TRƯỜNG ĐẠI HỌC MỎ - ĐỊA CHẤT

KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO SINH HOẠT HỌC THUẬT

ĐỀ TÀI: "TRIỂN KHAI MÔ HÌNH HỌC MÁY VỚI FAST API"

Người báo cáo: **ThS. Nguyễn Thùy Dương**

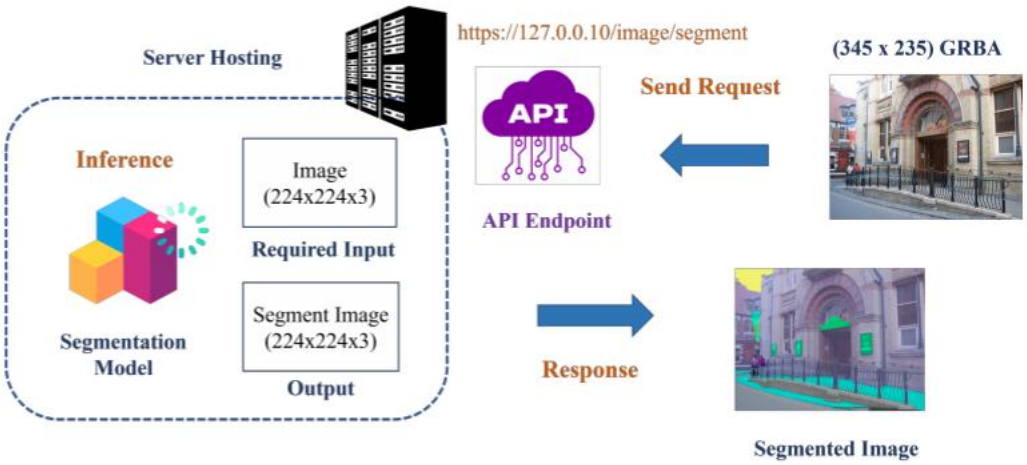
Đơn vị: **Bộ môn Khoa học máy tính**

Hà nội, 05-2026

TRIỂN KHAI MÔ HÌNH MACHINE LEARNING VỚI FASTAPI

I. Giới thiệu

Khi đối mặt với một vấn đề cần sử dụng AI hay cụ thể là Machine Learning để giải quyết, chúng ta biết quy trình để giai quyết vấn đề đó chính là xây dựng một mô hinh ML phù hợp với yêu cầu của tác vụ đó, bao gồm: thu thập dữ liệu, xử lý dữ liệu, huấn luyện, đánh giá mô hình. Saukhi có được một mô hình đủ tốt, bước tiếp theo cần quan tâm đó là làm sao đưa mô hình nàytừ môi trường research sang môi trường production (như app mobile, website). Có một thuậtngữ được gọi là serving model, nghĩa là model sẽ được deploy và được xem như một web service,nhận đầu vào input và trả về output, hay nhận yêu cầu request rồi trả về kết quả response.



Hình 1: Quy trình deploy và serving mô hình segmentation qua API: server nhận ảnh đầu vào, xử lý bằng mô hình qua inference và trả về ảnh segment thông qua API endpoint.

Để thực hiện được điều đó, một trong những khái niệm không thể không nhắc tới đó là API. API (viết tắt là Application Programming Interface - Giao diện Lập trình Ứng dụng) là một tập hợp các quy tắc và công cụ cho phép các ứng dụng, hệ thống hoặc dịch vụ khác nhau giao tiếp và trao đổi dữ liệu với nhau. Deploy mô hình là quá trình mà chúng ta sẽ hosting mô hình đó tại một server, triển khai các API endpoint để nhận các request (tức các input đầu vào), sau đó thay vì xử lý business logic như bình thường, thứ chũng ta cần là biến đổi dữ liệu

đầu vào sao cho phù hợp với mô hình, đợi mô hình thực hiện tính toàn (inference) rồi trả về output mong muốn.

Trong bài viết này, chúng ta sẽ tìm hiểu về FastAPI, một framework Python hiện đại, mạnh mẽ, được thiết kế để xây dựng các API nhanh chóng, hiệu quả, và tiện lợi.

FastAPI không chỉ nổi bật với hiệu năng cao nhờ xây dựng trên nền tảng ASGI (Asynchronous Server Gateway Interface) mà còn cung cấp các tính năng như tự động hóa tài liệu API hay validation dữ liệu với Pydantic. Trước khi đi vào ví dụ sử dụng FastAPI để triển khai mô hình Machine Learning, chúng ta sẽ thông qua một ví dụ thực tế: Student Management API - để minh họa cách xây dựng một ứng dụng FastAPI và để dễ dàng hiểu các khái niệm cơ bản nhất về framework này.

II. Các khái niệm cơ bản về FastAPI và ứng dụng quản lý sinh viên

Chúng ta sẽ từng bước tìm hiểu về các khái niệm của FastAPI và thực hiện triển khai code mẫu để hiểu hơn về nó

2.1. Routing, Include Route

Trong FastAPI, routing là cơ chế cốt lõi để liên kết một đường dẫn URL cụ thể với một hàm Python xử lý yêu cầu. Về mặt lý thuyết, routing trong các web framework như FastAPI dựa trên nguyên tắc của URL dispatching, nơi server phân tích URL để xác định hành động cần thực hiện.

