From Concept to Code: A General Framework for Building a Medical Vision-Language Baseline Model

A Step-by-Step Guide to Building Your First Medical AI Model

If you're a student just entering the vibrant and complex world of medical image computing, you've likely felt the intimidating gap between understanding a high-level concept and actually building a functional deep learning model.

While specific model architectures and state-of-the-art techniques are constantly evolving, the end-to-end framework of a research project - from messy raw data to a clean, final prediction - remains remarkably consistent. It's a foundational roadmap that distills the complex process of creating a medical AI model into a clear, step-by-step framework.

We will journey through four core modules, using the FLARE25 challenge as our guiding example, to construct a complete baseline for a Medical Vision-Language Model (VLM). All the code for this tutorial is available in our **GitHub Repository**.

- The Blueprint: Project Scoping & Setup
- The Data Crucible: A General Approach to Multimodal Data
- The Training Engine: Principles of Fine-Tuning
- Beyond the Notebook: Inference & Deployment

By the end of this series, you will have a working model and also a reusable mental model for tackling your own innovative projects in the MICCAI field.

Module 1: The Blueprint — Project Scoping & Setup

A solid foundation is everything in deep learning. Before you write a single line of model code, making informed decisions about your tools and creating a clean, reproducible environment will save you countless hours of debugging and frustration down the road. Think of this module as *drawing the blueprint for your house* before you start laying bricks.

Part 1: Choosing Your Base Model

For newcomers, building a state-of-the-art model **from scratch** is not a practical approach. The real skill lies in smartly selecting and adapting existing open-source models. This is where "transfer learning" comes in — we take a

powerful, general-purpose model and fine-tune it to become a specialist in our specific medical task.

For this tutorial, we will focus on Vision-Language Models (VLMs), which are designed to understand both images and text. This makes them ideal for tasks such as Medical Visual Question Answering (VQA), where the model must analyze a medical image to answer a corresponding textual question.

Two popular and powerful open-source VLMs are:

- **LLaVA (Large Language and Vision Assistant):** A widely recognized model known for its strong performance and a great community. It's an excellent choice for many projects.
- **Qwen-VL:** A strong competitor from Alibaba Cloud. The latest version, Qwen2.5-VL, shows impressive capabilities in handling high-resolution images and performing complex reasoning tasks.
- **InternVL2:** A powerful series of models that has demonstrated state-of-the-art performance, often matching or exceeding closed-source models on various benchmarks. It excels at complex reasoning and document understanding.
- **Florence-2:** A model from Microsoft with a unified, prompt-based representation for a wide variety of vision tasks, including object detection, captioning, and segmentation, all within a single model.

For this tutorial, we will use **Qwen/Qwen2.5-VL-7B-Instruct** as our base model.

Part 2: Setting Up Your Development Environment

A clean, reproducible development environment is non-negotiable. It ensures that your code works consistently, whether on your machine or someone else's, and is a hallmark of a professional researcher.

Step 1: Isolate Your Project with a Virtual Environment Never install Python packages directly into your system's global Python installation. Always use a virtual environment to create an isolated space for each project's dependencies.

```
# Create a virtual environment named 'venv'
python -m venv venv

# Activate it (on macOS/Linux)
source venv/bin/activate
# On Windows, use:
# venv\Scripts\activate
```

Step 2: Install the Core Dependencies The requirements.txt file in the provided code lists all the necessary Python libraries. Here are the most critical

ones and what they do:

- **torch:** The fundamental library for deep learning in Python.
- **transformers:** Hugging Face's library for downloading, configuring, and using pre-trained models like Qwen-VL.
- **peft:** The Parameter-Efficient Fine-Tuning library, which allows us to use memory-saving techniques like LoRA.
- **datasets:** For loading and processing our training data.
- **bitsandbytes:** Enables the 4-bit quantization that makes training large models on consumer hardware possible.
- **trl:** The Transformer Reinforcement Learning library, which provides the `SFTTrainer` we will use to fine-tune our model.

