Disease_models

Siddharthakhandelwal/Disease_models

english

gemini-2.5-flash

9a93469

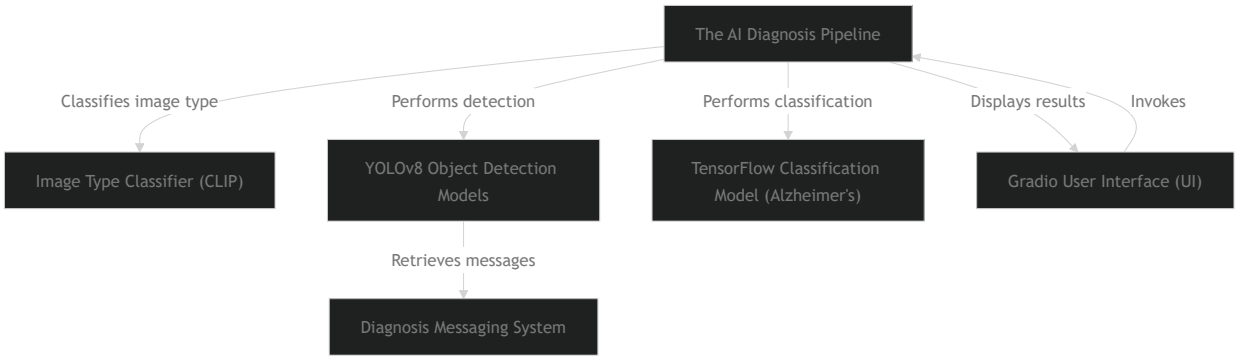
Aug 3, 2025

Chapters

Tutorial: Disease_models

This project is an **AI-powered web application** designed to help analyze *medical images* for disease detection. Users can **upload an image**, which is then automatically *classified by type* (e.g., brain scan, X-ray) using one AI model, and subsequently *diagnosed for specific conditions* like Alzheimer's or fractures by specialized AI models. Finally, the app displays the **processed image and a clear diagnosis** through an easy-to-use interface.

Visual Overview



Chapters

Generated by AI Codebase Knowledge Builder.

Disease_models

Siddharthakhandelwal/Disease_models

english

gemini-2.5-flash

9a93469

Aug 3, 2025

Chapters

Chapter 1: Gradio User Interface (UI)

Welcome to the `Disease_models` project tutorial! In this first chapter, we're going to explore the "Gradio User Interface" – or UI for short. Think of the UI as the friendly "front desk" of our entire medical image analysis application. It's what you'll see and interact with in your web browser.

Why Do We Need a UI?

Imagine you have a super smart AI doctor that can analyze medical images and tell you if there are any signs of diseases like Alzheimer's or bone fractures. That sounds amazing, but what if this AI doctor only understood complicated computer code? You'd need to be a computer expert just to ask it a question!

This is where the User Interface comes in. The problem it solves is simple: **how do we make powerful AI tools easy for *everyone* to use, even if they don't know how to code?**

Our goal is to let you, the user, simply upload a medical image and quickly see the AI's diagnosis, just like you would interact with any simple website. No technical skills required!

What is Gradio?

Gradio is a special "tool kit" (a Python library) that helps us build these user-friendly web interfaces for AI models very quickly. It's like having a magic wand that turns your complex AI code into a simple web application with buttons, image displays, and text boxes.

Here's what Gradio provides for our project:

- **Simple Image Upload:** An easy way for you to pick an image from your computer.
- **Result Display:** A clear spot to show you the processed image (maybe with highlights) and the AI's diagnosis message.
- **Web Browser Access:** It automatically creates a web link that you can open in any browser, making the app accessible.

How to Use Our Gradio UI

Using the `Disease_models` application is super straightforward, thanks to Gradio!

Step 1: Run the Application

First, you need to start the application. This is like turning on the "front desk" of our AI doctor.

Open a terminal or command prompt on your computer and type:

```
python app.py

```

After a few moments, Gradio will give you a local web address (usually something like `http://127.0.0.1:7860`). Copy this address and paste it into your web browser.

Step 2: Interact with the UI

Once you open the link in your browser, you'll see our application's "front desk":

- You'll see a prominent area where you can **upload your image**. Just click on it, and you'll be able to browse your computer for a medical image file (like a JPEG or PNG).

- Once you upload an image, the AI will get to work.
- After a short wait, you'll see two main outputs:
- A **processed image**, which might have markings or highlights from the AI's analysis.
- A **diagnosis result text**, explaining what the AI found.

Example Input and Output:

Imagine you upload an X-ray of a bone.

- **Input:** Your X-ray image file.
- **Output:**
- An image of your X-ray, possibly with a red box drawn around a detected fracture, or a "Not fractured" label if clear.
- A text message like: "No fracture detected in the X-ray. This result indicates healthy bone structure, but if symptoms persist, consult an orthopedic specialist."

How Does Gradio Set Up the UI? (Under the Hood)

Let's peek behind the scenes to see how our `app.py` file uses Gradio to create this friendly interface.

The "Front Desk" Setup

In `app.py`, the core of our Gradio UI is defined using `gr.Interface`. This is where we tell Gradio:

- What function (our "AI brain") it should call when you interact with the UI.
- What type of inputs it should expect (like an image).
- What type of outputs it should display (like another image and some text).

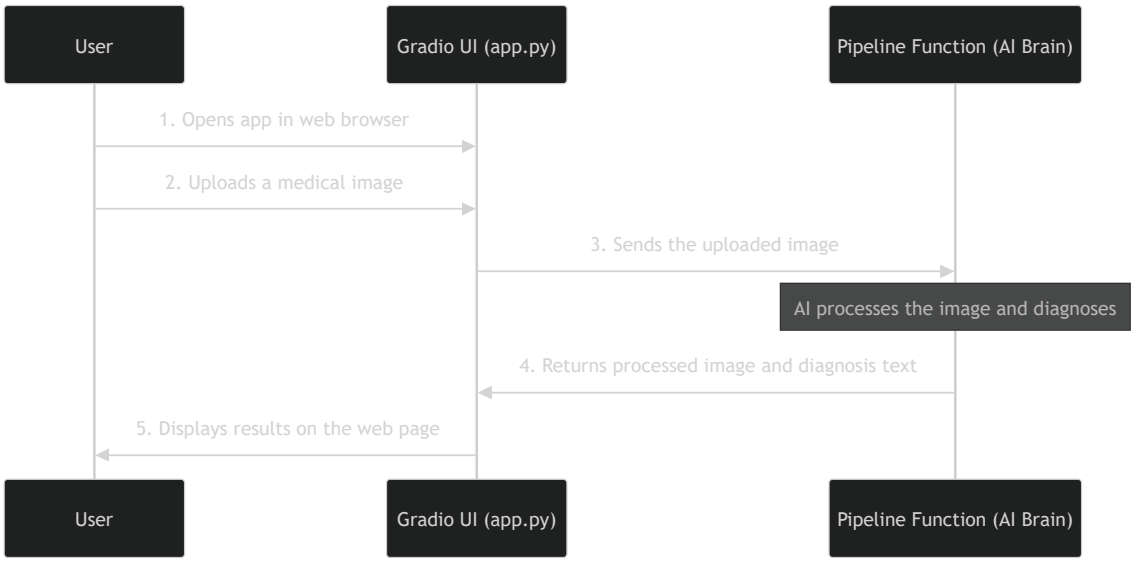
Here's a simplified look at the Gradio setup code in `app.py`:

```
import gradio as gr
from PIL import Image # Used for handling images

# This is our main AI logic function.
# It takes an image and returns a processed image and diagnosis text
# We'll learn more about this 'pipeline' in the next chapter!
def pipeline(input_image):
    # For now, just know it does the AI magic!
```

```
    # It receives your uploaded image and will eventually return results.
    return input_image, "Processing your image..." # Placeholder for now

# This creates our Gradio web interface!
demo = gr.Interface(
    fn=pipeline,                              # When someone uses the UI, call our '
    inputs=gr.Image(type="pil"),              # We expect an image upload.
    outputs=[gr.Image(type="pil"), gr.Text()], # We'll show an image and some tex
    title="Let's examine your reports!"       # The title for our web app.
)

# This command actually starts the web server and makes the UI available!
if __name__ == "__main__":
    demo.launch()

```

The UI's Interaction Flow

When you interact with the Gradio UI, here's a simple sequence of what happens:



1. **User Opens App:** When you run `python app.py` and open the link, the **Gradio UI** (created by `demo = gr.Interface( ... )`) loads in your web browser.
2. **User Uploads Image:** You select and upload your medical image through the U (`inputs=gr.Image( ... )`).