FastAPI sử dụng thư viện Starlette làm nền tảng, hỗ trợ routing không đồng bộ (asynchronous) nhờ ASGI (Asynchronous Server Gateway Interface), giúp ứng dụng xử lý nhiều yêu cầu đồng thời mà không bị chặn. Điều này đặc biệt hữu ích trong các ứng dụng thời gian thực hoặc có lượng truy cập cao.

Modular design là cách tổ chức code thành các module độc lập, mỗi module chịu trách nhiệm một chức năng riêng, giúp dễ bảo trì và mở rộng (thường mỗi module sẽ chứa 1 file __init__.py).

main.py

```python
from fastapi import FastAPI
from .routers import students

app = FastAPI()
app.include_router(students.router)

@app.get("/")
def read_root():
    return {"message": "Chào mừng đến với hệ thống quản lý sinh viên!"}
```

routers/students.py

```python
from fastapi import APIRouter
from .models import Student

router = APIRouter(
    prefix="/students",
    tags=["Students"]
)

# Sample data
students_db = {
    1: {"name": "Nguyễn Văn A", "age": 20, "is_active": True},
    2: {"name": "Trần Thị B", "age": 22, "is_active": True}
}

@router.post("/", response_model=Student)
def create_student(student: Student):
    student_id = len(students_db) + 1
    students_db[student_id] = student
    return student

@router.get("/")
def get_all_students():
    return list(students_db.values())
```

2.2. Giao thức HTTP

Giao thức HTTP (Hypertext Transfer Protocol) là nền tảng của mọi giao tiếp trên web, được định nghĩa trong các RFC như RFC 7230-7235. Về lý thuyết, HTTP hoạt động theo mô hình client-server stateless, nghĩa là mỗi yêu cầu độc lập và không lưu trữ trạng thái giữa các request.

FastAPI hỗ trợ đầy đủ các phương thức HTTP chính, mỗi phương thức đại diện cho một loại hành động, tuân theo nguyên tắc RESTful (Representational State Transfer). REST nhấn mạnh vào việc sử dụng HTTP methods để thao tác tài nguyên, giúp API dễ mở rộng và dễ hiểu.

Stateless nghĩa là server không lưu thông tin về trạng thái của client giữa các request, mỗi request phải chứa đủ thông tin để xử lý.

FastAPI sử dụng ASGI để hỗ trợ HTTP/2 và HTTP/3, mang lại hiệu suất cao hơn so với HTTP/1.1 nhờ multiplexing (xử lý nhiều stream đồng thời) và header compression. Việc chọn đúng method HTTP giúp tránh các vấn đề bảo mật như idempotency (ví dụ: GET và DELETE nên an toàn khi lặp lại mà không thay đổi trạng thái). Các status code (như 200 OK, 404 Not Found) và headers (như

Content-Type) cũng được FastAPI xử lý tự động, nhưng chúng ta cần chú ý tùy chỉnh để tránh tiết lộ thông tin không cần thiết.

Idempotent nghĩa là gọi cùng một request nhiều lần sẽ cho ra cùng một kết quả như lần đầu tiên, không tạo thêm thay đổi mới trên server.

• GET: Lấy dữ liệu từ server, nên là idempotent và safe.

• POST: Gửi dữ liệu mới để tạo tài nguyên, không idempotent.

• PUT: Cập nhật toàn bộ một tài nguyên, idempotent.

• PATCH: Cập nhật một phần của tài nguyên, không nhất thiết idempotent.

• DELETE: Xóa một tài nguyên, idempotent.

Theo https://fastapi.tiangolo.com, FastAPI khuyến khích sử dụng async def cho các route để tận dụng đầy đủ lợi thế của HTTP không đồng bộ, đặc biệt trong I/O-bound tasks như gọi database.

I/O-bound task (nhiệm vụ bị ràng buộc bởi I/O) là loại công việc mà thời gian thực hiện chủ yếu bị chi phối bởi việc nhập/xuất dữ liệu (Input/Output), chứ không phải do CPU tính toán.

2.3. Path parameter

Path parameter là một phần của URL, được sử dụng để xác định một tài nguyên cụ thể, dựa trên nguyên tắc URI (Uniform Resource Identifier) trong HTTP. FastAPI sử dụng cú pháp: {tên_tham_số} để định nghĩa path parameter, và tận dụng type hinting của Python để tự động validate dữ liệu. Về lý thuyết, điều này áp dụng khái niệm type safety từ lập trình chức năng, giúp phát hiện lỗi sớm tại runtime thay vì deep trong code.

Type safety là cơ chế đảm bảo dữ liệu đúng kiểu (int, str, v.v.) trước khi xử lý, giúp giảm lỗi runtime và tăng độ tin cậy của code.

Ví dụ, để lấy thông tin của một sinh viên có ID cụ thể: FastAPI tự động xác thực kiểudữ liệu (ở đây là int) và trả về lỗi nếu dữ liệu không hợp lệ, như 422 Unprocessable Entity.

Ở đây, chúng ta thường nên thêm constraints như enum hoặc regex (biểu thức chính quy) (ví dụ: student_id: int = Path(..., ge=1)) để hạn chế input xấu, giảm tải cho database và tránh injection attacks. Tài liệu FastAPI gợi ý sử dụng Path từ fastapi để tùy chỉnh description, deprecated, v.v., giúp docs API chi tiết hơn.

