_proj"] (common in attention layers).Tradeoff:Fewer target modules = smaller model, faster training, but less adaptation power.More modules = better task performance but more compute.
🔷 6. task_type : Model type for PEFT to configure properly.Examples:"CAUSAL_LM" → GPT-style models."SEQ_CLS" → sequence classification."TOKEN_CLS" → token-level classification (NER, POS tagging)."SEQ_2_SEQ_LM" → seq2seq (BART, T5, etc.).



LoraConfig

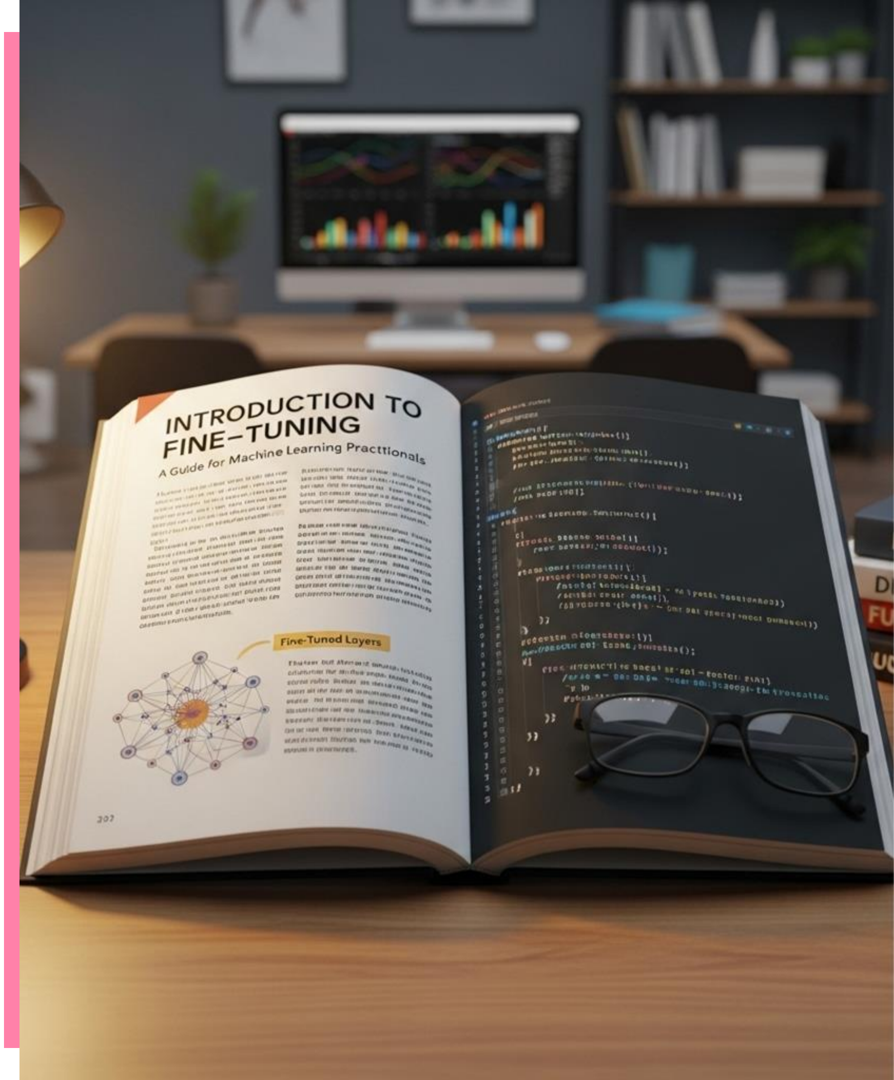
- ✓ q_proj
- ✓ v_proj
- ✓ k_proj
- ✓ o_proj
- ✓ gate_proj
- ✓ down_proj
- ✓ up_proj

