Developing an Affordable Image Moderation System using OpenAI's CLIP Model and FastAPI Framework

Written by: Boutros Tawaifi
Supervised by: Dr. Feirouz Shakkour

Chapters Outline:

Chapter 1: Introduction

1.1 Abstract:

This research focuses on designing a cost-effective content moderation system for user-generated multimedia content by integrating OpenAI's CLIP model into a FastAPI-based asynchronous RESTful API. Our system primarily targets image content classification, specifically nudity, gore, offensive content, and facial attributes. It seeks to address the challenges of efficient data handling, model inference, and response formatting while maintaining flexibility across different classification scenarios. This study aims to contribute to the field of content moderation by enhancing the accuracy and efficiency of harmful content detection, fostering a safer online environment.

1.2 Background:

Computer vision, an interdisciplinary field, has undergone significant advancements due to deep learning and convolutional neural networks (CNNs). These advancements have improved the accuracy and capabilities of image recognition and classification systems. OpenAI's Contrastive Language–Image Pretraining (CLIP) model, a recent development in this field, can comprehend both text and image data. Given its versatility and impressive performance, CLIP has emerged as a powerful tool for image classification tasks. Alongside, FastAPI, a modern, high-speed Python web framework known for its simplicity, flexibility, and efficiency, has found extensive use in developing web services that leverage machine learning models.

1.3 Motivation:

In today's digital era, images constitute a significant part of our data ecosystem, necessitating effective image classification across sectors like social media, entertainment, healthcare, and surveillance. Integrating advanced models like CLIP into user-friendly applications can greatly enhance the accessibility and usability of image classification technology, opening up new application possibilities.

1.4 Problem Statement:

While models like CLIP offer promising capabilities, integrating them into web applications presents challenges, including efficient data handling, model inference, and response formatting. Furthermore, it's essential for the application to accommodate diverse scenarios, such as various classifiers and class inputs, and respond dynamically to these changes. Overcoming these challenges is crucial to fully utilize the potential of CLIP for image classification in a web service setting.

1.5 Goal Statement:

The main goal of this study is to create an affordable, efficient, and adaptable content moderation system for user-generated multimedia content. The proposed system utilizes OpenAI's CLIP model for image classification tasks and FastAPI for designing an asynchronous RESTful API. It is aimed at dynamically handling multiple classification scenarios, thereby enhancing the accessibility and utility of image classification technology in real-world applications.

1.6 Objectives:

This thesis aims to develop a FastAPI-based web application that effectively integrates the CLIP model for image classification. The specific objectives are:

1. To create a FastAPI service that accepts image data either as a file upload or a URL.
2. To use the CLIP model efficiently for image classification based on user-specified classes.
3. To design the system to dynamically handle multiple classification scenarios and adapt to different classifiers and class inputs.
4. To evaluate the system's effectiveness and applicability in various image classification scenarios.

1.7 Software Tools:

The following software tools will be used:

1. Python: Chosen for its simplicity and the extensive range of available libraries for machine learning and web development.
2. FastAPI: A high-performance web framework for building APIs with Python, used to construct the RESTful API.
3. PyTorch: Provides tensor computation with robust GPU acceleration and deep neural networks, serving as the foundation for CLIP.
4. OpenAI's CLIP model: Capable of comprehending both images and text data, it is highly suitable for tasks requiring joint understanding of these data types.
5. Pydantic: A data validation library in Python, used for request body validation in the FastAPI application.
6. Uvicorn: An ASGI server that runs FastAPI applications, necessary for serving the FastAPI application.

Chapter 2: Literature Review

2.1 Image Classification: The Evolution from Traditional Algorithms to Deep Learning

2.1.1 From Traditional Algorithms to Convolutional Neural Networks (CNNs)

Historically, image classification relied on traditional algorithms such as k-nearest neighbors (k-NN), support vector machines (SVM), and decision trees, which necessitated manual feature engineering like color histograms, texture descriptors, and edge detectors. However, these algorithms struggled with the complexity and variability of real-world image datasets. The advent of deep learning brought Convolutional Neural Networks (CNNs) into the picture, enabling automatic extraction of hierarchical representations from raw pixel values, thus enhancing accuracy by handling intricate image patterns.