Injection Attack là khi kẻ xấu cố tình chèn mã (như SQL) vào input để thao túng hệ thống, ví dụ: nhập lệnh xóa dữ liệu thay vì tên.

routers/students.py

```python
from fastapi import Path
from .models import Student

@router.get("/{student_id}", response_model=Student)
def get_student(student_id: int = Path(..., ge=1)):
    student = students_db.get(student_id)
    if not student:
        raise HTTPException(status_code=404, detail="Không tìm thấy sinh viên")
    return student
```

2.4. Pydantic và data model

Pydantic là thư viện được FastAPI tích hợp sẵn để xác thực và xử lý dữ liệu, dựa trên Python type hints và dataclass. Về lý thuyết, Pydantic áp dụng data validation và serialization /deserialization để đảm bảo dữ liệu sạch, tuân thủ schema. Điều này giống như JSON Schema nhưng tích hợp sâu với Python, giúp tránh lỗi như type mismatch hoặc missing fields.

Serialization/Deserialization là quá trình chuyển đổi dữ liệu thành định dạng có thể lưu trữ/gửi (serialization) và ngược lại (deserialization).

Bằng cách định nghĩa các model kế thừa từ pydantic.BaseModel, chúng ta có thể đảm bảo dữ liệu đầu vào và đầu ra tuân thủ một cấu trúc nhất định. Nested models (như StudentDetails) hỗ trợ dữ liệu phức tạp, và validators tùy chỉnh (ví dụ: @validator) cho phép logic business cụ thể. Pydantic giúp loại bỏ sớm các bug bằng cách validate sớm, đặc biệt trong microservices nơi dữ liệu di chuyển giữa các service. Ngoài ra, nó hỗ trợ ORM integration với SQLAlchemy, giúp map model trực tiếp với database schema.

models.py

```python
from pydantic import BaseModel, validator

class Student(BaseModel):
    name: str
    age: int
    is_active: bool = True

    @validator('age')
    def age_must_be_positive(cls, v):
        if v < 0:
            raise ValueError('Age must be positive')
        return v

class Course(BaseModel):
    course_name: str
    credits: int

class StudentDetails(BaseModel):
    student_info: Student
    courses: list[Course]
```

2.5. Tự động hóa tài liệu và Swagger UI

Một trong những điểm mạnh nhất của FastAPI là khả năng tự động tạo tài liệu API tương tác dựa trên OpenAPI standard (https://swagger.io/specification/). Về lý thuyết, OpenAPI định nghĩa schema cho API, bao gồm paths, parameters, responses, giúp tool như Swagger UI render giao diện. Chỉ cần chạy ứng dụng và truy cập /docs, bạn sẽ thấy giao diện Swagger UI, cho phép bạn xem và thử nghiệm các endpoint một cách trực quan. Ngoài ra, /redoc cung cấp giao diện khác với docs.

OpenAPI là chuẩn để mô tả cấu trúc API, giúp tự động tạo tài liệu và giao diện thử nghiệm, giảm công sức viết tài liệu thủ công

Tính năng này tiết kiệm hàng giờ so với viết docs thủ công như ở Django, và dễ tùy chỉnh bằng `response_model`, `summary`, `description`, và `operation_id` trong `decorator`.

Ví dụ, bạncó thể thêm operation_id="get_student_by_id" để đặt tên endpoint trong Swagger UI, hoặc summary="Lấy thông tin sinh viên" để mô tả ngắn gọn. Để tùy chỉnh sâu hơn, FastAPI cho phép override schema OpenAPI bằng cách định nghĩa hàm custom_openapi trong ứng dụng, hữu ích khi cần thêm metadata hoặc ẩn một số endpoint khỏi tài liệu.

custom_openapi.py

```python
from fastapi import FastAPI
from fastapi.openapi.utils import get_openapi

app = FastAPI()

def custom_openapi():
    if app.openapi_schema:
        return app.openapi_schema
    openapi_schema = get_openapi(
        title="Hệ thống quản lý sinh viên",
        version="1.0.0",
        description="API quản lý sinh viên với FastAPI",
        routes=app.routes,
    )
    openapi_schema["info"]["x-logo"] = {
        "url": "https://fastapi.tiangolo.com/img/logo-margin/logo-teal.png"
    }
    app.openapi_schema = openapi_schema
    return app.openapi_schema

app.openapi = custom_openapi
```

2.6. Xây dựng CRUD đầy đủ cho ứng dụng

CRUD là viết tắt của Create, Read, Update, Delete, đại diện cho các hoạt động cơ bản trên dữ liệu bền vững. Về lý thuyết, CRUD map trực tiếp với HTTP methods trong REST, đảm bảo API idempotent và stateless. FastAPI làm cho việc implement CRUD dễ dàng nhờ dependency injection và model validation.

REST (Representational State Transfer) là phong cách kiến trúc API sử dụng HTTP methods để thao tác tài nguyên, đảm bảo tính mở rộng và đơn giản.