LoRA in Multi-Head Attention



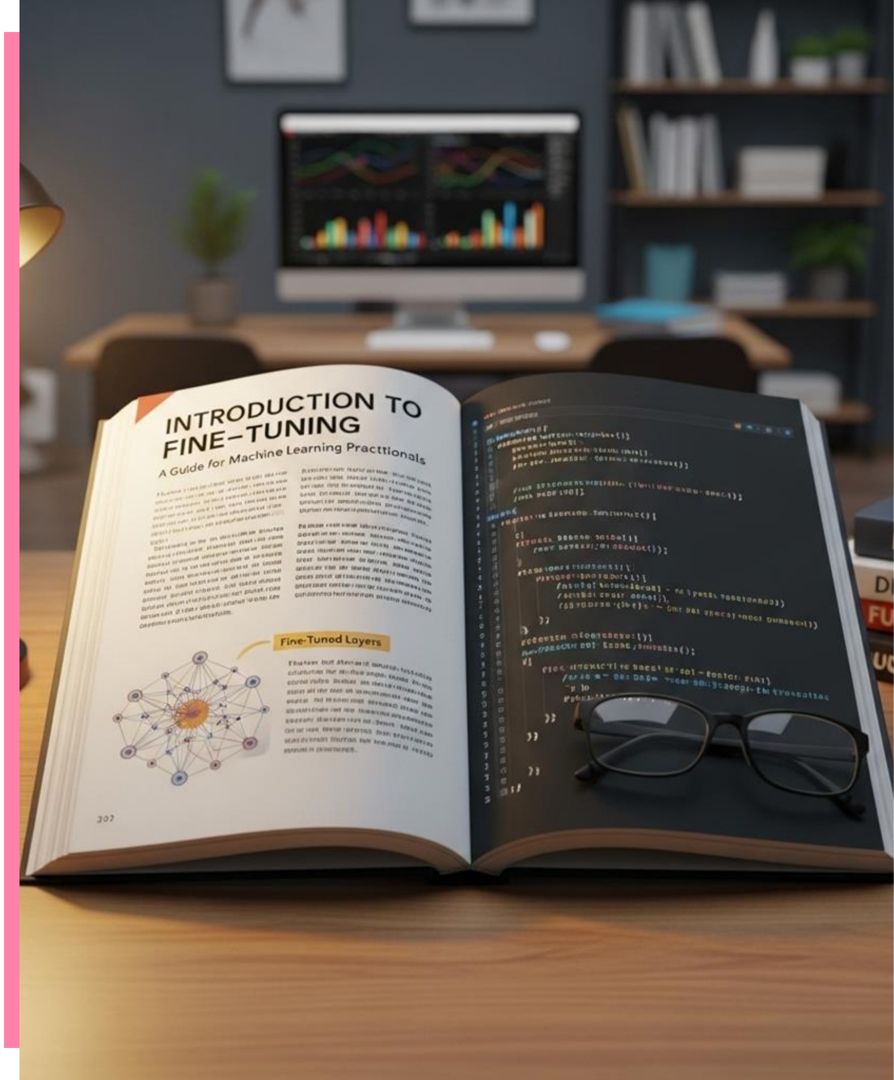
Code Demo 2: Fine-tuning LLM

- The notebook demonstrates how to **fine-tune a large language model (Qwen 1.5B Instruct)** using **LoRA (Low-Rank Adaptation)** with **4-bit quantization (QLoRA)** so that it runs efficiently on limited hardware.
- The fine-tuned model is saved locally.
- Save model on HuggingFace (HF)
- Load model from HF and use it.



Code Demo 3: Fine-tuning LLM

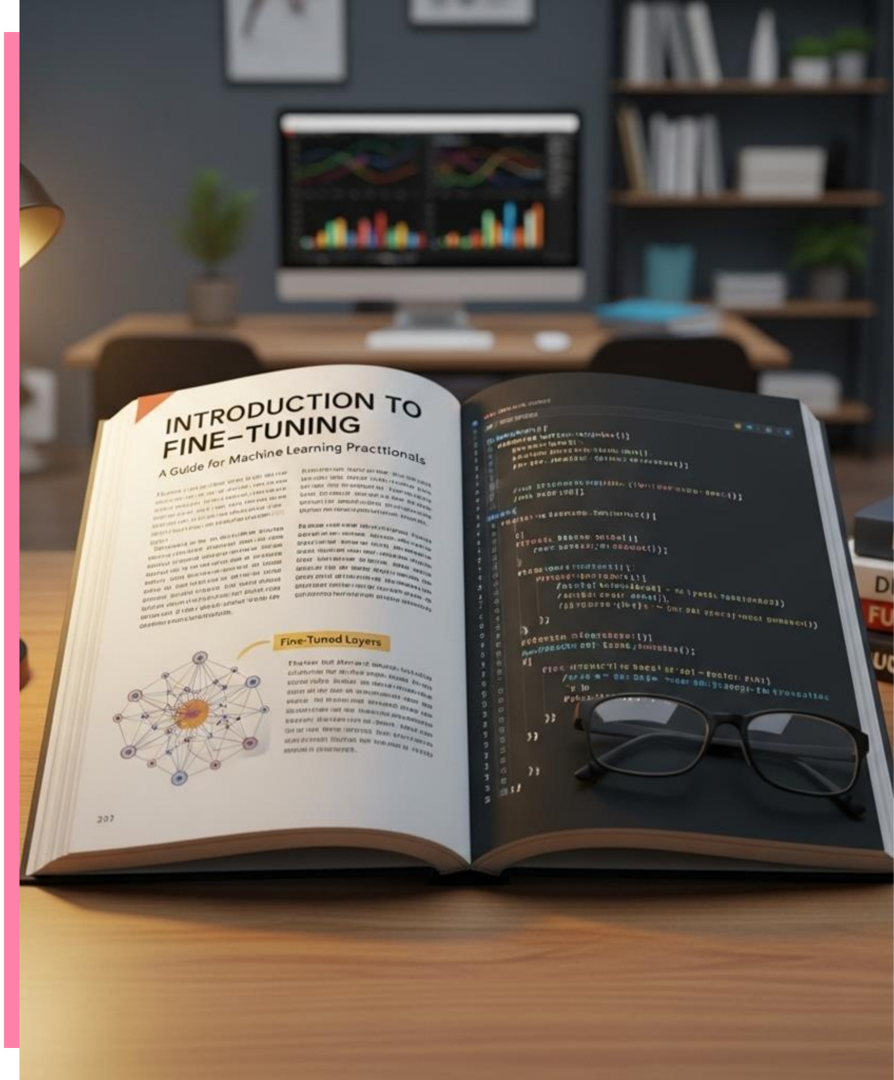
- Data split - training and evaluation (85-15)
- Change the LR
- Train + Eval losses
- Use max_steps
- Draw loss curves to see progress