3. **Gradio Calls the Brain:** Gradio takes your uploaded image and automatically sends it to the `pipeline` function (`fn=pipeline`). This `pipeline` function is the true "brain" of our application, where all the AI magic happens.

4. **Brain Processes & Returns:** The `pipeline` function does its work (which we'll explore in the next chapter!) and then gives back two things: a new, processed image and a text message with the diagnosis.

5. **Gradio Displays Results:** Finally, Gradio receives these results and displays them neatly in the `outputs` sections of the UI (`outputs=[gr.Image( ... ), gr.Text()]`) for you to see!

Conclusion

In this chapter, we learned that the **Gradio User Interface** is the friendly face of our `Disease_models` application. It allows anyone to easily upload medical images and receive AI-powered diagnoses without needing any technical skills. Gradio acts as the bridge between you and the complex AI models running behind the scenes, making powerful technology accessible.

Next, we'll dive deeper into how that "AI Brain"—the `pipeline` function—actually works, step-by-step.

Next Chapter: The AI Diagnosis Pipeline

Generated by AI Codebase Knowledge Builder. **References**: [1], [2], [3], [4]

Disease_models

Siddharthakhandelwal/Disease_models

english

gemini-2.5-flash

9a93469

Aug 3, 2025

Chapters

Chapter 2: The AI Diagnosis Pipeline

Welcome back! In Chapter 1: Gradio User Interface (UI), we learned about the friendly "front desk" of our `Disease_models` application. You saw how easy it is to upload an image and receive a diagnosis using the Gradio UI. But what actually happens *after* you click "upload" and *before* you see the results?

That's where "The AI Diagnosis Pipeline" comes in!

What Problem Does the Pipeline Solve?

Imagine you walk into a large, modern hospital with a medical image (like an X-ray or MRI scan). You wouldn't just hand it to the first person you see, right? It needs to go to the right

department and the right specialist.

Similarly, our `Disease_models` project has several smart AI "doctors," each specialized in different types of medical images and diseases:

- One AI is great at looking for **Alzheimer's** in brain scans.
- Another AI is an expert at spotting **bone fractures** in X-rays.
- A different AI is skilled at detecting **pneumonia** in chest X-rays.
- And so on!

The problem is: **How does the application know *which* AI specialist to send your image to?** And once the specialist AI does its job, **how does it deliver the results back to you in a clear way?**

This is exactly what the **AI Diagnosis Pipeline** solves! It's like the highly organized system in a hospital that ensures your medical image goes to the correct department, gets analyzed by the right expert, and then the results are efficiently sent back to you. It connects all the different AI tools seamlessly.

What is "The AI Diagnosis Pipeline"?

Think of the **AI Diagnosis Pipeline** as an automated, multi-step medical lab. It's the complete journey your medical image takes from the moment you upload it until you receive a diagnosis. It's built as a single, powerful function in our code that orchestrates all the AI models.

It has three main stages:

1. **Image Triage (The AI Nurse):** First, an AI acts like a triage nurse. It quickly looks at your image and figures out what *kind* of medical image it is (e.g., "Is this a brain scan? Is it a bone X-ray?").
2. **Specialist Analysis (The AI Doctor):** Once the image type is known, it's sent to the *correct* specialist AI model for detailed analysis. This AI model then does its specific job, like finding a fracture or classifying Alzheimer's.
3. **Diagnosis Messaging (The Results Reporter):** Finally, the raw results from the specialist AI are translated into a clear, easy-to-understand diagnosis message for you, along with the processed image.

This entire process is handled by a single Python function named `pipeline` in our `app.py` file.

How to Use the Pipeline (from the User's V

You already know how to use it! In Chapter 1: Gradio User Interface (UI), we showed you how to run `python app.py` and upload an image in your web browser.

When you upload an image, Gradio (our UI tool) automatically calls this central `pipeline` function.

```python
# From app.py (simplified)
import gradio as gr
# ... other imports and AI model setup ...

# This is our central AI Diagnosis Pipeline function!
# It takes your uploaded image (input_image)
# and will return the processed image and a diagnosis message.
def pipeline(input_image):
    # This is where all the multi-step AI magic happens!
    # For now, just imagine it does super smart analysis.
    processed_image = input_image # Just a placeholder for now
    diagnosis_message = "Your image is being carefully analyzed by our AI special
    return processed_image, diagnosis_message

# This tells Gradio to use our 'pipeline' function
demo = gr.Interface(
    fn=pipeline,                          # When you interact, call 'pipeline'
    inputs=gr.Image(type="pil"),          # Expects an image as input
    outputs=[gr.Image(type="pil"), gr.Text()] # Shows an image and text as output
    # ... other UI settings ...
)

if __name__ == "__main__":
    demo.launch()

```

Explanation:

- The `pipeline` function is the core of our application's intelligence.
- When you upload an image in the Gradio UI, Gradio automatically sends that image to our `pipeline` function.
- The `pipeline` function then does all the complex work behind the scenes and returns two things: a potentially modified image (with highlights or boxes) and a text message describing the diagnosis.

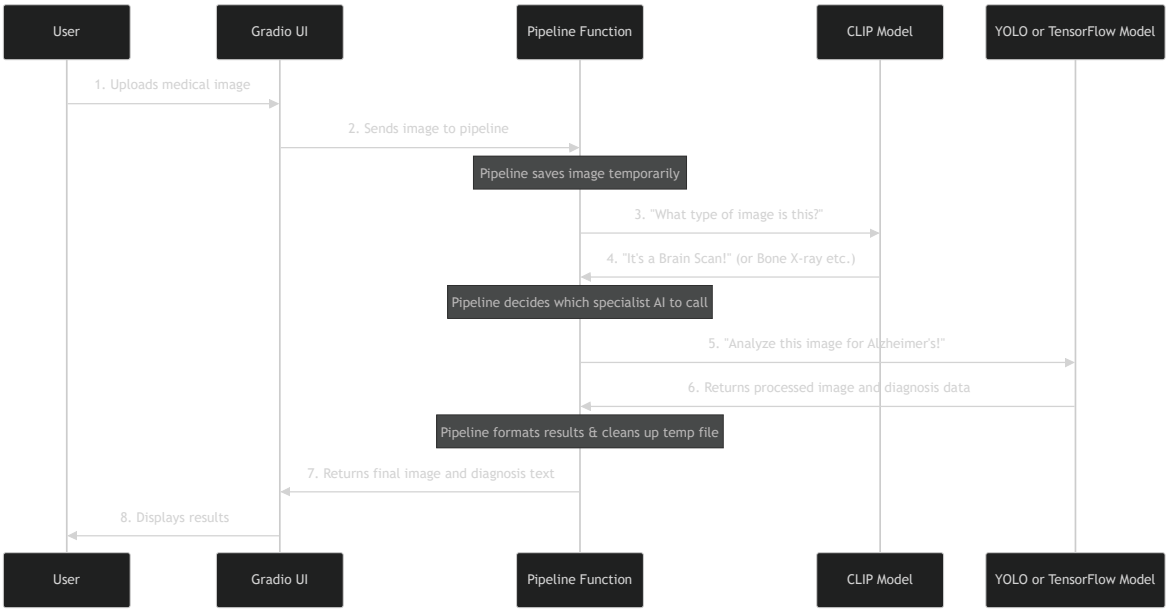
- Gradio then displays these two results back to you in your web browser.

Inside the Pipeline: How It Works (Under the Hood)

Let's take a peek at what really happens step-by-step when your image goes through the `pipeline` function.

The Pipeline's Journey (Step-by-Step Walkthrough)

Here's the sequence of events when you upload an image:



1. **User Uploads Image:** You select an image file through the Gradio web interface.
2. **Gradio Calls Pipeline:** The Gradio UI receives your image and passes it directly to our `pipeline` function.
3. **Temporary Storage:** The `pipeline` function first saves your uploaded image to a temporary file on the computer. This is so the AI models, which often work with file paths, can easily access it.
4. **Triage (CLIP Model):** The `pipeline` then sends this temporary image file to the Image Type Classifier (CLIP) model. This AI's job is to figure out what kind of medical image it is (e.g., "bone_fracture", "alzheimers", "Pneumonia").
5. **Specialist Selection:** Based on what the Image Type Classifier (CLIP) tells it, the function intelligently decides which specialist AI model should analyze the image fu
6. **Specialist Analysis (YOLO or TensorFlow):**

- If it's a "brain scan" image, the `pipeline` calls the TensorFlow Classification Model (Alzheimer's) to check for signs of Alzheimer's.
- If it's a "bone fracture" or "Pneumonia" image, the `pipeline` calls one of the YOLOv8 Object Detection Models to find specific objects or conditions.