Ví dụ, sử dụng POST để tạo dữ liệu mẫu và các route khác xử lý dữ liệu nhất quán. Chúng ta sử dụng response_model để đảm bảo output đúng schema và thêm error handling để graceful degradation.

routers/students.py

```python
from fastapi import HTTPException, APIRouter
from .models import Student

router = APIRouter(prefix="/students", tags=["Students"])

students_db = {
    1: Student(name="Nguyễn Văn A", age=20, is_active=True),
    2: Student(name="Trần Thị B", age=22, is_active=True)
}

# C (Create)
@router.post("/", response_model=Student)
```

```python
def create_student(student: Student):
    student_id = len(students_db) + 1
    students_db[student_id] = student
    return student

# R (Read)
@router.get("/", response_model=list[Student])
def read_students():
    return list(students_db.values())

@router.get("/{student_id}", response_model=Student)
def read_student(student_id: int):
    student = students_db.get(student_id)
    if not student:
        raise HTTPException(status_code=404, detail="Không tìm thấy sinh viên")
    return student

# U (Update) - PUT for full update
@router.put("/{student_id}", response_model=Student)
def update_student(student_id: int, student: Student):
    if student_id not in students_db:
        raise HTTPException(status_code=404, detail="Không tìm thấy sinh viên")
    students_db[student_id] = student
    return student

# U (Update) - PATCH for partial update
@router.patch("/{student_id}", response_model=Student)
def partial_update_student(student_id: int, student_update: dict):
    if student_id not in students_db:
        raise HTTPException(status_code=404, detail="Không tìm thấy sinh viên")
    current_student = students_db[student_id].dict()
    current_student.update(student_update)
    students_db[student_id] = Student(**current_student)
    return students_db[student_id]

# D (Delete)
@router.delete("/{student_id}")
def delete_student(student_id: int):
    if student_id not in students_db:
        raise HTTPException(status_code=404, detail="Không tìm thấy sinh viên")
    del students_db[student_id]
    return {"message": "Sinh viên đã được xóa"}
```

2.7. Security, Middleware và mở rộng

Security: FastAPI hỗ trợ bảo mật API bằng cách sử dụng các Dependency, dựa trên dependency injection pattern để reusable code. Về lý thuyết, security schemes như OAuth2, API keys tuân theo OWASP guidelines để chống lại threats như unauthorized access. Tích hợp JWT (JSON Web Tokens) với libraries như python-jose giúp secure API mà không lưu session, lý tưởng cho scalable apps.

Dependency Injection là kỹ thuật cung cấp các phụ thuộc (như hàm xác thực) cho một hàm khác, giúp code linh hoạt và dễ kiểm thử.

security_example.py

```python
from fastapi import Depends, HTTPException
from fastapi.security import OAuth2PasswordBearer
from typing import Annotated

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

def is_admin(token: Annotated[str, Depends(oauth2_scheme)]):
    if token != "admin_secret_token":
        raise HTTPException(status_code=401, detail="Unauthorized")

@router.delete("/{student_id}", dependencies=[Depends(is_admin)])
def delete_student_secure(student_id: int):
    return {"message": "Sinh viên đã được xóa bởi admin"}
```

Đoạn code minh họa cách bảo vệ API bằng token trong FastAPI.

• `OAuth2PasswordBearer(tokenUrl="token")` - cách FastAPI yêu cầu client gửi kèm token trong request. Thông thường, token được gửi trong phần Authorization Header dưới dạng: `Authorization: Bearer <token>`

• `def is_admin(...)` - Hàm này là một dependency, dùng để kiểm tra token mà người dùng gửi lên. Nếu token khác "`admin_secret_token`" → trả về lỗi 401 Unauthorized. Nếu đúng → cho phép tiếp tục.

•`@router.delete("/student_id", ependencies=[Depends(is_admin)])` - API này định nghĩa một endpoint DELETE, dùng để xoá sinh viên theo ID. Nhưng trước khi chạy, FastAPI sẽ gọi is_admin() để đảm bảo chỉ admin mới có quyền.

Tóm lại: Nếu muốn gọi API xoá sinh viên, cần phải đưa ra *token admin,* nếu không có chìa khoá đúng, sẽ bị từ chối ngay.

Middleware: Middleware là các hàm chạy trước và sau mỗi yêu cầu, dựa trên chain of responsibility pattern. Nó hữu ích cho các tác vụ như xử lý CORS (Cross-Origin Resource Sharing), logging hoặc nén dữ liệu. Về lý thuyết, middleware giúp tách biệt cross-cutting concerns khỏi business logic. Tài liệu FastAPI gợi ý custom middleware cho advanced use như rate limiting với slowapi.

Chain of Responsibility là mẫu thiết kế nơi nhiều đối tượng xử lý một yêu cầu theo chuỗi, mỗi đối tượng quyết định tiếp tục hay dừng.

cors_middleware.py

```python
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

• `allow_origins=["*"]` Cho phép mọi domain (nguồn) đều có thể gửi request đến API. (Trong thực tế nên giới hạn cụ thể, ví dụ: ["https://myapp.com"]).

• `allow_credentials=True` Cho phép gửi kèm thông tin nhạy cảm như cookie, Authorization header.

• `allow_methods=["*"]` Cho phép tất cả các phương thức HTTP (GET, POST, PUT, DELETE, . . .).

• `allow_headers=["*"]` Cho phép mọi loại header trong request.

Đơn giản hóa: Middleware này giống như "cửa bảo vệ", quyết định xem trình duyệt từ đâu được phép gọi API. Ở đây thì mở cửa cho tất cả, nên tiện cho phát triển nhưng kém an toàn khi triển khai thật.

III. Triển khai mô hình với FastAPI

3.1. Mô hình Object Detection

Ở phàn này, chúng ta sẽ chọn một mô hình Machine learning, Deep Learning bất kỳ. Ở bài viết này, chúng ta sẽ sử dụng mô hình Object Detection được public ở link **HuggingFace** trên *yainage90/fashion-object-detection*.

Khi triển khai bất cứ mô hình nào, chúng ta đều cần đọc và hiểu rõ định dạng Input của mô hình này sử dụng.

Code mẫu quá trình inference của mô hình:

