Complete Guide to LLMs and Fine-tuning: Medical Chatbot Project

Table of Contents

What are Large Language Models (LLMs)?

Definition

Large Language Models are artificial intelligence systems trained on vast amounts of text data to understand and generate human-like text. They learn patterns, relationships, and structures in language to perform various tasks like:

- **Text Generation**: Creating coherent, contextually relevant text
- **Question Answering**: Providing informative responses to queries
- **Conversation**: Engaging in human-like dialogue
- **Text Completion**: Finishing incomplete sentences or documents
- **Language Translation**: Converting text between languages

Deep Dive: How LLMs Actually Work

1. The Foundation: Neural Networks and Transformers

Traditional Neural Networks vs. Transformers

Before transformers, language models used:

- **Recurrent Neural Networks (RNNs)**: Processed text sequentially, word by word
- **Long Short-Term Memory (LSTM)**: Better at remembering long sequences
- **Problem**: Sequential processing was slow and struggled with long-range dependencies

The Transformer Revolution (2017) The "Attention is All You Need" paper introduced transformers that:

- Process entire sequences simultaneously (parallel processing)
- Use attention mechanisms to understand relationships between any two words
- Scale much more effectively with increased model size and data

Architecture Deep Dive

1. Input Processing

```
Text: "The cat sat on the mat"
↓
Tokenization: ["The", "cat", "sat", "on", "the", "mat"]
↓
Token IDs: [1234, 5678, 9012, 3456, 1234, 7890]
↓
Embeddings: 768-dimensional vectors for each token
```

2. Positional Encoding Since transformers process all tokens simultaneously, they need to know word order:

```
# Simplified positional encoding
pos_encoding[pos][2i] = sin(pos/10000^(2i/d_model))
pos_encoding[pos][2i+1] = cos(pos/10000^(2i/d_model))
```

3. Multi-Head Attention Mechanism The core innovation that allows understanding context:

```
Query (Q): "What am I looking for?"
Key (K): "What information do I have?"
Value (V): "What is the actual information?"

Attention(Q,K,V) = softmax(QK^T/√d_k)V
```

Example: In "The cat sat on the mat", when processing "sat":

- **Query**: "sat" asks "what subject am I related to?"
- **Keys**: All words offer their relevance
- **Attention scores**: "cat" gets high score, "the" gets lower score
- **Output**: Weighted combination emphasizing "cat"

4. Multi-Head Attention Instead of one attention mechanism, use multiple "heads":

- Head 1: Focuses on subject-verb relationships
- Head 2: Focuses on spatial relationships ("on the mat")
- Head 3: Focuses on semantic similarity
- Combined: Rich understanding of sentence structure

5. Feed-Forward Networks After attention, each position goes through:

$$FFN(x) = ReLU(xW_1 + b_1)W_2 + b_2$$

This allows the model to:

- Transform attention outputs
- Add non-linearity
- Capture complex patterns

6. Layer Normalization and Residual Connections

```
output = LayerNorm(x + Attention(x))
output = LayerNorm(output + FFN(output))
```

These ensure:

- Stable training of deep networks
- Information preservation across layers
- Gradient flow optimization

2. Training Process: From Raw Text to Intelligence

Phase 1: Pre-training (Self-Supervised Learning)

Data Collection

- **Scale**: Trillions of tokens from diverse sources
- **Sources**: Books, Wikipedia, news articles, web pages, academic papers
- **Preprocessing**: Cleaning, deduplication, filtering for quality

Next Token Prediction The fundamental task during pre-training:

```
Input:  "The weather is"
Target: "nice"
Loss:   Cross-entropy between predicted probability distribution and actual next word
```

What the Model Learns Through this simple task, the model discovers:

- **Grammar**: Subject-verb agreement, sentence structure
- **Facts**: "Paris is the capital of France"
- **Reasoning**: Cause and effect relationships
- **Common Sense**: "Ice is cold", "Fire is hot"
- **Patterns**: Writing styles, formats, conversational patterns

Training Dynamics

- **Forward Pass**: Input → Embeddings → Transformer Layers → Predictions
- **Loss Calculation**: Compare predictions with actual next tokens
- **Backward Pass**: Calculate gradients using backpropagation
- **Parameter Update**: Adjust billions of parameters using optimizers like Adam