7. **Result Formatting:** The chosen specialist AI model performs its analysis and returns raw results (like coordinates of a bounding box or probability scores). The `pipeline` then processes these raw results and uses the Diagnosis Messaging System to turn them into a clear, friendly message for the user.

8. **Cleanup:** The temporary image file is deleted to keep things tidy.

9. **Return Results to Gradio:** The `pipeline` function sends the final processed image (maybe with detected areas highlighted) and the clear diagnosis message back to the Gradio UI.

10. **Display to User:** Gradio displays these results on the web page for you to see!

Peeking at the Code (`app.py`)

Let's look at the actual `pipeline` function in `app.py` . We'll simplify parts that are explained in later chapters, focusing on the overall flow.

```python
# app.py (simplified pipeline function)

def pipeline(input_image):
    # Make sure an image was uploaded
    if input_image is None:
        return None, "❌ Please upload an image"

    # 1. Save the uploaded image to a temporary file
    # This is important because our AI models often work with file paths.
    temp_input = f"temp_{uuid.uuid4().hex}.jpg" # Creates a unique temporary filer
    input_image.save(temp_input) # Saves the image to that file

    # 2. Stage 1: Ask the AI Triage Nurse (CLIP) what kind of image it is
    # This part figures out if it's "alzheimers", "bone_fracture", etc.
    # We'll learn how get_image_embedding and cosine_similarity work in the CLIP c
    # For now, just know it gives us the 'best_label' (e.g., "alzheimers" or "bone
    # and 'best_score' (how confident it is).
    #
    # best_label = get_image_type_using_clip(temp_input) # Simplified
    best_label = "alzheimers" # Let's assume for this example
    best_score = 0.95 # A high confidence score

    # Optional check: If the AI nurse isn't confident, reject the image.
```

```
    if best_score < 0.25:
        os.remove(temp_input) # Clean up temp file
        return input_image, "❌ Upload a Valid Medical Image (low similarity scor

    # 3. Stage 2: Send to the right specialist AI model for detailed analysis
    try:
        if best_label == "alzheimers":
            # If the AI nurse says "brain scan", call the Alzheimer's specialist!
            # We'll dive into the Dementia function in the TensorFlow chapter.
            output_img, result_text = Dementia(temp_input) # Calls our TensorFlow
        elif best_label in YOLO_MODELS: # Check if it's a type handled by YOLO mo
            # If it's a bone, spine, pneumonia, etc., call the right YOLO special
            model_path = YOLO_MODELS[best_label] # Get the specific YOLO model fi
            # We'll explain yolo_predict in the YOLO chapter.
            output_img, result_text = yolo_predict(temp_input, model_path, best_l
        else:
            # If it's an unknown type (shouldn't happen with our known list)
            result_text = f"❌ Unknown image type: {best_label}"
            output_img = input_image
    except Exception as e:
        # Catch any errors during the AI analysis stage
        print(f"Error in model inference: {e}")
        output_img = input_image
        result_text = f"❌ Error during model inference: {str(e)}"

    # 4. Clean up the temporary file
    if os.path.exists(temp_input):
        os.remove(temp_input)

    return output_img, result_text

```

Explanation of the Code:

- `temp_input = f"temp_{uuid.uuid4().hex}.jpg"` : This line creates a unique name for a temporary file to save your uploaded image. This is like putting a unique tag on your medical image when it enters the lab.
- `input_image.save(temp_input)` : This command saves the image you uploade[illegible] temporary file.
- `best_label = get_image_type_using_clip(temp_input)` : (Simplified) This [illegible]he first major step: using the Image Type Classifier (CLIP) to determine the image category.

- `if best_score < 0.25:` : This is a safety check. If the `CLIP` model isn't very confident about what kind of image it is (low `best_score`), it tells you to upload a valid medical image. It's like the triage nurse saying, "I can't tell what this is, please provide a clearer image."
- `if best_label == "alzheimers":` : This is where the pipeline makes its intelligent decision! If the "AI nurse" (`CLIP`) identified the image as an "alzheimers" type (meaning a brain scan), it knows to call the `Dementia` function.
- `Dementia(temp_input)` : This function (which we'll explore in the TensorFlow Classification Model (Alzheimer's) chapter) is our specialist AI for Alzheimer's.
- `elif best_label in YOLO_MODELS:` : If the `best_label` isn't "alzheimers" but is one of the types handled by our YOLOv8 Object Detection Models (like "bone_fracture", "Pneumonia", etc.), the pipeline selects the appropriate YOLO model.
- `yolo_predict(temp_input, model_path, best_label)` : This function (explained further in the YOLOv8 Object Detection Models chapter) is our specialist AI for detecting things like fractures or pneumonia.
- `os.remove(temp_input)` : After all the analysis is done, this line cleans up by deleting the temporary image file.

Conclusion

The **AI Diagnosis Pipeline** is the powerful engine that connects all the different AI "doctors" in our `Disease_models` project. It ensures that every medical image you upload goes through a structured, intelligent process: first, it's identified, then it's analyzed by the right specialist AI, and finally, the results are presented clearly. It's the secret sauce that makes our application so versatile and easy to use, seamlessly linking the user interface to the complex AI models.

Next, we'll dive into the very first step of this pipeline: how our AI "triage nurse" identifies the type of medical image using a tool called CLIP.

Next Chapter: Image Type Classifier (CLIP)

Disease_models

Siddharthakhandelwal/Disease_models

english

gemini-2.5-flash

9a93469

Aug 3, 2025

Chapters

Chapter 3: Image Type Classifier (CLIP)

Welcome back! In Chapter 2: The AI Diagnosis Pipeline, we explored the journey your medical image takes inside our `Disease_models` application. We learned that the very first step in this pipeline is handled by an "AI triage nurse" that figures out what kind of medical image you've uploaded.

In this chapter, we're going to dive deep into that "AI triage nurse" itself: the **Image Type Classifier, powered by CLIP**.

What Problem Does the CLIP Classifier Sol

Imagine you're visiting a large hospital. When you arrive, you don't just walk into any room; you first go to a **triage nurse**. This nurse quickly assesses your situation and directs you to the right specialist or department (e.g., "You need to see the orthopedic doctor for your broken arm X-ray," or "You need the neurologist for your brain scan results").

Our `Disease_models` application faces a similar challenge. We have different specialized AI "doctors" (like our Alzheimer's model or our bone fracture model). Each of these AI doctors is trained for a *specific type* of image and a *specific disease*.

The problem is: **How does our application know which specialist AI to send your uploaded image to?** If you upload a brain MRI, we don't want to accidentally send it to the bone fracture AI! That would lead to completely wrong results.

This is where the **Image Type Classifier (CLIP)** comes in. It solves this crucial problem by acting as our AI "triage nurse." Before any specific disease detection happens, CLIP quickly looks at your image and determines what kind of scan it is—like a bone X-ray, a brain MRI, or a hair image. This step ensures that the correct, specialized disease-detection model is used for accurate analysis, preventing misinterpretations from using the wrong tool for the job.

What is CLIP?

CLIP stands for "Contrastive Language–Image Pre-training." Sounds complicated, right? Don't worry, we'll simplify it!

Think of CLIP as a super-smart AI that has learned to understand **both images and descriptions of images**. It's like a highly educated librarian who can not only recognize books by their covers but also understand what they are about just by reading a short summary.

In our project, we use CLIP's amazing ability to understand images. It works by converting any image into a special kind of "digital fingerprint" or "essence" called an **embedding**. Images that are similar will have similar digital fingerprints.

How Does CLIP Identify Image Types?