Install them all with a single command:

```
pip install -r requirements.txt
```

Step 3: Structure Your Project Directory A well-organized project is easy to navigate and debug. Here is an example structure you may follow:

```
/your_project_folder
  dataset/                  # Your organized image and question files
  processed_data/           # The tokenized, ready-to-use dataset
  finetuned_qwenvl/         # Saved model checkpoints and final model
  logs/                     # Log files
  venv/                     # Your Python virtual environment
  prepare_data.py           # Script for data preprocessing
  finetune.py               # Script for model training
  evaluate.py               # Script for model evaluation
  requirements.txt          # List of Python dependencies
```

This structure separates raw data, processed data, model outputs, and code, which is a good practice in machine learning projects.

Step 4: Verify Your Setup Before diving into the code, it's wise to run a quick check to ensure everything is installed and configured correctly. The repository includes a test_installation.py script for this purpose, which confirms that all libraries can be imported and that the system can detect the GPU.

You can also run a simple Python command to check for CUDA availability:

```
import torch
print(f"CUDA available: {torch.cuda.is_available()}")
if torch.cuda.is_available():
    print(f"Device Name: {torch.cuda.get_device_name(0)}")
```

If this returns `True`, your GPU is ready for action.

You have now laid the groundwork for our project. You've chosen a powerful base model and set up a clean, organized, and reproducible development envi-

ronment. These initial steps are the blueprint that will guide the rest of our journey.

Module 2: The Data Crucible — A General Approach to Multimodal Data

Welcome to the crucible. Data is the lifeblood of any deep learning model, but in the medical field, it's also the source of our greatest challenges. Real-world medical datasets are rarely clean, uniform, or simple. This module focuses on transforming raw, messy data into a clean, unified, and model-ready format.

> To follow along interactively, you can use the **Google Colab Notebook for Data Preparation**.

Part 1: Taming the Chaos — Handling Diverse Datasets

After processing 19 different medical imaging datasets for a single project, I've learned that a hardcoded path for each dataset is inefficient and also a recipe for disaster. A robust data pipeline begins with a flexible and systematic approach to discovering and parsing data. The strategy in our `prepare_data.py` script is to:

- **Define a Standard Structure:** Assume a top-level directory (e.g., `dataset/`) that contains splits like `training/`. Inside, data is grouped by category (e.g., `Xray/`) and then by dataset name (e.g., `boneresorption/`).
- **Automate Discovery:** The script iterates through these directories to find the relevant image and question files, making the pipeline adaptable.
- **Handle Multi-Image Samples:** The script explicitly checks if the `"ImageName"` field is a single string or a list of strings, ensuring your model receives all the visual context it needs.

Part 2: The Gatekeeper — Writing Robust Validation Functions

Never trust your data. Corrupted images or missing files can silently poison your training process. Our `prepare_data.py` script acts as a gatekeeper, using `PIL.Image.verify()` to check for corrupted files and attempt to convert every image to RGB format to catch subtle loading issues. Any image that fails is logged and skipped, preventing it from derailing the training process.

Part 3: Creating a Unified Format - From Chaos to Clarity

Once you've validated your data, the final step is to convert it into a single, unified format. Modern VLMs are often fine-tuned using a "chat" or "instruction" format. The script transforms each sample into a consistent structure, like the input to a chatbot:

```
# A conceptual example of the final format
messages = [
    {
        "role": "user",
        "content": [
            {"type": "image", "image": "/path/to/image1.jpg"},
            {"type": "text", "text": "Examine these images and \
             identify all relevant labels..."}
        ]
    },
    {
        "role": "assistant",
        "content": "Atelectasis; Cardiomegaly"
    }
]
```

This approach is powerful because it's flexible, explicit, and standardized, allowing us to combine all 19 datasets into a single, massive training set.