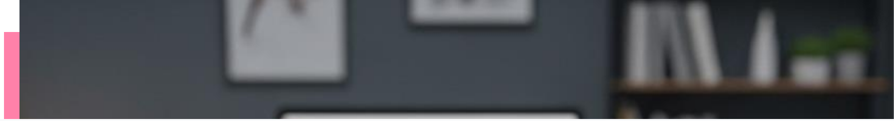
Code Demo 2 & 3: Fine-tuning LLM

- Source Code Python Notebook: https://drive.google.com/file/d/1ZTG7Or3L2adjH9uzU9w4YiQBRNKqSAld/view?usp=sharing

Perform Code Execution



Code 1: Guide to LLM Fine-tuning



After training a LoRA adapter, you can merge the adapter weights back into the base model. >>>

```python
import torch
from transformers import AutoModelForCausalLM
from peft import PeftModel

# 1. Load the base model
base_model = AutoModelForCausalLM.from_pretrained(
    "base_model_name", torch_dtype=torch.float16, device_map="auto"
)

# 2. Load the PEFT model with adapter
peft_model = PeftModel.from_pretrained(
    base_model, "path/to/adapter", torch_dtype=torch.float16
)

# 3. Merge adapter weights with base model
merged_model = peft_model.merge_and_unload()
```