Our application has a small "photo album" of known medical images, each labeled with its type:

Image Type Label	Example Reference Image
`bone_fracture`	A sample bone X-ray
`alzheimers`	A sample brain MRI
`spine`	A sample spine X-ray

Image Type Label	Example Reference Image
`brain_tumor`	A sample brain tumor scan
`Pneumonia`	A sample chest X-ray
`Hair`	A sample hair/scalp image
`Breast`	A sample breast scan image

When you upload *your* medical image:

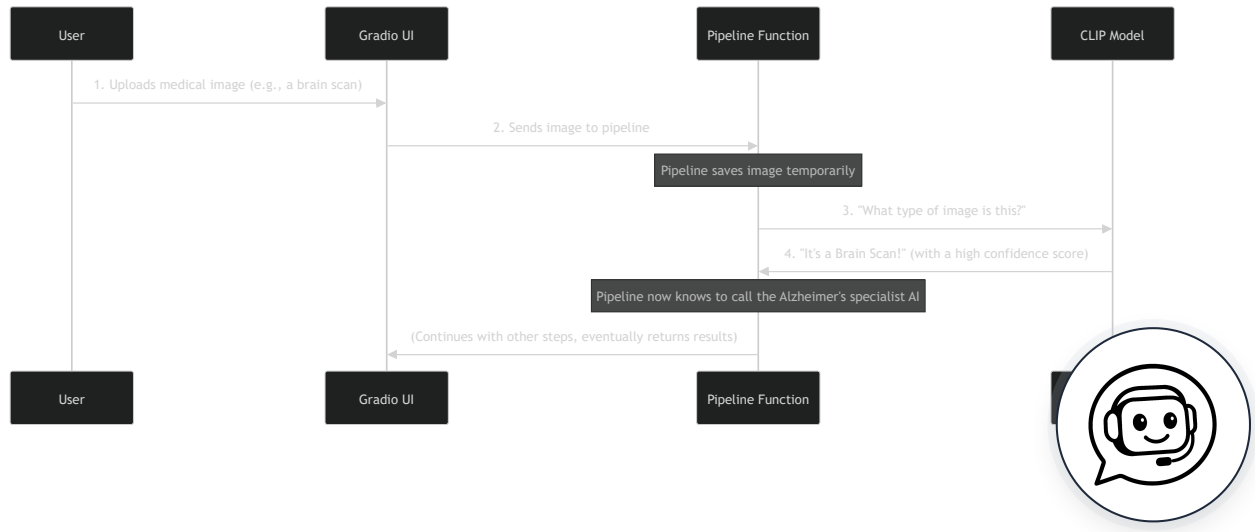
1. CLIP creates a **digital fingerprint** for your uploaded image.
2. It then compares your image's fingerprint to the fingerprints of all the **known example images** in its "photo album."
3. CLIP calculates a **similarity score** for each comparison. The higher the score, the more alike the images are.
4. The image type with the highest similarity score is chosen as the `best_label` (e.g., "alzheimers" or "bone_fracture").

This `best_label` is then passed along the pipeline so the right specialist AI model can be called.

How to Use CLIP in Our Project (Implicitly)

As a user, you don't directly "use" CLIP. Instead, it's the very first intelligent step our `pipeline` function (from Chapter 2: The AI Diagnosis Pipeline) takes after you upload your image.

When you upload an image via the Gradio UI:



This ensures that your brain scan is correctly identified as an "alzheimers" type image, allowing the pipeline to select the TensorFlow Classification Model (Alzheimer's) for accurate analysis.

Inside the CLIP Classifier (Under the Hood)

Let's look at the actual code (`app.py`) that makes this "AI triage nurse" work.

Setting Up CLIP

First, our application needs to load the CLIP model itself and prepare the "digital fingerprints" for our known example images. This happens when the application starts.

```python
# app.py (simplified)
import torch
from PIL import Image
from sklearn.metrics.pairwise import cosine_similarity
from transformers import CLIPProcessor, CLIPModel
import os # For file operations

model = None
processor = None

# This function loads the CLIP model and its helper (processor)
def load_models():
    global model, processor
    if model is None or processor is None:
        model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
        processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")
    return model, processor

# Our "photo album" of known images and their paths
KNOWN_IMAGES = {
    "bone_fracture": "images/bone.jpg",
    "alzheimers": "images/alzaimers.jpg",
    "spine": "images/spine.jpg",
    # ... other known image types
}

# Load the CLIP model when the script starts
model, processor = load_models()

# For each known image, create its digital fingerprint (embedding)
```

```
known_embeddings = {
    label: get_image_embedding(path) for label, path in KNOWN_IMAGES.items()
}

```

Explanation:

- `CLIPModel` and `CLIPProcessor` are tools from the Hugging Face library that help us use CLIP.
- `load_models()` makes sure CLIP is loaded only once to save time.
- `KNOWN_IMAGES` is a Python dictionary that stores the names (labels) for our image types and the file paths to their example images.
- `known_embeddings` then stores the "digital fingerprints" for each of these example images. These are prepared upfront.

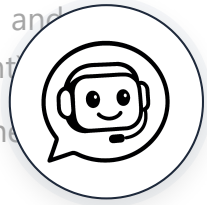
Creating a Digital Fingerprint for Your Image

When you upload an image, a special function is called to convert *your* image into its digital fingerprint.

```
# app.py (simplified)

# This function takes an image file path
# and turns it into a CLIP digital fingerprint (embedding)
def get_image_embedding(image_path):
    image = Image.open(image_path).convert("RGB") # Open the image
    inputs = processor(images=image, return_tensors="pt") # Prepare it for CLIP
    with torch.no_grad(): # Don't track changes, just get the result
        embeddings = model.get_image_features(**inputs) # Get the fingerprint!
        # Make sure the fingerprint is normalized (standardized length)
        return embeddings / embeddings.norm(p=2, dim=-1, keepdim=True)

```

Explanation:

- `get_image_embedding` is the core function that uses the loaded CLIP `model` and `processor` to turn any image into its unique `embeddings` (the digital fingerprint).
- `torch.no_grad()` is a technical detail that tells the computer we just want the fingerprint, not to prepare for training the AI, which saves memory and speeds things up.



The Pipeline's Triage Step

Finally, let's look at how the `pipeline` function uses these fingerprints to identify the image type:

```python
# app.py (simplified pipeline function)
import uuid # For creating unique filenames

def pipeline(input_image):
    if input_image is None:
        return None, "❌ Please upload an image"

    # 1. Save the uploaded image to a temporary file
    temp_input = f"temp_{uuid.uuid4().hex}.jpg" # Unique temporary filename
    input_image.save(temp_input) # Saves the image

    # 2. Stage 1: CLIP Classification (The AI Triage Nurse)
    input_embedding = get_image_embedding(temp_input) # Get fingerprint of YOUR i

    # Calculate how similar your image's fingerprint is to all known fingerprints
    similarities = {
        label: cosine_similarity(input_embedding.cpu().numpy(), known_embed.cpu()
        for label, known_embed in known_embeddings.items()
    }

    # Find the image type (label) with the highest similarity score
    best_label = max(similarities, key=similarities.get)
    best_score = similarities[best_label]

    # Optional check: If the AI nurse isn't confident enough, reject the image.
    if best_score < 0.25: # A score below 0.25 means it's not very similar to any
        os.remove(temp_input) # Clean up temp file
        return input_image, "❌ Upload a Valid Medical Image (low similarity scor

    # Now 'best_label' tells us what kind of image it is (e.g., "alzheimers", "bo
    # ... The pipeline then proceeds to call the correct specialist AI based on 'b

    # 3. Clean up the temporary file
    if os.path.exists(temp_input):
        os.remove(temp_input)

```

```
    return # ... processed image and diagnosis text ...

```

Explanation:

- `input_embedding = get_image_embedding(temp_input)` : This line gets the digital fingerprint for the image *you* just uploaded.
- `cosine_similarity( ... )` : This is a mathematical calculation that tells us how "alike" two digital fingerprints are. A score of 1 means they are identical, 0 means completely different.
- `best_label = max(similarities, key=similarities.get)` : After comparing your image's fingerprint to all the known ones, this line picks the label of the known image that had the highest similarity score.
- `if best_score < 0.25:` : This is a safety measure. If CLIP isn't very confident (the `best_score` is too low), it means your image probably isn't one of the medical image types it knows about, or it's too blurry. In such cases, it tells you to upload a valid medical image instead of trying to force a diagnosis.

Conclusion

The **Image Type Classifier (CLIP)** is like the smart "triage nurse" at the beginning of our AI diagnosis pipeline. By quickly and accurately identifying the type of medical image you upload (e.g., brain scan, bone X-ray, hair image), it ensures that your image is sent to the correct specialized AI model for analysis. This crucial first step prevents misdiagnosis and makes our application robust and reliable.

Now that we know how the pipeline chooses the right specialist, let's explore one of those specialists: the YOLOv8 models, which are great at finding specific objects or problems within an image!