```python
from PIL import Image
import torch
from transformers import  AutoImageProcessor, AutoModelForObjectDetection

device = 'cpu'
if torch.cuda.is_available():
    device = torch.device('cuda')
elif torch.backends.mps.is_available():
    device = torch.device('mps')

```

```python
ckpt = 'yainage90/fashion-object-detection'
image_processor = AutoImageProcessor.from_pretrained(ckpt)
model = AutoModelForObjectDetection.from_pretrained(ckpt).to(device)

image = Image.open('<path/to/image>').convert('RGB')

with torch.no_grad():
    inputs = image_processor(images=[image], return_tensors="pt")
    outputs = model(**inputs.to(device))
    target_sizes = torch.tensor([[image.size[1], image.size[0]]])
    results = image_processor.post_process_object_detection(outputs, threshold=0.4,
                                                target_sizes=target_sizes)[0]

    items = []
    for score, label, box in zip(results["scores"], results["labels"], results["boxes"
                                                ]):
        score = score.item()
        label = label.item()
        box = [i.item() for i in box]
        print(f"{model.config.id2label[label]}: {round(score, 3)} at {box}")
        items.append((score, label, box))
```

Thông qua code mẫu, chúng ta biết chỉ cần convert Image sang RGB channel và sử dụng imageprocessor để xử lý ảnh trước khi đưa vào mô hình Object Detection chính (và mô hình này tất nhiên đã được pretrained). Qua đó, để thực hiện khởi tạo và detect ảnh, chúng ta sẽ triển khai 2 services như sau:

model_service.py

```python
import torch
from transformers import AutoImageProcessor, AutoModelForObjectDetection
from PIL import Image
from typing import List, Dict, Any
import time

from app.core.config import settings
from app.utils.logger import logger

class ModelService:
    def __init__(self):
        self.device = self._get_device()
        self.image_processor = None
        self.model = None
        self.load_model()

    def _get_device(self) -> torch.device:
        """Determine the best available device"""
        if torch.cuda.is_available():
            device = torch.device('cuda')
            logger.info("Using CUDA device")
        elif torch.backends.mps.is_available():
            device = torch.device('mps')
```

```
            logger.info("Using MPS device")
        else:
            device = torch.device('cpu')
            logger.info("Using CPU device")
        return device

    def load_model(self):
        """Load the model and processor"""
        try:
            logger.info(f"Loading model from {settings.MODEL_CHECKPOINT}")
            self.image_processor = AutoImageProcessor.from_pretrained(settings.
                                                MODEL_CHECKPOINT)
            self.model = AutoModelForObjectDetection.from_pretrained(settings.
                                                MODEL_CHECKPOINT).to(self.device)
            self.model.eval()
            logger.info("Model loaded successfully")
        except Exception as e:
            logger.error(f"Error loading model: {str(e)}")
            raise

    def preprocess_image(self, image: Image.Image) -> Dict[str, torch.Tensor]:
        """Preprocess image for model input"""
        return self.image_processor(images=[image], return_tensors="pt")

    def postprocess_detections(self, outputs, target_sizes, threshold: float) -> List[
                                                Dict[str, Any]]:
        """Postprocess model outputs into readable format"""
        results = self.image_processor.post_process_object_detection(
            outputs, threshold=threshold, target_sizes=target_sizes
        )[0]

        detections = []
        for score, label, box in zip(results["scores"], results["labels"], results["
                                                boxes"]):
            detection = {
                "label": self.model.config.id2label[label.item()],
                "score": round(score.item(), 4),
                "bounding_box": {
                    "xmin": round(box[0].item(), 2),
                    "ymin": round(box[1].item(), 2),
                    "xmax": round(box[2].item(), 2),
                    "ymax": round(box[3].item(), 2)
                }
            }
            detections.append(detection)

        return detections

    def detect_objects(self, image: Image.Image, threshold: float = None) -> Dict[str,
                                                Any]:
        """Main detection method"""
        if threshold is None:
            threshold = settings.DETECTION_THRESHOLD

```

```
        start_time = time.time()

        with torch.no_grad():
            inputs = self.preprocess_image(image)
            inputs = {k: v.to(self.device) for k, v in inputs.items()}

            outputs = self.model(**inputs)
            target_sizes = torch.tensor([[image.size[1], image.size[0]]]).to(self.
                                                device)

            detections = self.postprocess_detections(outputs, target_sizes, threshold)
            processing_time = time.time() - start_time

        return {
            "detections": detections,
            "processing_time": processing_time,
            "image_size": {"width": image.size[0], "height": image.size[1]}
        }

# Global model service instance
model_service = ModelService()
```

detect_service.py

```
from PIL import Image
import io
from typing import List, Dict, Any

from app.services.model_service import model_service
from app.models.responses import DetectionResponse, ErrorResponse
from app.utils.image_processor import image_processor
from app.utils.logger import logger

class DetectionService:
    @staticmethod
    def detect_from_bytes(image_bytes: bytes, threshold: float = None) ->
                                                DetectionResponse:
        """Detect objects from image bytes"""
        try:
            # Validate image
            if not image_processor.validate_image(image_bytes):
                return ErrorResponse(
                    success=False,
                    message="Invalid image file",
                    error_code="INVALID_IMAGE",
                    details={"file_type": "Unable to determine image format"}
                )