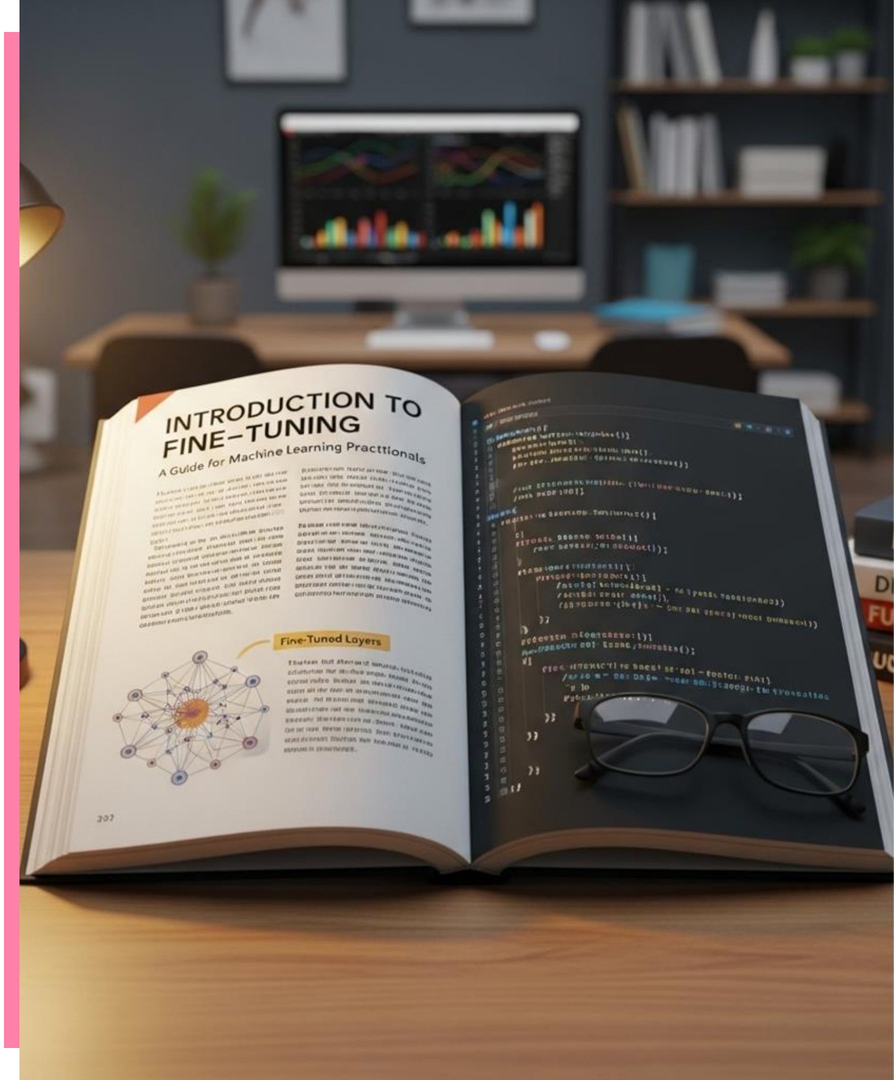
Code 1: Guide to LLM Fine-tuning

Saving the model

```
# Save both model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("base_model_name")

merged_model.save_pretrained("path/to/save/merged_model")

tokenizer.save_pretrained("path/to/merged_model")
```



Code 1: Guide to LLM Fine-tuning

Step: Import following libraries

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig
import bitsandbytes as bnb
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training, PeftModel, PeftConfig
from datasets import load_dataset
from transformers import TrainingArguments, pipeline
from trl import SFTTrainer
```

Code 1

Step:
Load a model and tokenizer

Model Chosen:
LLama2 - 7B

```
[ ] from huggingface_hub import login
    login()
```

```
[ ] repo_id = "meta-llama/Llama-2-7b-chat-hf" # Modify to whatever model you want to use

    base_model = AutoModelForCausalLM.from_pretrained(
        repo_id,
        device_map='auto',
        load_in_8bit=True,
        trust_remote_code=True,
    )

    tokenizer = AutoTokenizer.from_pretrained(repo_id)
    tokenizer.add_special_tokens({'pad_token': '[PAD]'})
    tokenizer.pad_token = tokenizer.eos_token

    base_model.config.use_cache = False
```

```
[ ] print(base_model) # use it to check what target module should be
```

```
[ ] base_model.get_memory_footprint() # Check the memory
```

Code 1: Guide to LLM Fine-tuning

Step:
Load a model and tokenizer

Helper function to see how many parameters the model has:

```
[ ]  def print_trainable_parameters(model):
         """
         Prints the number of trainable parameters in the model.
         """
         trainable_params = 0
         all_param = 0
         for _, param in model.named_parameters():
             all_param += param.numel()
             if param.requires_grad:
                 trainable_params += param.numel()
         print(
             f"trainable params: {trainable_params} || all params: {all_param} || trainable%: {100 * trainable_params / all_param:.2f}"
         )
```

Code 1: Gu

Step:
Test the base model

```
device = "cuda:0"

def user_prompt(human_prompt):
    # This has to change if your dataset isn't formatted as Alpaca
    prompt_template=
    f"""
    ### HUMAN:\n{human_prompt}
    \n\n
    ### RESPONSE:\n
    """
    return prompt_template