Module 3: The Training Engine — Principles of Fine-Tuning

With our data forged into a clean, unified format, we arrive at the heart of the project: training. This is where we take our general-purpose Qwen-VL model and teach it to become a specialist in medical VQA.

> You can explore the code in the **Google Colab Notebook for Model Fine-Tuning**.

Part 1: The Memory-Saving Toolkit (PEFT & QLoRA)

How do we fit a giant model into a small memory budget? Instead of modifying all 7 billion parameters, we use **Parameter-Efficient Fine-Tuning (PEFT)**.

- **The Core Idea of PEFT:** We freeze the vast majority of the pre-trained model's weights and insert a very small number of new, trainable parameters (an "adapter").
- **LoRA (Low-Rank Adaptation):** This is the specific PEFT technique we use. LoRA adds pairs of small matrices to certain layers of the model, which are the only parts updated during training.
- **QLoRA (Quantized LoRA):** We save even more memory by quantizing the base model, reducing the precision of its weights from 32-bit to 4-bit numbers.

Together, QLoRA allows us to fine-tune a massive model on a single GPU.

Part 2: The Control Panel — Configuring the `SFTTrainer`

The Hugging Face `trl` library provides a powerful `SFTTrainer` that handles the **complex training loop** for us. We configure it using an `SFTConfig` object, which acts as our main control panel for critical settings like `learning_rate`, `num_train_epochs`, `batch_size`, and saving strategies. We also enable `gradient_checkpointing`, another essential memory-saving trick that trades a bit of computation speed for a significant reduction in VRAM usage.

Part 3: The Final Piece — The Multimodal Data Collator

The data collator bundles our individual samples into a single batch for the model. Its most critical job is **masking the prompt**. When we calculate the training loss, we only want to penalize the model for getting the *answer* wrong. The collator achieves this by creating a `labels` tensor where all tokens corresponding to the user's question are replaced with an ignore index (`-100`). This ensures the model learns to be a helpful assistant, not just a parrot.

Module 4: Beyond the Notebook — Inference & Deployment

Training a model is a significant achievement, but the ultimate goal is to create something *usable*. This final module focuses on writing a clean, self-contained inference script to apply your trained model to new data.

Part 1: The Anatomy of a Clean Inference Script

A good inference script is a self-sufficient, command-line utility. Here is a code snippet showing how you might use your trained model to make a prediction:

```
import torch
from PIL import Image
from transformers import AutoTokenizer, AutoProcessor, \
Qwen2_5_VLForConditionalGeneration, BitsAndBytesConfig
from peft import PeftModel

# --- 1. Load Model and Adapters ---
base_model_path = "Qwen/Qwen2.5-VL-7B-Instruct"
# Path to your fine-tuned weights
adapter_path = "./finetuned_qwenvl/final"
device = "cuda" if torch.cuda.is_available() else "cpu"

# Configure 4-bit quantization
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
```

```
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_use_double_quant=True,
)

# Load the base model in 4-bit
model = Qwen2_5_VLForConditionalGeneration.from_pretrained(
    base_model_path,
    torch_dtype=torch.float16,
    quantization_config=bnb_config,
    device_map="auto",
    trust_remote_code=True
)

# Apply the LoRA adapter
model = PeftModel.from_pretrained(model, adapter_path)
model.eval()

# Load tokenizer and processor
tokenizer = AutoTokenizer.from_pretrained(base_model_path, \
trust_remote_code=True)
processor = AutoProcessor.from_pretrained(base_model_path, \
trust_remote_code=True)

# Handle padding token if not set
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

# --- 2. Prepare Inputs ---
image_path = "path/to/your/medical_image.jpg"
question = "What abnormality is visible in this chest x-ray?"

try:
    image = Image.open(image_path).convert("RGB")
except Exception as e:
    print(f"Error loading image: {e}")
    exit(1)

prompt = f"Look at the image and answer the question accurately \
based on what you observe.\n\n{question}"

# Format for the model
messages = [
    {
        "role": "user",
        "content": [
            {"type": "image", "image": image},
```

```
            {"type": "text", "text": prompt}
        ]
    }
]

text = processor.apply_chat_template(messages, tokenize=False, \
add_generation_prompt=True)
inputs = processor(text=[text], images=[image], return_tensors="pt", \
padding=True)
inputs = {k: v.to(device) for k, v in inputs.items()}

# --- 3. Generate Prediction ---
with torch.no_grad():
    generated_ids = model.generate(
        **inputs,
        max_new_tokens=512,
        do_sample=False,  # Use greedy decoding for consistent results
        temperature=0.1,
        pad_token_id=tokenizer.pad_token_id
    )

    # Extract only the new tokens (remove input tokens)
    input_token_len = inputs['input_ids'].shape[1]
    response_ids = generated_ids[:, input_token_len:]
    response = tokenizer.batch_decode(response_ids, \
    skip_special_tokens=True)[0]

print(f"Model's Answer: {response.strip()}")
```