Phase 2: Scaling Laws and Emergent Abilities

Scaling Laws Research shows predictable relationships:

- **Model Size** (parameters): 7B → 13B → 70B → 175B
- **Data Size**: More diverse, high-quality data improves performance
- **Compute**: Training time scales with model and data size

Emergent Abilities As models scale, new capabilities emerge:

- **Few-shot Learning**: Learn tasks from just examples
- **Chain-of-Thought**: Step-by-step reasoning
- **Code Generation**: Writing functional programs
- **Mathematical Reasoning**: Solving complex problems

3. From Pre-training to Useful AI: The Alignment Problem

The Raw Pre-trained Model

A fresh pre-trained model:

- Completes any text continuation
- No distinction between helpful/harmful content
- No conversational abilities
- No task-specific optimization

Instruction Tuning (Supervised Fine-tuning)

Transform the model into a helpful assistant:

Data Format:

```
{
  "instruction": "Explain photosynthesis",
  "input": "",
  "output": "Photosynthesis is the process by which plants..."
}
```

Training Process:

1. Convert instructions to chat format
2. Train model to follow instructions
3. Teach specific response patterns
4. Optimize for helpfulness and accuracy

Reinforcement Learning from Human Feedback (RLHF)

Further align the model with human preferences:

1. **Reward Model Training**:
    - Humans rank multiple model responses
    - Train a reward model to predict human preferences
    - Learn what makes responses "good" or "bad"
2. **Policy Optimization**:
    - Use reinforcement learning (PPO - Proximal Policy Optimization)
    - Model generates responses
    - Reward model scores them
    - Update model to generate higher-scoring responses

4. Memory and Context Understanding

Context Window

- **Definition**: Maximum number of tokens the model can process at once
- **Sizes**: GPT-3 (4K), GPT-4 (8K-32K), Claude (100K-200K), GPT-4 Turbo (128K)
- **Challenge**: Attention computation scales quadratically with context length

How Models "Remember"

LLMs don't have persistent memory like humans:

- **Within Context**: Perfect recall of everything in current conversation
- **Between Conversations**: No memory of previous interactions
- **Knowledge**: Fixed at training time, encoded in parameters

Attention Patterns

Different layers learn different types of relationships:

- **Early Layers**: Syntax, basic word relationships
- **Middle Layers**: Semantic understanding, entity recognition
- **Late Layers**: Complex reasoning, task-specific patterns

5. Popular LLM Architectures

GPT (Generative Pre-trained Transformer) - OpenAI

- **Architecture**: Decoder-only transformer
- **Training**: Next token prediction
- **Strengths**: Text generation, completion

- **Evolution**: GPT-1 (117M) → GPT-2 (1.5B) → GPT-3 (175B) → GPT-4 (unknown, likely >1T)

LLaMA (Large Language Model Meta AI) - Meta

- **Innovation**: More efficient training, better performance per parameter
- **Sizes**: 7B, 13B, 30B, 65B parameters
- **Features**: RMSNorm, SwiGLU activation, rotary positional embeddings
- **Open Source**: Weights available for research (LLaMA 2 more permissive)

BERT (Bidirectional Encoder Representations) - Google

- **Architecture**: Encoder-only transformer
- **Training**: Masked language modeling + next sentence prediction
- **Strengths**: Understanding tasks (classification, Q&A)
- **Limitation**: Not designed for text generation

T5 (Text-to-Text Transfer Transformer) - Google

- **Innovation**: Everything as text-to-text
- **Training**: Span corruption (predicting missing text spans)
- **Flexibility**: Single architecture for all NLP tasks

6. Computational Requirements

Training Costs

- **GPT-3**: ~$4.6 million in compute costs
- **Hardware**: Thousands of high-end GPUs (V100, A100)
- **Time**: Months of continuous training
- **Energy**: Equivalent to hundreds of homes' annual electricity usage

Inference Costs

- **Memory**: 175B parameter model needs ~350GB RAM (FP16)
- **Compute**: Each token generation requires full forward pass
- **Optimization**: Techniques like quantization, pruning, distillation