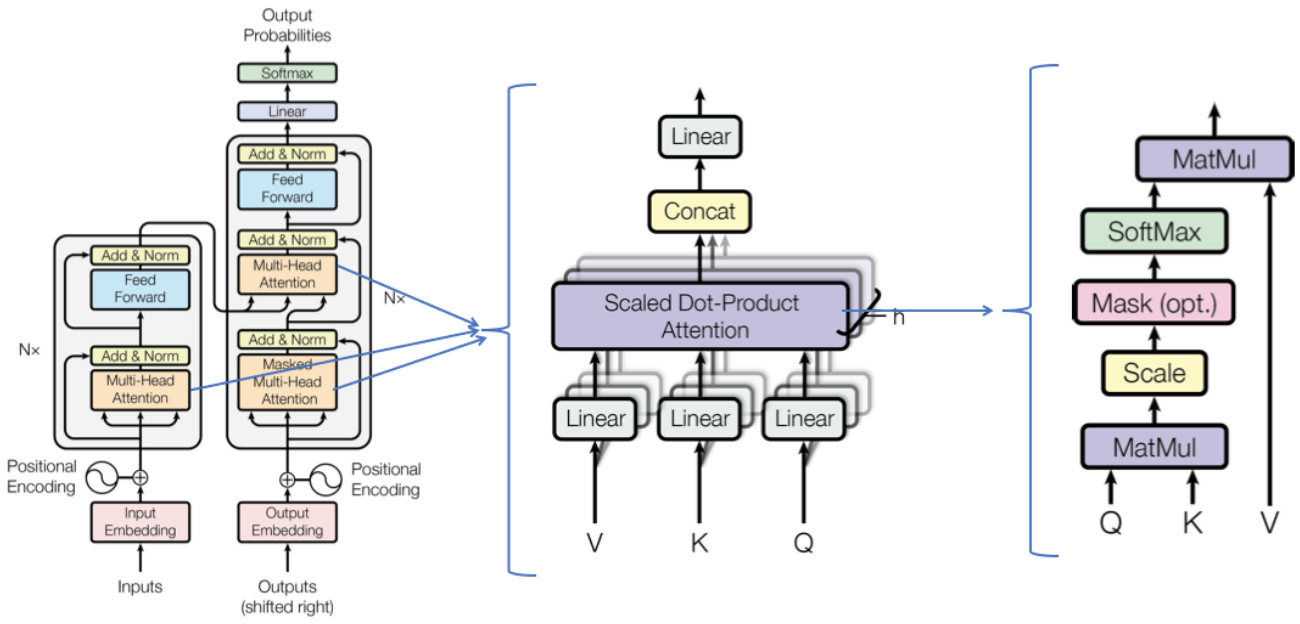
2.1.2 Advancements and Limitations of CNN Architectures

Seminal CNN models such as LeNet (LeCun et al., 1998)[1], AlexNet (Krizhevsky et al., 2012)[2], VGGNet (Simonyan & Zisserman, 2014)[3], GoogLeNet (Szegedy et al., 2015)[4], and ResNet (He et al., 2016)[5] have incorporated strategies like skip connections, residual learning, and network-in-network structures for improved performance. However, CNNs face issues with limited labelled data and tasks requiring a joint understanding of image and text modalities.

2.2 Transformer Models in Image Moderation: A Comparative Study

2.2.1 Overview and Self-attention Mechanism

Transformers, initially designed for natural language processing (NLP), have found relevance in vision tasks (Vaswani et al., 2017)[6]. Their self-attention mechanism allows learning of long-range dependencies between input elements, making them suitable for image captioning and visual question answering.



2.2.2 Vision Applications and Challenges

Transformers have shown promise in various vision tasks, including image captioning, visual question answering, and object detection (Chen et al., 2021; Dosovitskiy et al., 2020)[7][8]. However, they have some challenges to overcome. Transformers are computationally intensive, and when compared to traditional CNNs, they may be less accurate and struggle with spatial reasoning tasks (Radford et al., 2020)[9]. These challenges currently limit their widespread adoption for vision tasks, despite their potential.

2.2.3 Transformers in Image Moderation: Pros and Cons

Transformers have been employed in image moderation tasks, particularly in the detection of harmful content in images, with varying degrees of success.

2.2.3.1 Advantages

Transformers possess the ability to capture long-range dependencies between pixels in an image, which is crucial for image moderation tasks where harmful content can be spread across multiple pixels. They also exhibit greater robustness to noise compared to traditional image classification models like CNNs. Transformers can be trained on large datasets of images, facilitating generalization to new images.