            # Convert to RGB PIL Image
            image = image_processor.convert_to_rgb(image_bytes)
            image_info = image_processor.get_image_info(image)

```

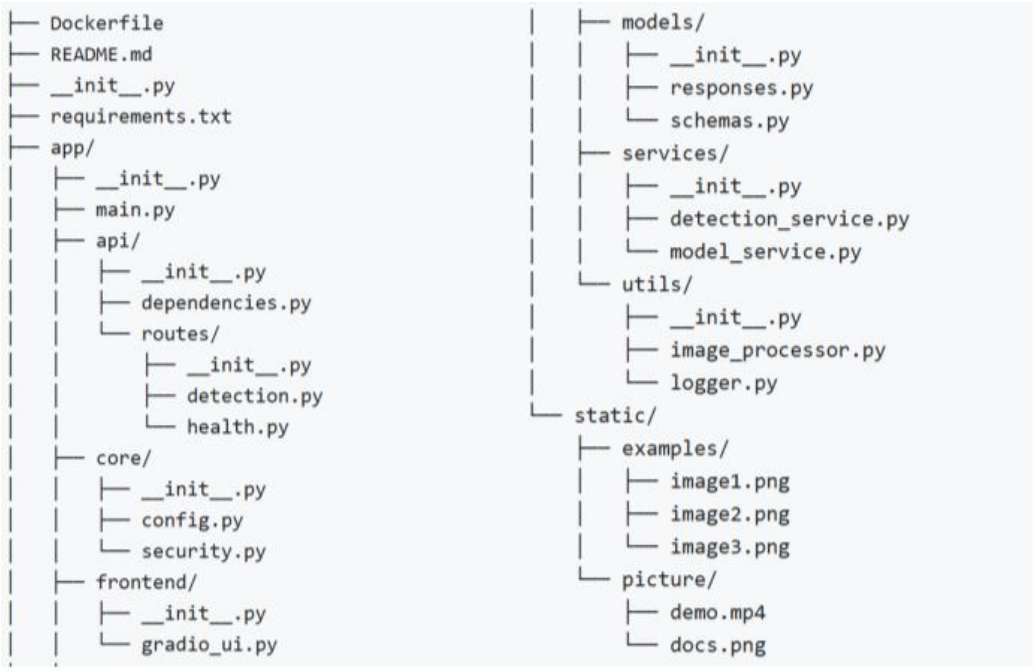
```
            # Detect objects
            result = model_service.detect_objects(image, threshold)

            return DetectionResponse(
                success=True,
                message="Detection completed successfully",
                detections=result["detections"],
                processing_time=round(result["processing_time"], 4),
                image_size=result["image_size"],
                total_detections=len(result["detections"])
            )

        except Exception as e:
            logger.error(f"Error in detection from bytes: {str(e)}", exc_info=True)
            return ErrorResponse(
                success=False,
                message="Failed to process image",
                error_code="PROCESSING_ERROR",
                details={"error": str(e)},
                stack_trace=str(e) if logger.level == 10 else None  # Only include
                                                                    stack trace in debug
            )

    @staticmethod
    def detect_from_pil(image: Image.Image, threshold: float = None) ->
                                                                    DetectionResponse:
        """Detect objects from PIL Image"""
        try:
            image_info = image_processor.get_image_info(image)

            # Detect objects
            result = model_service.detect_objects(image, threshold)

            return DetectionResponse(
                success=True,
                message="Detection completed successfully",
                detections=result["detections"],
                processing_time=round(result["processing_time"], 4),
                image_size=result["image_size"],
                total_detections=len(result["detections"])
            )

        except Exception as e:
            logger.error(f"Error in detection from PIL: {str(e)}", exc_info=True)
            return ErrorResponse(
                success=False,
                message="Failed to process image",
                error_code="PROCESSING_ERROR",
                details={"error": str(e)}
            )