7. Emergent Behaviors and Capabilities

Zero-shot Learning

Model performs tasks it wasn't explicitly trained for:

```
Input: "Translate to French: Hello world"
Output: "Bonjour le monde"
```

Few-shot Learning

Learning from just a few examples:

```
Input:
"English: Hello, French: Bonjour
English: Goodbye, French: Au revoir
English: Thank you, French:"
Output: "Merci"
```

Chain-of-Thought Reasoning

Breaking down complex problems:

```
Input: "Roger has 5 tennis balls. He buys 2 more cans with 3 balls each. How many does he have?"
Output:
"Let me think step by step:
- Roger starts with 5 tennis balls
- He buys 2 cans with 3 balls each: 2 × 3 = 6 balls
- Total: 5 + 6 = 11 tennis balls"
```

In-context Learning

Using the context window as a temporary "memory":

- Provide examples and instructions within the prompt
- Model adapts its behavior based on context
- No parameter updates needed

Understanding Fine-tuning

What is Fine-tuning?

Fine-tuning is the process of adapting a pre-trained language model to perform better on specific tasks or domains. Instead of training a model from scratch (which requires enormous computational resources), we start with a model that already understands language and teach it specialized knowledge.

Why Fine-tune?

1. **Domain Specialization**: Make the model expert in specific fields (medical, legal, technical)
2. **Task Adaptation**: Optimize for particular use cases (customer service, coding assistance)
3. **Resource Efficiency**: Much faster and cheaper than training from scratch
4. **Performance**: Often achieves better results than general-purpose models on specialized tasks

Types of Fine-tuning

1. Full Fine-tuning

- Updates all parameters in the model
- Requires significant computational resources
- Provides maximum customization

2. Parameter-Efficient Fine-tuning (PEFT)

- Updates only a small subset of parameters
- Much more resource-efficient
- Techniques include:
    - **LoRA (Low-Rank Adaptation)** : Adds small adapter layers
    - **Prefix Tuning**: Optimizes input prefixes
    - **Adapter Layers**: Inserts small neural networks between existing layers

3. Quantization

- Reduces model precision (e.g., from 32-bit to 4-bit)
- Significantly reduces memory usage
- Enables training on consumer hardware

Project Overview: Medical Chatbot

Goal

Transform LLaMA 3 8B-Chat into a specialized medical assistant that can provide helpful responses to patient questions.

Dataset

- **Source**: `ruslanmv/ai-medical-chatbot`
- **Size**: 250k dialogues between patients and doctors
- **Format**: Patient questions paired with doctor responses
- **Sample Size**: 1000 conversations (for demo purposes)

Technical Approach

1. **Base Model**: LLaMA 3 8B-Chat (conversational AI model)
2. **Fine-tuning Method**: LoRA (Low-Rank Adaptation)
3. **Quantization**: 4-bit precision with QLoRA
4. **Framework**: Hugging Face Transformers + PEFT + TRL

Expected Outcome

A medical chatbot that can:

- Understand patient symptoms and concerns
- Provide medically-informed responses
- Maintain conversational flow
- Demonstrate specialized medical knowledge

Step-by-Step Code Walkthrough

Cell 1: Project Introduction

```
# Finetuning Llama 3
We'll fine-tune the Llama 3 8B-Chat model using the ruslanmv/ai-medical-chatbot dataset.
The dataset contains 250k dialogues between a patient and a doctor.
```

Purpose: Sets up the project context and explains the objective.

Cell 2: Environment Setup

```
/kaggle/input/llama-3/transformers/8b-chat-hf/1
GPU P100
huggingface_token and wandb to be active
```

Purpose: Documents the computational environment and requirements:

- **Kaggle**: Cloud platform providing GPU access
- **GPU P100**: NVIDIA Tesla P100 for training acceleration
- **Tokens**: Authentication for Hugging Face Hub and Weights & Biases

Cell 3: Basic Imports

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
```

Purpose: Imports fundamental data science libraries for numerical computing and data manipulation.

Cell 4: Suppress Warnings

```
import warnings
warnings.filterwarnings('ignore')
```

Purpose: Reduces output noise by hiding non-critical warning messages during execution.

Cell 5: Install Required Libraries