pipe = pipeline(
    task="text-generation",
    model=base_model,
    tokenizer=tokenizer,
    max_length=150,
    repetition_penalty=1.15,
    top_p=0.95
    )
result = pipe(user_prompt("""You are an expert youtuber.
Give me some ideas for a youtube title for a video about fine tuning LLM"""))
print(result[0]['generated_text'])
```

Code 1: Guide to LLM Fine-tuning

Step:
Prepare LoRA and preprocess the model for training

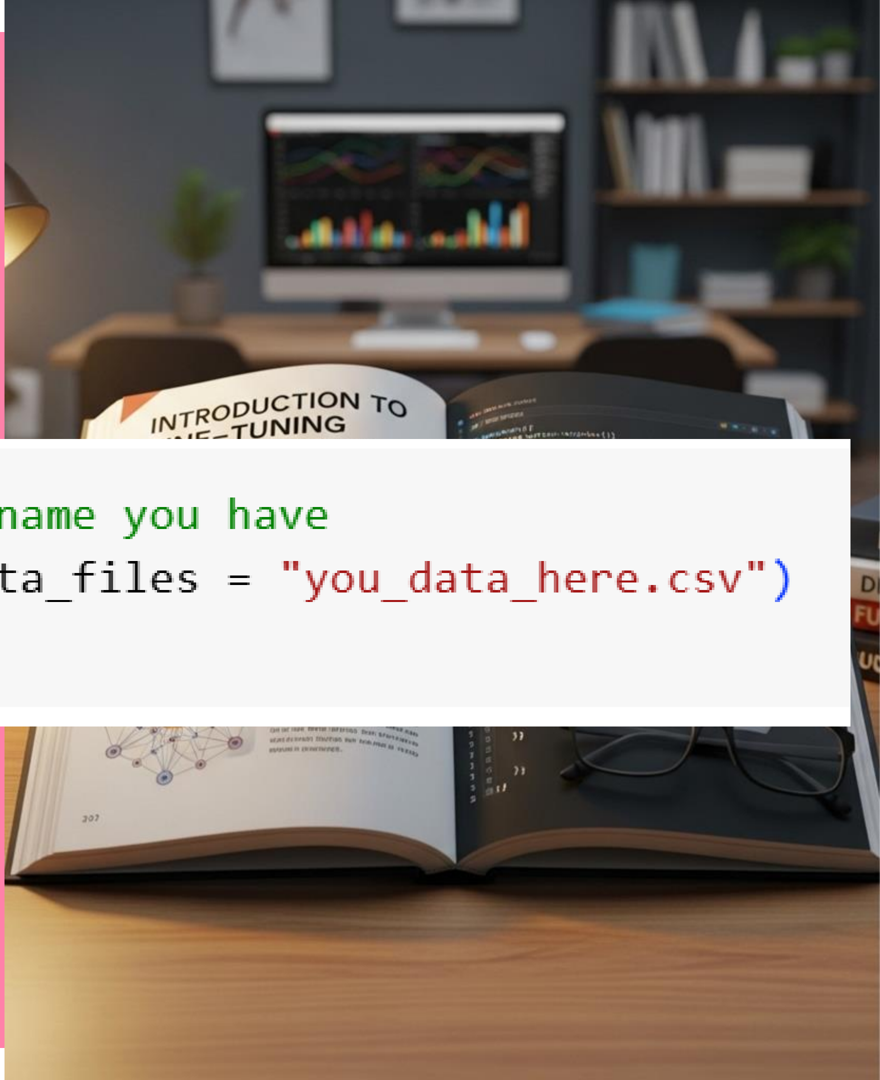
```
[ ]  config = LoraConfig(
         r=8,
         lora_alpha=32,
         # you have to know the target modules, it varies from model to model
         target_modules=["q_proj", "v_proj", "k_proj", "o_proj"],
         lora_dropout=0.05,
         bias="none",
         task_type="CAUSAL_LM"
     )

     # Wrap the base model with get_peft_model() to get a trainable PeftModel
     model = get_peft_model(base_model, config)
     print_trainable_parameters(model)
```

Code 1: Guide to LLM Fine-tuning

Step:

- ✓ Load a dataset from datasets library