    @staticmethod
    def get_annotated_image(image: Image.Image, detections: List[Dict[str, Any]]) ->
                                                                    Image.Image:
        """Get image with bounding boxes drawn"""
        return image_processor.draw_bounding_boxes(image, detections)
```

3.2. Triển khai Project FastAPi chuẩn

Project fashion object detection được upload tại fashion-detection-app. Chúng ta sẽ cùng nhau phân tích cấu trúc repository và flow đọc hiểu code.

Hiện tại, mô hình đã được triển khai sẵn sàng để xử lý input đầu vào theo yêu cầu. Do đó, trọng tâm của chúng ta là thiết kế API endpoint để nhận ảnh, xử lý, và trả về kết quả. Đối với API endpoint, chúng ta cần thiết kế một Route để nhận ảnh thông qua request, đồng thời quy định rõ ràng về schema mà hệ thống sẽ nhận và trả về. Dưới đây là phân tích chi tiết về cấu trúc repository và cách tổ chức code.

Tổ chức Thư mục



Hình 2: Cấu trúc project triển khai ML/DL Model bằng FastAPI

Cấu trúc thư mục của project được thiết kế theo mô hình modular, tuân thủ các tiêu chuẩn của một ứng dụng FastAPI, giúp dễ dàng mở rộng và bảo trì. Cụ thể:

- `app/`: Thư mục chính chứa toàn bộ mã nguồn ứng dụng.
  - `api/`: Chứa các định nghĩa route và endpoint của FastAPI. Ví dụ, một endpoint `/detect` sẽ được định nghĩa để nhận ảnh và trả về kết quả phát hiện.
  - `core/`: Chứa các cấu hình (như biến môi trường) và logic bảo mật (như xác thực JWT hoặc API key).

– `models/`: Chứa các schema Pydantic để validate dữ liệu đầu vào (request) và đầu ra (response), đảm bảo dữ liệu đúng định dạng.

– `services/`: Chứa logic mô hình, bao gồm tải mô hình Hugging Face (yainage90/fashion -object-detection), xử lý ảnh, và chạy inference.

– `utils/`: Các hàm tiện ích hỗ trợ, như xử lý file ảnh hoặc logging.

– `frontend/`: Giao diện người dùng được xây dựng bằng Gradio, cho phép upload ảnh và hiển thị kết quả trực quan.

– `main.py`: Điểm khởi đầu của ứng dụng FastAPI, nơi khởi tạo ứng dụng, include routes, và thiết lập middleware.

• `requirements.txt`: Liệt kê các thư viện Python cần thiết, như fastapi, uvicorn, transformers, pydantic, v.v.

• `Dockerfile`: Cấu hình để đóng gói ứng dụng vào container Docker, hỗ trợ triển khai production.

• `README.md`: Tài liệu hướng dẫn cách cài đặt, chạy, và sử dụng project.

Cấu trúc này đảm bảo tính tách biệt (separation of concerns), cho phép phát triển độc lập từng module. Ví dụ, có thể cập nhật mô hình trong services/ mà không ảnh hưởng đến giao diện trong frontend/.

3.3. Thiết kế API Endpoint

API endpoint là thành phần cốt lõi để nhận và xử lý ảnh đầu vào. Trong project này, một endpoint chính (ví dụ: POST /detect) được thiết kế để:

• Nhận input: Người dùng upload một file ảnh (định dạng JPEG, PNG, v.v.) thông qua request HTTP.

• Validate input: Sử dụng schema Pydantic trong models/ để kiểm tra tính hợp lệ của dữ liệu đầu vào (ví dụ: kiểm tra định dạng file, kích thước ảnh).

• Xử lý ảnh: Gọi các hàm từ services/ để tải mô hình, thực hiện inference, và trả về kết quả (boxes, labels, scores).

• Trả về response: Trả về kết quả dưới dạng JSON, sử dụng schema Pydantic để đảm bảo định dạng chuẩn.

api/detect.py

```
from fastapi import APIRouter, UploadFile, File, HTTPException, Depends, Query
from fastapi.responses import JSONResponse
from typing import Optional

```

```python
from app.services.detection_service import DetectionService
from app.models.schemas import DetectionResponse, ErrorResponse, DetectionRequest
from app.api.dependencies import get_token_header

router = APIRouter(
    prefix="/detect",
    tags=["detection"],
    dependencies=[Depends(get_token_header)],
    responses={401: {"description": "Unauthorized"}}
)

@router.post(
    "/image",
    response_model=DetectionResponse,
    responses={400: {"model": ErrorResponse}, 500: {"model": ErrorResponse}}
)
async def detect_objects(
    file: UploadFile = File(..., description="Image file to process"),
    threshold: Optional[float] = Query(None, description="Detection confidence
                                            threshold")
):
    """
    Detect fashion objects in an uploaded image.

    - **file**: Image file (JPEG, PNG, etc.)
    - **threshold**: Optional confidence threshold (default: 0.4)
    """
    # Validate file type
    if not file.content_type.startswith('image/'):
        raise HTTPException(
            status_code=400,
            detail="File must be an image (JPEG, PNG, etc.)"
        )

    # Read and process image
    try:
        image_bytes = await file.read()
        result = DetectionService.detect_from_bytes(image_bytes, threshold)

        if not result.success:
            raise HTTPException(status_code=500, detail=result.error)

        return result

    except Exception as e:
        raise HTTPException(
            status_code=500,
            detail=f"Error processing image: {str(e)}"
        )
```

3.4. Schema Dữ liệu

Schema được định nghĩa trong models/ sử dụng Pydantic để đảm bảo tính hợp lệ của dữ liệu.

Ví dụ:

- `DetectionRequest`: Xác định cấu trúc của input, thường chỉ cần UploadFile cho ảnh.

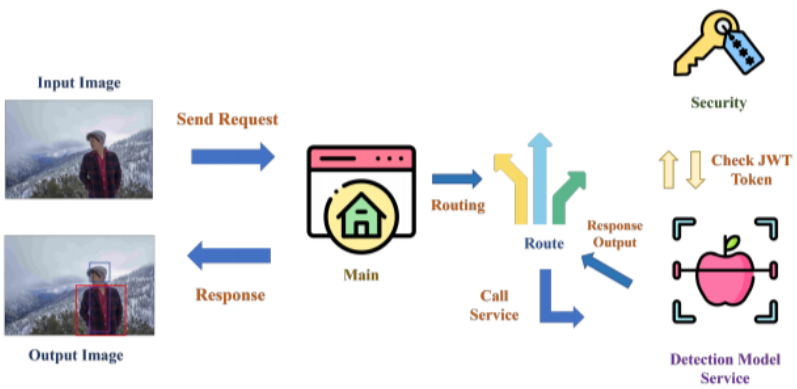
- `DetectionResponse`: Xác định cấu trúc của output, ví dụ danh sách các item được phát hiện với thông tin label, score, và box.

models/schemas.py