```
%%capture
%pip install -U transformers
%pip install -U datasets
%pip install -U accelerate
%pip install -U peft
%pip install -U trl
%pip install -U bitsandbytes
%pip install -U wandb
```

Purpose: Installs essential libraries for the project:

- **transformers**: Hugging Face library for pre-trained models
- **datasets**: Data loading and processing utilities
- **accelerate**: Distributed training and optimization
- **peft**: Parameter-efficient fine-tuning methods
- **trl**: Transformer Reinforcement Learning
- **bitsandbytes**: 8-bit optimization and quantization
- **wandb**: Experiment tracking and visualization

Cell 6: Key Imports

```
from transformers import (
    AutoModelForCausalLM,      # For loading language models
    AutoTokenizer,             # For text tokenization
    BitsAndBytesConfig,        # For quantization configuration
    TrainingArguments,         # For training parameters
    pipeline, logging,         # Utilities
)
from peft import (
    LoraConfig,                # LoRA configuration
    get_peft_model,            # Apply PEFT to model
)
import os, torch, wandb        # System, PyTorch, experiment tracking
from datasets import load_dataset  # Dataset loading
from trl import SFTTrainer, setup_chat_format  # Supervised fine-tuning
```

Purpose: Imports all necessary components for:

- Model loading and configuration
- Parameter-efficient fine-tuning
- Training setup and execution
- Experiment tracking

Cell 7: Authentication Setup

```
from huggingface_hub import login
from kaggle_secrets import UserSecretsClient

# Get tokens from Kaggle secrets
user_secrets = UserSecretsClient()
hf_token = user_secrets.get_secret("HUGGINGFACE_TOKEN")
wb_token = user_secrets.get_secret("wandb")

# Login to services
login(token = hf_token)
wandb.login(key=wb_token)

# Initialize experiment tracking
run = wandb.init(
    project='Fine-tune Llama 3 8B on Medical Dataset',
    job_type="training",
    anonymous="allow"
)
```

Purpose:

- Securely accesses stored API tokens
- Authenticates with Hugging Face Hub (for model access)
- Sets up Weights & Biases for experiment tracking
- Creates a new experiment run for monitoring training progress

Cell 8: Model Configuration

```
base_model = "/kaggle/input/llama-3/transformers/8b-chat-hf/1"
dataset_name = "ruslanmv/ai-medical-chatbot"
new_model = "llama-3-8b-chat-doctor"
```

Purpose: Defines key paths and names:

- **base_model**: Path to the pre-trained LLaMA 3 model
- **dataset_name**: Medical conversation dataset identifier
- **new_model**: Name for the fine-tuned model

Cell 9: Model Parameters

```
torch_dtype = torch.float16        # Use 16-bit precision
attn_implementation = "eager"      # Attention mechanism type
```

Purpose: Sets computational parameters:

- **float16**: Reduces memory usage while maintaining performance
- **eager**: Standard attention implementation (vs. optimized variants)

Cell 10: Quantization Configuration

```
# QLoRA config
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,                    # Enable 4-bit quantization
    bnb_4bit_quant_type="nf4",           # Use NF4 quantization
    bnb_4bit_compute_dtype=torch_dtype,   # Computation precision
    bnb_4bit_use_double_quant=True,      # Double quantization for efficiency
)

# Load model with quantization
model = AutoModelForCausalLM.from_pretrained(
    base_model,
    quantization_config=bnb_config,
    device_map="auto",                    # Automatic device placement
    attn_implementation=attn_implementation
)
```

Purpose:

- **4-bit Quantization**: Reduces model size from ~32GB to ~8GB
- **NF4**: Optimal quantization method for language models
- **Double Quantization**: Further compression for efficiency
- **Auto Device Mapping**: Automatically distributes model across available GPUs

Key Concept: QLoRA (Quantized Low-Rank Adaptation) enables fine-tuning large models on consumer hardware by drastically reducing memory requirements.

Cell 11: Tokenizer Setup

```
# Load tokenizer
tokenizer = AutoTokenizer.from_pretrained(base_model)
model, tokenizer = setup_chat_format(model, tokenizer)
```

Purpose:

- Loads the tokenizer that converts text to numbers
- **setup_chat_format**: Configures the model for conversational AI using ChatML format
- Ensures model and tokenizer are compatible for chat-based interactions

Cell 12: LoRA Configuration