Next Chapter: YOLOv8 Object Detection Models

Generated by AI Codebase Knowledge Builder. **References**: [1], [2], [3]

Disease_models

Siddharthakhandelwal/Disease_models

english

gemini-2.5-flash

9a93469

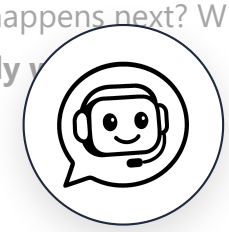
Aug 3, 2025

Chapters

Chapter 4: YOLOv8 Object Detection Models

Welcome back! In Chapter 3: Image Type Classifier (CLIP), we learned how our "AI triage nurse" (CLIP) intelligently identifies the type of medical image you've uploaded, like a bone X-ray or a brain scan. This crucial first step ensures that your image gets sent to the right specialist AI.

Now, imagine your image has been identified as a "bone fracture" X-ray. What happens next? We need a specialist AI that can actually *look* at the bone X-ray and **pinpoint exactly w fracture might be**, not just tell us it's a bone.



This is where our **YOLOv8 Object Detection Models** step in!

What Problem Do YOLOv8 Models Solve?

Think of our specialized AI "doctors" in two main categories:

1. **Classifiers:** These AIs look at an *entire image* and tell you what *kind* of image it is, or what *overall condition* is present (like our Alzheimer's model, which tells you the stage of dementia for the whole brain scan).

2. **Object Detectors:** These AIs are much more precise. They don't just tell you "This is an X-ray of a bone." They go further to say, "Okay, on this bone X-ray, I see a potential **fracture right here**!" and they'll even draw a box around it to show you.

The problem our **YOLOv8 Object Detection Models** solve is: **How do we not only detect a condition but also show its exact location on a medical image?**

Our goal is to provide visual evidence, like drawing a box around a fracture, a tumor, or an area of pneumonia. This makes the AI's diagnosis much more helpful and actionable for a medical professional.

What is YOLOv8?

YOLO stands for "You Only Look Once." This name might sound a bit quirky, but it's famous in the world of AI because it's a very fast and accurate way to detect "objects" in images.

In our project, we use YOLOv8 as a team of specialized "detectives" for different kinds of medical images:

- **Bone Detective:** A YOLO model specifically trained to find fractures on bone X-rays.
- **Spine Detective:** A YOLO model trained to identify specific parts of the spine.
- **Pneumonia Detective:** A YOLO model trained to spot signs of pneumonia on chest X-rays.
- And so on!

Each of these YOLO models is like a super-smart pair of eyes that can:

1. **Scan the Image:** Quickly look over the entire X-ray or image.

2. **Find Objects:** Identify potential problems (like a fracture) or specific features (like a spine segment).

3. **Draw Boxes:** Draw a "bounding box" (a rectangle) around anything it detects, pinpointing its exact location.

4. **Label Objects:** Tell you *what* it found within that box (e.g., "Fracture," "Pneumo

How Do YOLOv8 Models Work? (Simplified)

Imagine you're training a dog to find a specific toy. You show it many pictures of the toy, and each time, you draw a box around the toy. Eventually, the dog learns what the toy looks like and where it usually appears in different settings.

YOLO models are trained in a similar way:

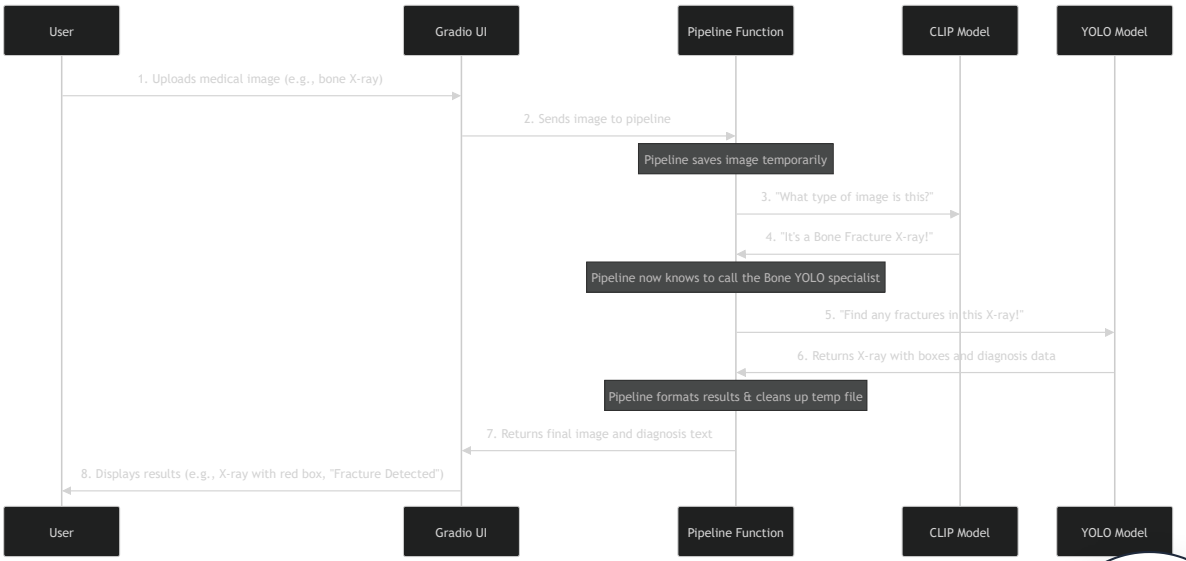
1. **Training:** We show a YOLO model thousands of medical images where experts have already drawn precise boxes around things like fractures, tumors, or signs of pneumonia. We also tell it what each box contains (e.g., "This box is a fracture").
2. **Learning Patterns:** The YOLO model learns to recognize the visual patterns associated with these "objects" and how to predict their locations.
3. **Detection:** When you give a *new* image to a trained YOLO model, it quickly scans the image. If it sees something it recognizes (like a fracture), it draws a box around it and tells you what it thinks is inside that box.

Because each YOLO model in our project is trained on a specific type of medical image (e.g., only bone X-rays for the `bone.pt` model), it becomes incredibly good at its specialized task.

How to Use YOLOv8 in Our Project (Implicitly)

Just like with CLIP, you don't directly interact with the YOLO models. They are an integral part of our AI Diagnosis Pipeline.

Here's how it fits into the overall flow:



As you can see, once the Image Type Classifier (CLIP) determines the image type, the ...ne function then smartly selects the appropriate YOLO model for the specific detection task.

Inside YOLOv8 Detection (Under the Hood)

Let's peek at the `yolo_predict` function in `app.py` and `try_model.py`, which is responsible for running our YOLO models.

Which YOLO Model to Use?

First, the `pipeline` function needs to know *which* YOLO model file to load. We keep a simple list for this:

```
# app.py (simplified)

YOLO_MODELS = {
    "bone_fracture": "bone.pt",      # Use 'bone.pt' for bone fracture detection
    "spine": "spine.pt",             # Use 'spine.pt' for spine analysis
    "brain_tumor": "brain_tumor.pt", # Use 'brain_tumor.pt' for brain tumors
    "Pneumonia": "Pneumonia.pt",     # Use 'Pneumonia.pt' for pneumonia detection
    "Hair":"Hair.pt",                # Use 'Hair.pt' for hair/scalp conditions
    "Breast":"breast.pt"             # Use 'breast.pt' for breast scans
}

# ... inside pipeline function ...
# after CLIP has determined best_label, e.g., "bone_fracture"
# model_path = YOLO_MODELS[best_label] # This gets "bone.pt"
# output_img, result_text = yolo_predict(temp_input, model_path, best_label)

```

Explanation: The `YOLO_MODELS` dictionary acts like a directory, mapping the image type identified by CLIP (like `"bone_fracture"`) to the specific YOLO model file (`"bone.pt"`) that handles that type.

Running the YOLO Prediction

The core of running a YOLO model happens in the `yolo_predict` function:

```
# app.py (simplified yolo_predict function)
from ultralytics import YOLO
from PIL import Image

def yolo_predict(image_path, model_path, best_label):
    try:
```

```
        # 1. Load the specific YOLO model (e.g., 'bone.pt')
        model = YOLO(model_path)

        # 2. Run the detection!
        # 'save=True' tells YOLO to save the image with detected boxes to a folder
        results = model(image_path, save=True, save_txt=False)

        # 3. Get the path to the processed image (with boxes)
        output_dir = results[0].save_dir
        output_img_path = os.path.join(output_dir, os.path.basename(image_path))

        # ... (rest of the function for handling results) ...