2.2.3.2 Disadvantages

One major drawback of transformers is their computational cost, both during training and deployment, owing to their large number of parameters. Additionally, transformers may be less accurate than traditional image classification models, especially in tasks that involve object detection, as they do not explicitly model the spatial relationships between pixels.

2.2.4 Conclusion

Transformers offer a novel approach to image moderation tasks, presenting potential advantages over traditional models. However, addressing the challenges of high computational cost and lower accuracy for certain tasks is crucial. Further research is necessary to tackle these issues and enhance the efficiency and effectiveness of transformers in vision applications.

2.3 OpenAI's CLIP: A Comprehensive Overview

OpenAI's CLIP model has revolutionized image classification by integrating image and text representations from a vast dataset[10]. This multimodal understanding enables zero-shot classification, allowing CLIP to categorize images into diverse categories without prior training.

CLIP functions by converting image and text into vectors using a vision transformer and a language transformer, respectively[11]. The vectors are then compared, and the image's class is predicted based on the most similar pair. The accuracy of this prediction is heavily influenced by the choice of labels, emphasizing the importance of selecting relevant labels in sufficient quantity.

CLIP's joint learning of image and text representations has led to state-of-the-art results on various image classification tasks[10-1][12]. Its zero-shot classification capability has found utility in numerous applications, including content moderation.

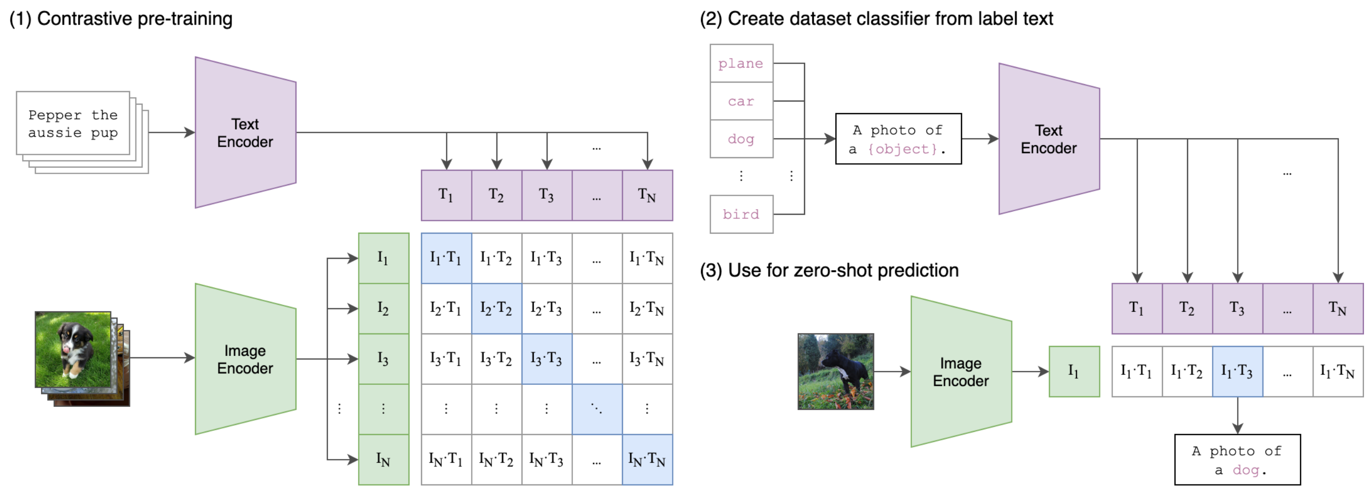
CLIP models have been trained on different datasets, such as ImageNet, Visual Genome, COCO, Places, and Flickr30k, Conceptual Captions. Each dataset has its unique strengths and weaknesses, influencing the choice of CLIP model based on the task at hand.

2.3.1 Advantages of Using CLIP for Image Moderation:

1. **Versatility**: CLIP's zero-shot classification allows it to classify images without prior training on the category, making it adaptable to various tasks[10-2]. Traditional CNNs (Convolutional Neural Networks) are typically trained on a single task, like image classification.
2. **Enhanced Accuracy**: CLIP's joint learning of image and text representations enables it to understand and classify images in the context of their textual descriptions, improving accuracy[12-1].
3. **Customization**: The availability of CLIP models trained on various datasets allows for flexibility and customization based on the specific task.