```
[ ]  # substitute with whatever file name you have
     dataset = load_dataset("csv", data_files = "you_data_here.csv")
     print("Dataset loaded")
```

Code 1: G

Step:

- Training step

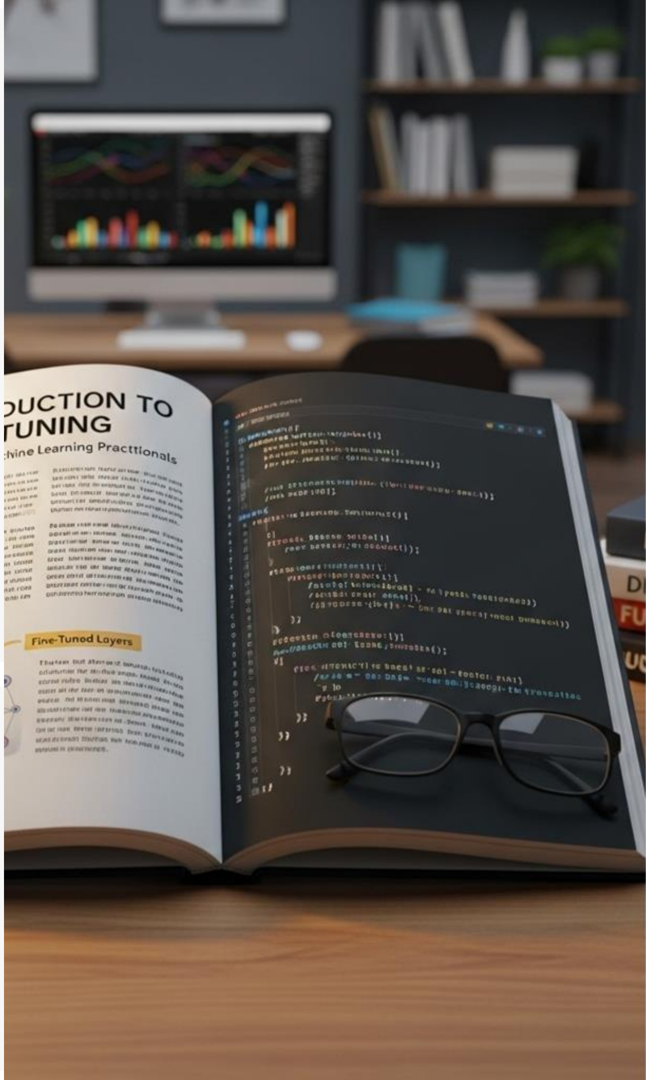
Model Chosen:
LLama2 - 7B

```
[ ]  adam_bits = 8

     training_arguments = TrainingArguments(
         output_dir = "Trainer_output",
         per_device_train_batch_size = 1,
         gradient_accumulation_steps = 4,
         run_name=f"deb-v2-xl-{adam_bits}bitAdam",
         logging_steps = 20,
         learning_rate = 2e-4,
         fp16=True,
         max_grad_norm = 0.3,
         max_steps = 300,
         warmup_ratio = 0.03,
         group_by_length=True,
         lr_scheduler_type = "constant",
     )
```

```
[ ]  trainer = SFTTrainer(
         model = model,
         train_dataset = dataset["train"],
         dataset_text_field="text",
         args = training_arguments,
         max_seq_length = 512,
     )

     trainer.train()
```



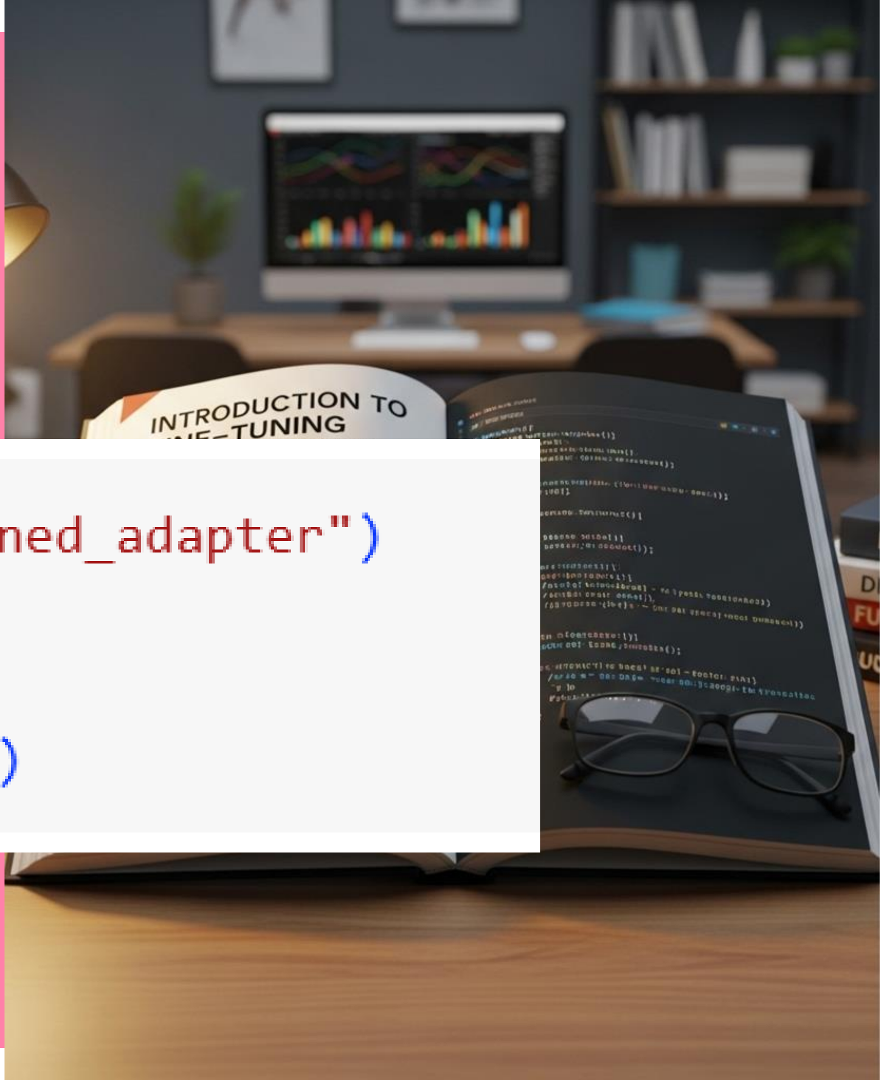
Code 1: Guide to LLM Fine-tuning

Step:

- ✓ Merge the base model and the adapter