```

Explanation:

- `model = YOLO(model_path)` : This line loads the specific YOLO model (e.g., `bone.pt`) into memory, ready to work.
- `results = model(image_path, save=True, save_txt=False)` : This is the magic line! It tells the YOLO model to analyze `image_path` .
- `save=True` : This is important! It automatically saves a *new version* of your image into a `runs` folder, with all the detected objects highlighted with bounding boxes and labels.
- `save_txt=False` : We don't need text files of coordinates, just the image.
- `output_img_path` : This line figures out where that new, processed image (with the boxes) was saved.

Handling "No Detections" (Special Case for Bone Fractures)

Sometimes, the AI might not find any fractures on an X-ray. For the bone fracture model, we want to explicitly state "No fracture detected" instead of just showing a blank image.

```
# app.py (simplified yolo_predict function, continued)
import os
from PIL import Image, ImageDraw, ImageFont # For drawing on images

def yolo_predict(image_path, model_path, best_label):
    # ... (previous code for loading model and running prediction)

    # 4. Check if anything was detected
    no_detections = results[0].boxes is None or len(results[0].boxes) == 0
```

```
10
11      if model_path == "bone.pt" and no_detections:
12          # If it's a bone X-ray and no fracture was found:
13          with Image.open(output_img_path) as img:
14              img = img.convert("RGB")
15              draw = ImageDraw.Draw(img)
16              width, height = img.size
17
18              # Draw a big red box over the entire image
19              draw.rectangle([(0, 0), (width, height)], outline="red", width=5)
20
21              # Add "Not fractured" text
22              try: # Try to use a nice font, otherwise use default
23                  font = ImageFont.truetype("arial.ttf", 30)
24              except:
25                  font = ImageFont.load_default()
26              draw.text((10, 10), "Not fractured", fill="red", font=font)
27
28              output_image = img.copy() # Get the modified image
29              response = "No fracture detected in the X-ray. This result indicates t
30      # ... (rest of the function for general detections) ...
31
```

Explanation:

- `no_detections` : This checks if the YOLO model found *any* objects.
- `if model_path == "bone.pt" and no_detections:` : This specific check is only for the bone model. If no fracture is detected, we draw a clear red box around the whole image and add "Not fractured" text using `ImageDraw` and `ImageFont` tools.
- `response` : A specific message is created for this "no fracture" scenario.

Processing General Detections and Generating Messages

If the YOLO model *does* find something, we then extract the details and get a diagnosis message.

```
1  # app.py (simplified yolo_predict function, continued)
2  import json # For loading diagnosis messages
3
4  def yolo_predict(image_path, model_path, best_label):
5      # ... (previous code for loading model, running prediction, and b      eck)
6
```

```
    else:
        # There ARE detections for other YOLO models (or bone.pt with detections)
        with Image.open(output_img_path) as img:
            output_image = img.convert("RGB").copy() # Get the image with boxes

            # Get the class of the first detected object (e.g., 'cancer', 'pneumor
            predicted_class_index = int(results[0].boxes.cls[0].item())

            # Load the pre-defined diagnosis messages from 'diagnosis_messages.jso
            with open("diagnosis_messages.json", "r", encoding="utf-8") as f:
                response_texts = json.load(f)

                # Get the specific message for this image type and detected class
                response = response_texts[best_label][str(predicted_class_index)]

    return output_image, response # Return the processed image and the diagnosis t

```

Explanation:

- `output_image = img.convert("RGB").copy()` : This line loads the image that YOLO saved, which already has the bounding boxes drawn on it.
- `predicted_class_index = int(results[0].boxes.cls[0].item())` : If YOLO finds multiple objects, this takes the class of the *first* detected object. Each YOLO model is trained to detect specific classes (e.g., for `breast.pt` , `0` means 'cancer', `1` means 'normal'). You can see these class mappings in `model_deatils.py` .
- `response_texts = json.load(f)` : This loads a special file (`diagnosis_messages.json`) that contains all our pre-written diagnosis messages.
- `response = response_texts[best_label][str(predicted_class_index)]` : This is where the Diagnosis Messaging System comes into play. It uses the `best_label` (e.g., `"Breast"`) and the `predicted_class_index` (e.g., `"0"` for cancer) to fetch the exact diagnosis message for the user.
- `return output_image, response` : Finally, the `yolo_predict` function returns the image with the bounding boxes and the helpful diagnosis text back to the `pipeline` function, which then sends it to the Gradio UI.

Conclusion

The **YOLOv8 Object Detection Models** are the specialist "detectives" in our `Dise` project. They go beyond simple classification by precisely locating and highlighting po

problems or features directly on medical images like X-rays. This visual evidence, combined with a clear textual diagnosis, makes our application a powerful tool for initial assessment.

Now that we've seen how the AI models generate their results, let's learn how these raw results are turned into friendly and informative messages for you, the user!

Next Chapter: Diagnosis Messaging System

Generated by AI Codebase Knowledge Builder. **References**: [1], [2], [3], [4], [5], [6]

Disease_models

Siddharthakhandelwal/Disease_models

english

gemini-2.5-flash

9a93469

Aug 3, 2025

Chapters

Chapter 5: Diagnosis Messaging System

Welcome back! In Chapter 4: YOLOv8 Object Detection Models, we saw how our specialized AI "detectives" like YOLO can pinpoint problems on medical images and even draw boxes around them. Similarly, our TensorFlow Classification Model (Alzheimer's) gives us a classification like "Mild Demented."

But here's a question: if an AI tells you `class_index = 0` or `predicted_label = 'Mild Demented'`, what does that *really* mean for a person who isn't a medical or AI expert? How do we communicate these findings clearly, especially when getting a diagnosis in a foreign language!

What Problem Does the Diagnosis Messaging System Solve?

Imagine you visit a doctor, and after reviewing your tests, they just hand you a piece of paper with a cryptic code: `ICD-10 Code: S82.101A`. Would that be helpful? Probably not! You'd want them to explain: "You have a fracture in your lower leg, and here's what it means for you."

Our AI models are incredibly smart, but their raw outputs are often technical. They might say:

- "Object detected: Class `0` with 98% confidence."
- "Image classified as `2`."

The problem is: **How do we turn these technical AI results (like numbers or internal labels) into clear, friendly, and helpful messages that a non-technical person can understand and act upon?**

This is exactly what the **Diagnosis Messaging System** solves! It acts like a medical "translator" or a "results reporter." After an AI model makes a technical prediction, this system looks up that result in a pre-written database. It then provides a clear, user-friendly, and informative text message that explains the diagnosis. This ensures that the complex AI output is understandable and provides practical advice, making the results meaningful for everyone.

What is the Diagnosis Messaging System?

Think of the **Diagnosis Messaging System** as a comprehensive "message book" or "dictionary" for our AI application. This book contains all the possible diagnoses and helpful advice, written in plain language.

Here's how it works:

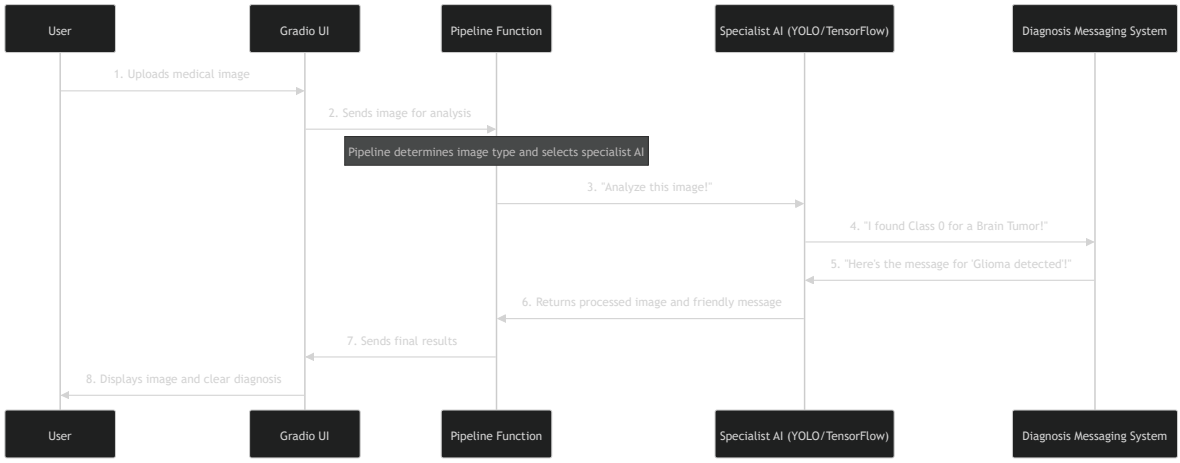
1. **AI Provides a "Code":** One of our specialist AI models (like YOLO for fractures or TensorFlow for Alzheimer's) performs its analysis and gives a technical result. For example, for a "brain tumor" scan, a YOLO model might detect an object and classify it as `class 0`.
2. **System Looks Up the Code:** The Diagnosis Messaging System takes this `class 0` and looks it up in its "message book" under the "brain tumor" section.
3. **Friendly Message is Returned:** It finds that `class 0` for "brain tumor" means "Glioma detected" and retrieves the full, detailed message explaining what a Glioma is and what steps to take.