This snippet encapsulates the key steps: loading the model, preparing the inputs (image and text), generating a response, and cleaning the final output.

Part 2: Packaging for Reproducibility

To truly share your work, you need to package the entire project as a cohesive unit. The gold standard for this is **Docker**. The provided repository includes a `docker_deployment/` directory containing a `Dockerfile`. This file contains all instructions to build a complete, isolated environment with all libraries, code, and model weights. By building a Docker image, you create a self-contained "snapshot" of your project that can be run on any machine, eliminating the "*it works on my machine*" problem.

> Please refer to the README.md if you want to test the Docker example yourself.

Conclusion: Your Journey Begins Now

We have journeyed from a high-level concept to a fully functional and packaged Vision-Language Model. We started by drawing our blueprint, entered the data crucible, fired up the training engine, and finally, packaged our model for inference.

This tutorial was designed to provide more than a walkthrough; it aimed to offer a mental model and a reusable code philosophy. The real journey starts now. The skills you've learned are the foundational building blocks for innovation.

Take this framework, adapt it, break it, and build upon it. The advice I would give my past self, and the advice I now give to you, is this: master the fundamentals of the end-to-end pipeline, and you will be empowered to reproduce existing work while confidently building, debugging, and innovating on your own.

> Welcome to the MICCAI community. Now, create something amazing.

References

- **FLARE 2025 Challenge**: The official website for the Fast, Low-resource, Accurate, Robust, and Effectual (FLARE) challenge at MICCAI 2025 provides details on the tasks, data, and evaluation. More information can be found on the MICCAI 2025 Challenges Page and the FLARE Task 5 Codabench Page.
- **Qwen-VL Model**: Bai, J., et al. (2023). Qwen-VL: A Versatile Vision-Language Model for Understanding, Localization, Text Reading, and Beyond. arXiv preprint arXiv:2308.12966.
- **LLaVA Model**: Liu, H., et al. (2023). Visual Instruction Tuning. arXiv preprint arXiv:2304.08485.
- **InternVL**: Chen, Z., et al. (2023). InternVL: Scaling up Vision Foundation Models and Aligning for Generic Visual-Linguistic Tasks. arXiv preprint arXiv:2312.14238.
- **Florence-2**: Xiao, B., et al. (2024). Florence-2: Advancing a Unified Representation for a Variety of Vision Tasks. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).
- **Parameter-Efficient Fine-Tuning (PEFT)**: The Hugging Face PEFT library is central to the methods described. The repository and documentation provide a comprehensive overview of techniques like LoRA and QLoRA. Hugging Face PEFT GitHub.
- **TRL and the SFTTrainer**: The Hugging Face TRL (Transformer Reinforcement Learning) library provides the SFTTrainer used for supervised fine-tuning. Hugging Face TRL Documentation.