```python
from pydantic import BaseModel, Field
from typing import List, Optional
from datetime import datetime

class BoundingBox(BaseModel):
    xmin: float = Field(..., description="Left coordinate of bounding box")
    ymin: float = Field(..., description="Top coordinate of bounding box")
    xmax: float = Field(..., description="Right coordinate of bounding box")
    ymax: float = Field(..., description="Bottom coordinate of bounding box")

class DetectionResult(BaseModel):
    label: str = Field(..., description="Detected object class")
    score: float = Field(..., description="Confidence score")
    bounding_box: BoundingBox = Field(..., description="Bounding box coordinates")

class DetectionRequest(BaseModel):
    threshold: Optional[float] = Field(None, description="Detection threshold")

class DetectionResponse(BaseModel):
    success: bool = Field(..., description="Request success status")
    detections: List[DetectionResult] = Field(..., description="List of detected
                                                objects")
    processing_time: float = Field(..., description="Time taken to process the image")
    image_size: dict = Field(..., description="Original image dimensions")

class HealthResponse(BaseModel):
    status: str = Field(..., description="Service status")
    version: str = Field(..., description="API version")
    model_loaded: bool = Field(..., description="Model loading status")
    device: str = Field(..., description="Device used for inference")

class ErrorResponse(BaseModel):
    success: bool = Field(False, description="Request success status")
    error: str = Field(..., description="Error message")
    details: Optional[str] = Field(None, description="Additional error details")
```

3.5. Flow Đọc hiểu Code

Luồng xử lý code trong project tuân theo quy trình từ nhận request đến trả response:



Hình 3: Flow xử lý của app khi nhận request tới response

1. main.py: Khởi tạo ứng dụng FastAPI, include router từ api/, và thiết lập middleware (như CORS hoặc authentication từ core/).

2. Route (api/): Khi request đến (ví dụ: POST /detect), endpoint nhận file ảnh, validate schema (từ models/), và gọi hàm xử lý từ services/.

3. Security (core/): Kiểm tra quyền truy cập của request, như xác thực user qua JWT Key, trước khi xử lý request.

4. Services (services/): Thực hiện logic chính:

• Tải mô hình từ Hugging Face (yainage90/fashion-object-detection).

• Sử dụng PIL để xử lý ảnh đầu vào.

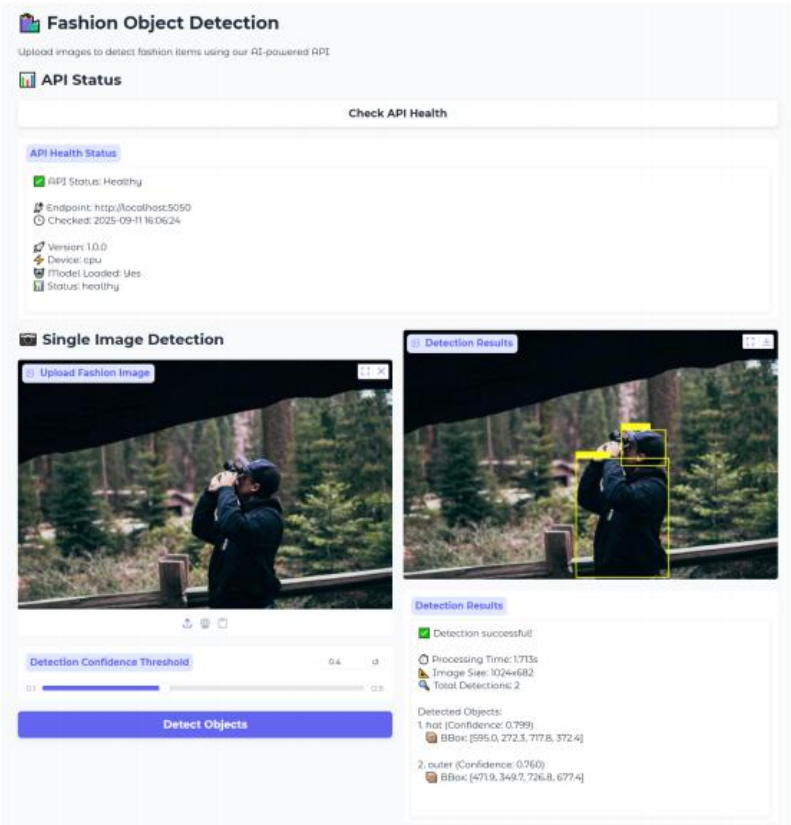
• Chạy inference với mô hình, trả về boxes, labels, scores.

• Xử lý kết quả để định dạng theo schema DetectionResponse.

5. Response: Kết quả từ services/ được đóng gói thành JSON và trả về client. Nếu có lỗi (như file ảnh không hợp lệ), FastAPI raise HTTPException.

3.6. Giao diện Gradio

Thực hiện cài đặt được hướng dẫn chi tiết ở **repo fashion-detection-app**, sau đó khởi chạy Gradio UI để thực hiện thử nghiệm detect ảnh theo batch hoặc từng ảnh một.



Hình 4: Giao diện trương tác với API endpoint triển khai bằng gradio

IV. Kết luận

FastAPI là một framework hiệu quả mà đơn giản, dễ triển khai. Đối với các AI reseacher hay AI engineer không quá giỏi software, FastAPi là một lựa chọn cực kỳ mạnh để chúng ta thực hiện serving model. Ngoài các phần đã được nói ở trong bài, mọi người có thể tìm hiểu thêm các khái niệm khác như kết nối database, testing API, logging, monitoring hay các kiến thức khác về security cho API. Từ đó phát triển được các ứng dụng tận dụng triệt để thế mạnh của mô hình mà mình xây dựng được trong quá trình nghiên cứu.