This system ensures that no matter what technical output the AI provides, the user receives a well-explained, actionable message.

How to Use the Diagnosis Messaging System (Implicitly)

As a user, you won't directly interact with the Diagnosis Messaging System. It works behind the scenes as a crucial part of our AI Diagnosis Pipeline. It's the very last step that converts the raw AI prediction into the final text message you see in the Gradio UI.

Here's a simplified flow showing where it fits in:



As you can see, the Specialist AI models (like YOLO and TensorFlow) are the ones that consult with the Diagnosis Messaging System to get the final, user-friendly text message.

Inside the Diagnosis Messaging System (Under the Hood)

Let's peek at how this "message book" is stored and how our application retrieves messages from it.

The "Message Book": `diagnosis_messages.json`

Our system stores all the pre-written messages in a special file called `diagnosis_messages.json`. This file is formatted in a way that computers can easily read and organize, called JSON (JavaScript Object Notation).

It's structured like a nested dictionary, where you first pick the `image_type` (like "B…"), then the AI's `predicted_class_index` (like `0`, `1`, `2`, etc.), and finally, you get th…e.

Here's a small part of how the `diagnosis_messages.json` file looks:

Key (Image Type)	Nested Key (AI's Predicted Class Index)	Value (User-Friendly Message)
"brain_tumor"	"0"	🧠 Diagnosis: Glioma detected. Gliomas are tumors that arise from glial cells...
	"1"	🧠 Diagnosis: Meningioma detected. Meningiomas are typically non-cancerous tumors...
	"2"	🧠 Good News: No signs of a brain tumor were detected...
"Pneumonia"	"0"	🫁 Result: No signs of pneumonia detected. If you experience cough, fever, or shortness of breath...
	"1"	🫁 Diagnosis: Pneumonia detected. A lung infection that requires prompt medical treatment...

Notice how `0` means something different for "brain_tumor" than it does for "Pneumonia." This is because each AI model has its own set of classes for what it detects. The `Diagnosis Messaging System` smartly handles this by first checking the `image_type` (which we get from CLIP).

Loading the Messages

When our application starts or when a prediction needs a message, it first opens and reads this `diagnosis_messages.json` file.

```
# From app.py (inside yolo_predict function)
import json # This library helps us work with JSON files

# ... (rest of the yolo_predict function before this point) ...

        else:
            # ... (code to get the processed image with boxes) ...

            # Get the technical class index predicted by the AI (e.g.,
            predicted_class = int(results[0].boxes.cls[0].item())

            # Open our "message book" file
            with open("diagnosis_messages.json", "r", encoding="utf-8") as f:
```

```
14                # Load all the messages into a Python variable called 'response_tex
15                response_texts = json.load(f)
16
17                # Now, use the image type (best_label) and the predicted class
18                # to get the exact message from our 'message book'
19                response = response_texts[best_label][str(predicted_class)]
20
21        # ... (rest of the yolo_predict function) ...
22
```

Explanation:

- `import json` : This line brings in a special tool (library) that helps Python understand and work with JSON files.
- `predicted_class = int(results[0].boxes.cls[0].item())` : This is the crucial line where we get the technical output (a number like `0` , `1` , `2`) from the YOLO AI model.
- `with open("diagnosis_messages.json", "r", encoding="utf-8") as f:` : This command opens our `diagnosis_messages.json` file for reading.
- `response_texts = json.load(f)` : This line reads the entire content of the JSON file and turns it into a Python dictionary. Now, `response_texts` holds all our messages!
- `response = response_texts[best_label][str(predicted_class)]` : This is the magical lookup!
  - `best_label` comes from Chapter 3: Image Type Classifier (CLIP) (e.g., `"brain_tumor"`).
  - `str(predicted_class)` converts the AI's number (`0`) into a text string (`"0"`) because the keys in our JSON file are text.
  - So, if `best_label` is `"brain_tumor"` and `predicted_class` is `0` , this line would retrieve the message: "🧠 Diagnosis: Glioma detected...".

The `Dementia` function (which we'll cover in the next chapter) uses a very similar approach to get its diagnosis messages, by mapping its predicted class index to a predefined message.

Conclusion

The **Diagnosis Messaging System** is the friendly voice of our `Disease_models` application. It transforms the complex, technical outputs of our AI models into clear, actionable, and easy-to-understand messages for the user. By doing so, it bridges the gap between powerful technology and practical real-world understanding, making the application truly u and valuable.

Now that we know how results are presented, let's dive into one of the key specialist AI models itself: the TensorFlow model used for Alzheimer's diagnosis!

Next Chapter: TensorFlow Classification Model (Alzheimer's)

Generated by AI Codebase Knowledge Builder. **References**: [1], [2]

Disease_models

Siddharthakhandelwal/Disease_models

english

gemini-2.5-flash

9a93469

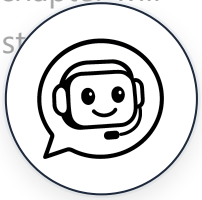
Aug 3, 2025

Chapters

Chapter 6: TensorFlow Classification Model (Alzheimer's)

Welcome back! In Chapter 5: Diagnosis Messaging System, we learned how our application takes the raw, technical results from our AI models and turns them into friendly, easy-to-understand messages for you. But where do those raw results come from in the first place?

That's where the **TensorFlow Classification Model (Alzheimer's)** comes in! This chapter will explain how this specific AI model looks at a brain MRI scan and tells us the likely st Alzheimer's disease.



What Problem Does This Model Solve?

Imagine you have a brain MRI scan. Our previous AI "detectives" (like the YOLOv8 Object Detection Models) are great at drawing boxes around specific things, like a fracture on an X-ray. But what if you need an **overall assessment** of the entire image?

For a brain scan, you might want to know the *general condition* of the brain related to a disease like Alzheimer's. You're not looking for a tiny spot, but rather a classification of the entire image into a category or stage.

The problem this model solves is: **How can a computer look at an entire brain MRI scan and tell us the overall stage of Alzheimer's disease, such as 'Mild Demented' or 'Non Demented', rather than just pointing out small features?**

This model acts like a dedicated "analyst" for brain MRI scans. It provides a broader classification, giving an overall assessment of the image's condition relative to its trained categories.

What is a TensorFlow Classification Model?

Let's break down the name:

- **TensorFlow**: This is a powerful "AI building kit" or "framework" created by Google. Think of it as a set of advanced tools and instructions that allow us to build complex AI models.
- **Keras**: Keras is like a user-friendly layer on top of TensorFlow. It makes building and training AI models much simpler, like using pre-made LEGO bricks instead of crafting each one from scratch. Our Alzheimer's model is built using Keras.
- **Classification Model**: This type of AI model specializes in putting entire items (like an image) into one of several predefined "boxes" or "categories." It learns to recognize patterns that differentiate one category from another.

How Does It Work for Alzheimer's?

Our Alzheimer's model is a type of AI called a **Convolutional Neural Network (CNN)**. Don't worry about the complex name! Just think of it as a very sophisticated image analyzer that works like this:

1. **Training**: We show the model thousands of brain MRI scans. Crucially, each scan is already labeled by medical experts with its correct Alzheimer's stage (e.g., 'Non Demented', 'Mild Demented', 'Moderate Demented', 'Very Mild Demented').

2. **Learning Patterns**: The CNN learns to identify subtle visual patterns, textures, and structures within these images that are characteristic of each stage of Alzheimer's. It's like learning a unique "fingerprint" of each stage.