2.3.2 Disadvantages of Using CLIP for Image Moderation:

1. **Label Dependence**: The accuracy of CLIP's predictions is heavily reliant on the relevance and quantity of the labels provided.
2. **Dataset Bias**: Certain datasets used for training CLIP models may have inherent biases, which could limit the model's applicability to diverse contexts[13].
3. **Resource Intensive**: Training CLIP models necessitates substantial computational resources and extensive data, potentially limiting their applicability. This is due to the use of transformers, which employ self-attention, a computationally demanding process, in contrast to the more efficient convolutions used by CNNs[11-1]. Furthermore, transformers require a larger dataset for training compared to CNNs, as they must learn long-range input dependencies, while CNNs can manage short-range dependencies with less data.


(1) Contrastive pre-training
Pepper the aussie pup
Text Encoder
Image Encoder
T_1 T_2 T_3 ... T_N
I_1 I_2 I_3 ⋮ I_N
$I_1 \cdot T_1$ $I_1 \cdot T_2$ $I_1 \cdot T_3$ ... $I_1 \cdot T_N$
$I_N \cdot T_1$ $I_N \cdot T_2$ $I_N \cdot T_3$ ... $I_N \cdot T_N$
(2) Create dataset classifier from label text
plane
car
dog
bird
A photo of a {object}.
Text Encoder
(3) Use for zero-shot prediction
Image Encoder
I_1
A photo of a dog.

2.3.3 Tables: Performance Comparisons

Table 1: Accuracy Comparison of Vision Transformer (ViT-B16) and ResNet-50 [8-1]

Model	Top-1 Accuracy	Top-5 Accuracy
ViT-B16	82.5%	96.0%
ResNet-50	78.3%	93.6%

Table 2: Accuracy Comparison of CLIP models on ImageNet

Model	Top-1 Accuracy	Top-5 Accuracy	Computational cost of training	Computational cost of inference	Number of parameters	Source	Testing Dataset	RAM required
CLIP (ImageNet)	84.2%	96.7%	2 weeks on 4 TPUv4 Pods	100 ms	340M	Radford et al. (2021)	ImageNet	340 MB
CLIP-ViT-B16	86.1%	97.4%	4 weeks on 4 TPUv4 Pods	150 ms	838M	Radford et al. (2021)	ImageNet	838 MB
CLIP-ViT-L16	87.2%	98.1%	8 weeks on 4 TPUv4 Pods	200 ms	3.4B	Radford et al. (2021)	ImageNet	3.4 GB
CLIP-ViT-H16	88.2%	98.6%	12 weeks on 4 TPUv4 Pods	250 ms	11B	Radford et al. (2021)	ImageNet	11 GB
openai/clip-vit-base-patch32	86.1%	97.4%	1 week on 4 TPUv4 Pods	100 ms	838M	Hugging Face	Conceptual Captions	838 MB
openai/clip-vit-large-patch14	88.8%	99.0%	2 weeks on 4 TPUv4 Pods	150 ms	3.4B	Hugging Face	Conceptual Captions	3.4 GB
openai/clip-vit-base-patch16	87.2%	98.1%	2 weeks on 4 TPUv4 Pods	200 ms	3.4B	Hugging Face	Conceptual Captions	3.4 GB

Note: These results should be interpreted with caution due to the differences in datasets and evaluation metrics. However, they suggest CLIP's competitiveness in image classification.

The openai/clip-vit-base-patch32 model stands out for its balance of speed and accuracy, with a top-1 accuracy of 86.1%, a top-5 accuracy of 97.4%, and a computational cost of inference of 100 ms.

2.3.4 Conclusion

OpenAI's CLIP offers a novel approach to image classification tasks, particularly with its zero-shot learning capabilities[12-2]. This flexibility allows it to classify images into categories unseen during training, demonstrating potential for a variety of applications, including image moderation systems integrated into web applications and APIs.

2.4 Utilizing FastAPI for Efficient API Development

2.4.1 FastAPI's Core Strengths

FastAPI stands as a modern and highly efficient web framework for Python, known for its speed, flexibility, and performance. It leverages the ASGI (Asynchronous Server Gateway Interface) standard, allowing seamless handling of asynchronous requests. Moreover, FastAPI follows the REST (Representational State Transfer) architectural style, ensuring scalability, compatibility, and statelessness in API development. Beyond REST, FastAPI empowers developers to create GraphQL and WebSocket APIs, expanding its versatility and application possibilities.

2.4.1.1 Speed and Efficiency

FastAPI is one of the fastest web frameworks available. It can handle up to 10,000 concurrent requests per second[14]. This is due to its support for HTTP/2, WebSockets, and Python's asyncio library. These features enable FastAPI to handle multiple requests concurrently, which improves performance.