```
# LoRA config
peft_config = LoraConfig(
    r=16,                     # Rank of adaptation
    lora_alpha=32,           # LoRA scaling parameter
    lora_dropout=0.05,       # Dropout for regularization
    bias="none",             # Don't adapt bias terms
    task_type="CAUSAL_LM",   # Causal language modeling
    target_modules=['up_proj', 'down_proj', 'gate_proj',
                    'k_proj', 'q_proj', 'v_proj', 'o_proj']  # Which layers to adapt
)
model = get_peft_model(model, peft_config)
```

Purpose: Configures LoRA (Low-Rank Adaptation):

- **r=16**: Rank of the low-rank matrices (controls adaptation capacity)
- **lora_alpha=32**: Scaling factor for LoRA weights
- **target_modules**: Specific attention and MLP layers to adapt
- **Efficiency**: Only ~0.1% of original parameters need training

Key Concept: LoRA adds small adapter matrices to existing layers, allowing effective fine-tuning with minimal parameters.

Cell 13: Dataset Loading and Formatting

```
# Load and sample dataset
dataset = load_dataset(dataset_name, split="all")
dataset = dataset.shuffle(seed=65).select(range(1000))  # Use 1000 samples

def format_chat_template(row):
    row_json = [{"role": "user", "content": row["Patient"]},
                {"role": "assistant", "content": row["Doctor"]}]
    row["text"] = tokenizer.apply_chat_template(row_json, tokenize=False)
    return row

dataset = dataset.map(format_chat_template, num_proc=4)
```

Purpose:

- Loads the medical conversation dataset
- **Sampling**: Uses 1000 conversations for quick training demo
- **Chat Formatting**: Converts patient-doctor pairs into standardized chat format
- **Parallel Processing**: Uses 4 processes for efficient data transformation

Sample Output:

```
<|im_start|>user
I have a headache and feel nauseous. What could be wrong?<|im_end|>
<|im_start|>assistant
Based on your symptoms, you might be experiencing a migraine or tension headache...
<|im_end|>
```

Cell 14: Dataset Splitting

```
dataset = dataset.train_test_split(test_size=0.1)
```

Purpose: Splits data into training (90%) and validation (10%) sets for proper evaluation.

Cell 15: Training Configuration

```
training_arguments = TrainingArguments(
    output_dir=new_model,                    # Where to save the model
    per_device_train_batch_size=1,          # Batch size per GPU
    per_device_eval_batch_size=1,           # Evaluation batch size
    gradient_accumulation_steps=2,           # Simulate larger batches
    optim="paged_adamw_32bit",              # Memory-efficient optimizer
    num_train_epochs=1,                     # Number of training epochs
    evaluation_strategy="steps",             # When to evaluate
    eval_steps=0.2,                         # Evaluate every 20% of training
    logging_steps=1,                        # Log every step
    warmup_steps=10,                        # Learning rate warmup
    learning_rate=2e-4,                     # Learning rate
    fp16=False, bf16=False,                 # Precision settings
    group_by_length=True,                   # Group similar-length sequences
    report_to="wandb"                       # Send metrics to Weights & Biases
)
```

Purpose: Defines all training hyperparameters:

- **Small Batches**: Accommodates limited GPU memory
- **Gradient Accumulation**: Simulates larger effective batch size
- **Single Epoch**: Quick training for demonstration
- **Learning Rate**: Balanced for fine-tuning (not too high/low)

Cell 16: Tokenizer Enhancement

```
# Add padding token
tokenizer.add_special_tokens({'pad_token': '[PAD]'})
model.resize_token_embeddings(len(tokenizer))
```

Purpose:

- Adds padding token for batch processing
- Updates model's vocabulary to include new token
- Ensures consistent sequence lengths in batches

Cell 17: Data Preprocessing

```
def preprocess_function(examples):
    return tokenizer(
        examples["text"],
        max_length=512,           # Maximum sequence length
        truncation=True,          # Cut off longer sequences
        padding="max_length"      # Pad shorter sequences
    )

# Apply preprocessing
tokenized_train_dataset = dataset["train"].map(preprocess_function, batched=True)
tokenized_test_dataset = dataset["test"].map(preprocess_function, batched=True)
```

Purpose:

- Converts text to numerical tokens
- **max_length=512**: Balances context and memory usage
- **Batch Processing**: Efficiently processes multiple examples
- Creates consistent-sized inputs for training

Cell 18: Trainer Setup