```
[ ]  trainer.save_model("Finetuned_adapter")
     adapter_model = model

     print("Lora Adapter saved")
```



Code 1: Guide to LLM Fine-tu

Step:

- Merge the base model and the adapter

```
[ ] # Can't merge the 8 bit/4 bit model with Lora so reload it

    repo_id = "meta-llama/Llama-2-7b-chat-hf"
    use_ram_optimized_load=False

    base_model = AutoModelForCausalLM.from_pretrained(
        repo_id,
        device_map='auto',
        trust_remote_code=True,
    )

    base_model.config.use_cache = False
```

```
[ ] base_model.get_memory_footprint()
```

```
[ ] # Load Lora adapter
    model = PeftModel.from_pretrained(
        base_model,
        "/content/Finetuned_adapter",
        )
    merged_model = model.merge_and_unload()

    merged_model.save_pretrained("/content/Merged_model")
    tokenizer.save_pretrained("/content/Merged_model")
```

Code 1: Gui

Step:
✓ Testing out Fine-Tuned model

```
device = "cuda:0"

def user_prompt(human_prompt):
    prompt_template=f"### HUMAN:\n{human_prompt}\n\n### RESPONSE:\n"
    return prompt_template

pipe = pipeline(
    task="text-generation",
    model=merged_model,
    tokenizer=tokenizer,
    max_length=150,
    repetition_penalty=1.15,
    top_p=0.95
    )
result = pipe(user_prompt("""You are an expert youtuber. Give me some
 ideas for a youtube title for a video about fine tuning LLM"""))
print(result[0]['generated_text'])
```