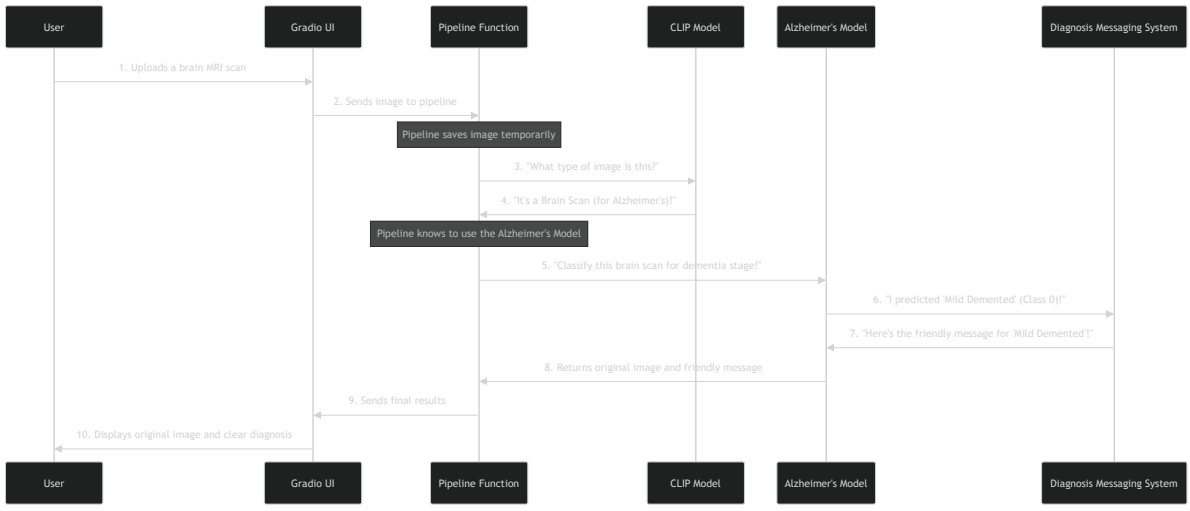
3. **Prediction**: When you give the trained model a *new*, unseen brain MRI scan, it analyzes its "fingerprint" and compares it to all the patterns it learned. It then predicts which stage the image most closely matches.

4. **Output**: The model doesn't just give you one answer; it provides a "confidence score" or "probability" for each possible category. For example, it might say: "90% chance of 'Mild Demented', 8% chance of 'Non Demented', and 2% for others." Our application then picks the category with the highest confidence.

How to Use This Model in Our Project (Implicitly)

As a user, you don't directly interact with the TensorFlow Alzheimer's model. It works automatically behind the scenes as part of our larger AI Diagnosis Pipeline.

Here's how it fits into the overall process when you upload a brain MRI:



As you can see, once the Image Type Classifier (CLIP) identifies your image as an "alzheimers" type (meaning a brain scan), the `pipeline` function intelligently calls our specialized Alzheimer's model.

Inside the Alzheimer's Model (Under the Hood)

Let's look at the `Dementia` function in `app.py` that runs our TensorFlow classification model.

Loading the Model

First, the application needs to load the trained Alzheimer's model into memory. This happens once when the application starts, so it's ready for quick predictions.

```
1 # app.py (simplified)
2 from tensorflow.keras.models import load_model
```

```
3
4  alzheimer_model = None # This will hold our loaded model
5
6  def load_alzheimer_model():
7      global alzheimer_model
8      if alzheimer_model is None:
9          try:
10             # We load the pre-trained model saved as 'alzheimers.h5'
11             alzheimer_model = load_model("alzheimers.h5")
12         except Exception as e:
13             print(f"Error loading Alzheimer's model: {e}")
14             return None
15     return alzheimer_model
16
17 # The model is loaded when the script runs
18 load_alzheimer_model()
19
```

Explanation:

- `from tensorflow.keras.models import load_model` : This line imports the special tool from Keras that lets us load a saved AI model.
- `alzheimer_model = load_model("alzheimers.h5")` : This is where the magic happens! It loads our pre-trained Alzheimer's model from a file named `alzheimers.h5` . Think of `.h5` as the file extension for a "brain" that has already learned.
- The `load_alzheimer_model` function ensures the model is loaded only once, which saves a lot of time and computer resources.

Making a Prediction

Now, let's see how the `Dementia` function uses this loaded model to classify your uploaded brain scan.

```
1  # app.py (simplified Dementia function)
2  from PIL import Image
3  import numpy as np
4  from tensorflow.keras.preprocessing import image as keras_image # Renamed to avoi
5
6  def Dementia(img_path):
7      model = load_alzheimer_model() # Get our loaded Alzheimer's mod
8      if model is None:
9          return Image.open(img_path), "❌ Error: Could not load Alzheimer's model"
```

```
10
11      # These are the categories our model was trained to predict
12      class_labels = ['Mild Demented', 'Moderate Demented', 'Non Demented', 'Very M
13
14      # 1. Load and prepare the image for the model
15      # Our model expects images to be 250x250 pixels and in RGB color.
16      img = keras_image.load_img(img_path, target_size=(250, 250))
17
18      # Convert the image into a format the AI can understand (numbers in an array)
19      img_array = keras_image.img_to_array(img)
20
21      # Add an extra dimension because the model expects a 'batch' of images, even :
22      img_array = np.expand_dims(img_array, axis=0)
23
24      # Normalize the pixel values from 0-255 to 0-1, as the model was trained this
25      img_array = img_array / 255.0
26
27      # 2. Make the prediction!
28      prediction = model.predict(img_array)
29
30      # 3. Interpret the prediction
31      # Find the index of the class with the highest probability (e.g., 0, 1, 2, or
32      predicted_index = np.argmax(prediction[0])
33
34      # Get the human-readable label using our list
35      predicted_label = class_labels[predicted_index]
36
37      # Get the confidence score for the predicted label
38      confidence = prediction[0][predicted_index]
39
40      # 4. Create the result message
41      result_text = f"🧠 Alzheimer Stage: {predicted_label} (Confidence: {confidenc
42      result_text += "🔍 All class probabilities:\n"
43      for label, prob in zip(class_labels, prediction[0]):
44          result_text += f"- {label}: {prob:.4f}\n"
45
46      return img, result_text # Return the original image and the diagnosis text
47
```

Explanation of the Code:

- `class_labels` : This list is super important! It tells us what each number the AI predicts actually means. For example, if the AI outputs `0`, it means 'Mild Demented'.
- `keras_image.load_img(img_path, target_size=(250, 250))` : AI models are very picky about the size of images they see. This line loads your image and makes sure it's resized to exactly 250x250 pixels, just like the images the AI was trained on.
- `img_array = np.expand_dims(img_array, axis=0) / 255.0` : This prepares the image data for the AI.
  - `np.expand_dims` : Most AI models are designed to process multiple images at once (a "batch"). Even if you only have one image, we add an extra dimension to make it look like a batch of one.
  - `/ 255.0` : This is called "normalization." Pixel values in images are usually between 0 and 255. Dividing by 255 changes them to be between 0 and 1. This helps the AI learn more effectively.
- `prediction = model.predict(img_array)` : This is the core line where the loaded AI model actually analyzes the prepared image and makes its prediction.
- `np.argmax(prediction[0])` : The `prediction` result is a set of numbers (probabilities) for each class. `np.argmax` finds the position (index) of the *highest* number, which means the class the AI is most confident about.
- `predicted_label = class_labels[predicted_index]` : Using the `predicted_index` (like `0` or `1`), we look up the actual human-readable label from our `class_labels` list.
- The `result_text` then formats all this information into a nice message, including the confidence score and probabilities for all classes, so you can see the AI's full "thinking" process.

This model was trained in a Jupyter Notebook file named `train_model_tensorflow.ipynb`. In that file, medical images of brains were carefully prepared, and the model learned to recognize patterns associated with different stages of Alzheimer's, eventually saving its learned "brain" into the `alzheimers.h5` file.

Conclusion

The **TensorFlow Classification Model (Alzheimer's)** is our specialized "analyst" for brain MRI scans. Unlike object detection models, it provides an overall assessment, classifying the entire image into predefined stages of Alzheimer's disease. This powerful AI, built with TensorFlow and Keras, translates complex visual patterns into clear diagnostic categories, providing a crucial piece of our `Disease_models` project's ability to offer comprehensive medical image analysis.

You've now explored all the key AI components that make the `Disease_models` pr
from the user interface to the intelligent diagnosis pipeline and its specialized AI "