```
trainer = SFTTrainer(
    model=model,                        # Fine-tuned model
    train_dataset=tokenized_train_dataset,  # Training data
    eval_dataset=tokenized_test_dataset,    # Validation data
    peft_config=peft_config,            # LoRA configuration
    tokenizer=tokenizer,                # Text tokenizer
    args=training_arguments,            # Training parameters
)
```

Purpose: Creates the supervised fine-tuning trainer that orchestrates the entire training process.

Cell 19: Model Training

```
trainer.train()
```

Purpose: Executes the actual fine-tuning process:

- Updates LoRA adapter weights
- Monitors training and validation loss
- Logs metrics to Weights & Biases
- Takes approximately 30-40 minutes on GPU

Cell 20: Cleanup and Testing

```
wandb.finish()                          # End experiment tracking
model.config.use_cache = True           # Enable caching for inference

# Test the model
messages = [{
    "role": "user",
    "content": "Hello doctor, I have a bad scar on my forehead. How do I get rid of it?"
}]

prompt = tokenizer.apply_chat_template(messages, tokenize=False, add_generation_prompt=True)
inputs = tokenizer(prompt, return_tensors='pt', padding=True, truncation=True).to("cuda")
outputs = model.generate(**inputs, max_length=150, num_return_sequences=1)
text = tokenizer.decode(outputs[0], skip_special_tokens=True)

print(text.split("assistant")[1])
```

Purpose:

- Finishes experiment tracking
- Tests the fine-tuned model with a medical query
- Demonstrates the model's specialized knowledge
- Shows improved medical response quality

Cell 21: Model Saving

```
trainer.model.save_pretrained(new_model)
trainer.model.push_to_hub(new_model, use_temp_dir=False)
```

Purpose:

- Saves the LoRA adapter weights locally
- Uploads the fine-tuned model to Hugging Face Hub
- Makes the model accessible for future use

Key Concepts and Techniques

1. QLoRA (Quantized Low-Rank Adaptation)

- **Quantization**: Reduces precision from 32-bit to 4-bit
- **LoRA**: Adds small adapter layers instead of modifying all parameters
- **Benefits**: 75% memory reduction, maintains performance quality

2. Chat Formatting

- Converts raw text into conversational format
- Uses special tokens to indicate user/assistant roles
- Enables natural dialogue flow

3. Parameter-Efficient Fine-tuning

- Only trains ~0.1% of original parameters
- Dramatically reduces computational requirements
- Achieves comparable results to full fine-tuning

4. Supervised Fine-tuning (SFT)

- Uses input-output pairs for training
- Teaches model to generate appropriate responses
- Adapts general language model to specific domain

Results and Evaluation

Training Metrics

- **Training Loss**: Measures how well model fits training data
- **Validation Loss**: Indicates generalization to unseen data
- **Learning Curves**: Tracked via Weights & Biases

Expected Improvements

1. **Domain Knowledge**: Better understanding of medical terminology
2. **Response Quality**: More appropriate and helpful medical advice
3. **Conversational Flow**: Natural patient-doctor interaction style
4. **Specificity**: Tailored responses to medical queries

Model Performance

The fine-tuned model should demonstrate:

- Improved medical vocabulary usage
- Better symptom recognition and response
- More empathetic and professional tone
- Appropriate medical disclaimers and advice

Limitations and Considerations

- **Not Medical Advice**: AI responses should not replace professional medical consultation
- **Training Data Quality**: Model performance depends on dataset quality
- **Evaluation Methods**: Requires human expert evaluation for medical accuracy
- **Ethical Use**: Must be deployed responsibly with appropriate disclaimers

---

Conclusion

This project demonstrates how modern AI techniques can adapt large language models for specialized domains efficiently. By combining quantization, LoRA, and supervised fine-tuning, we can create domain-specific AI assistants that provide valuable specialized knowledge while being computationally feasible.

The medical chatbot serves as an excellent example of how LLMs can be customized for specific professional domains, though always with appropriate safeguards and human oversight for critical applications like healthcare.