```
[ ] merged_model.push_to_hub("your_hg_id/name_fine_tuned_model")
```

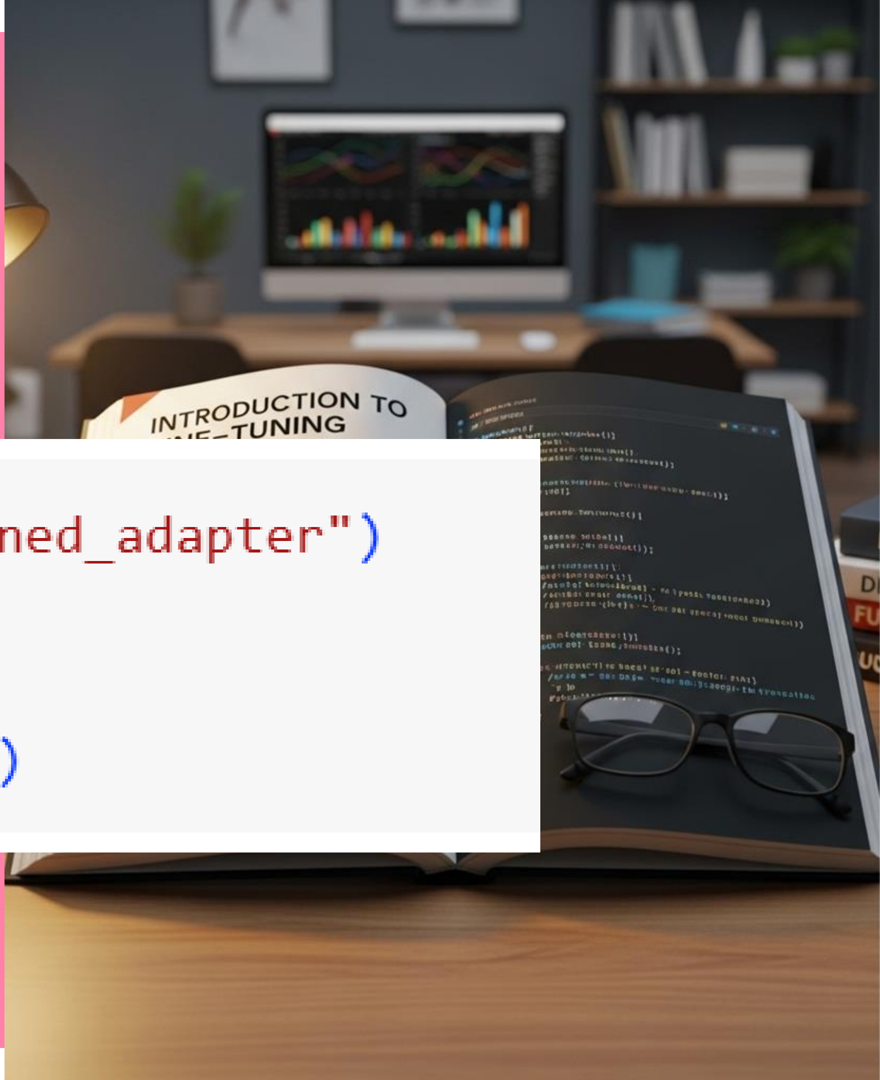
Code 1: Guide to LLM Fine-tuning

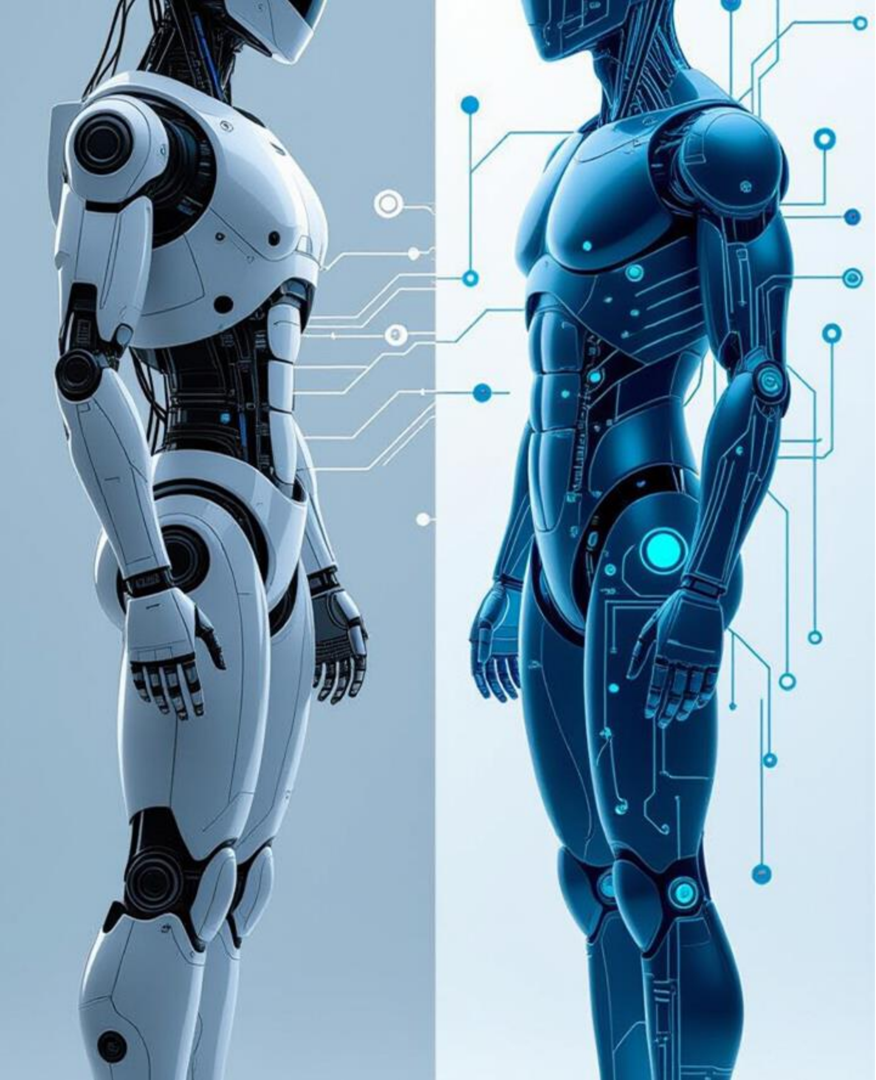
Step:

- ✓ Save the adapter

```
[ ]  trainer.save_model("Finetuned_adapter")
     adapter_model = model

     print("Lora Adapter saved")
```





Code Result

See the notes below

Configure Your Hugging Face Access Token in Colab Environment

- https://pyimagesearch.com/2025/04/04/configure-your-hugging-face-access-token-in-colab-environment/