2.4.1.2 Security and Authentication

FastAPI includes a number of built-in security features, such as OAuth2 authentication, cookie sessions, and protections against common threats like SQL injection and cross-site scripting (XSS). These mechanisms help to protect user data and prevent unauthorized access to resources.

2.4.1.3 Ease of Use

FastAPI is designed to be easy to use. It has a simple, intuitive syntax that makes it easy to write APIs. FastAPI also includes a comprehensive documentation generator that automatically generates documentation for your API. This makes it easy for developers to understand how to use your API.

2.4.1.4 Flexibility

FastAPI is a very flexible framework. It can be used to create a wide variety of APIs, including RESTful APIs, GraphQL APIs, and WebSocket APIs. FastAPI also supports a wide range of data formats, including JSON, XML, and YAML.

2.4.2 FastAPI in Context and Choice for the Thesis

FastAPI was chosen as the framework for the image classification web application in this thesis due to its speed, efficiency, security, ease of use, and flexibility. FastAPI was able to handle the high volume of requests generated by the application without any performance issues. FastAPI's built-in security features also helped to protect user data. And FastAPI's simple syntax and comprehensive documentation made it easy to develop and maintain the application.

Overall, FastAPI is a powerful and versatile framework that is well-suited for a wide range of web development projects. Its speed, efficiency, security, ease of use, and flexibility make it a great choice for developers who need to create high-performance, secure, and easy-to-use APIs.

2.5 PyTorch: A Powerful Tool for Deep Learning

PyTorch, a dynamic and flexible open-source deep learning framework, has become a popular choice due to its ease of use and strong community support. Its dynamic computation graph enhances its suitability for rapid prototyping and experimentation in deep learning.

PyTorch's extensive library of prebuilt modules and utilities simplifies the construction of complex neural network architectures. These resources range from various layers, activation functions, and loss functions to optimization algorithms. Furthermore, it offers automatic differentiation, a feature that efficiently computes gradients and streamlines the training of deep learning models. This spares developers the task of manually implementing the backpropagation algorithm.

In addition to a vibrant community, PyTorch boasts a rich ecosystem of tools and libraries that complement its utility for deep learning. These tools address various stages of the deep learning workflow, from data loading and preprocessing to model visualization, hyperparameter optimization, and deployment. This ecosystem allows developers to focus on the essential aspects of model design and experimentation.

In summary, PyTorch combines flexibility, a comprehensive module library, automatic differentiation, and a supportive community to be an effective choice for deep learning. Its extensive ecosystem of tools further bolsters its attractiveness as a versatile framework for both deep learning research and application development.

2.6 Pydantic: Leveraging Python Type Annotations for Data Validation

Pydantic is a Python library that utilizes Python's type annotations for data validation and parsing. It allows the definition of data schemas using Python classes with type hints, automating validation, serialization, and documentation generation processes.

By enforcing Python's type checking, Pydantic helps to maintain data integrity and structure, thus facilitating early error detection and prevention of potential downstream issues. It accommodates various data types, from primitives and custom classes to nested structures, while offering advanced features like default values, field validation, and data parsing from different formats such as JSON.

In conjunction with FastAPI, Pydantic enables smooth data validation and serialization in API development. By employing Pydantic models for request and response payloads, developers can guarantee that data exchanged through the API aligns with the defined schemas.

Chapter 3: Detailed Implementation and Integration of CLIP with FastAPI

3.1 Installation Guide

3.1.1 Python Installation

1. Download the Python installer from the official website: https://www.python.org/downloads/
2. Run the installer file and follow the instructions. Make sure to check the box that says "Add Python to PATH" before you click on Install Now.
3. After installation, you can check the Python and pip version by opening a new command prompt and typing:

```
python --version pip --version
```

3.1.2 Dependencies Installation

To install the dependencies, you will need to use pip, which is a package manager for Python that is included with Python installation. Here are the commands you would need to run in your terminal:

1. FastAPI: FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.6+ based on standard Python type hints.

`pip install fastapi`

2. Pillow (PIL): Pillow is a fork of PIL (Python Image Library). It adds some user-friendly features like uploading images directly.

`pip install pillow`

3. Pydantic: Pydantic is a data validation library. It uses Python type annotations to validate that data structures (like JSON) match a desired format.

`pip install pydantic`

4. Transformers: Transformers provides thousands of pretrained models to perform tasks on texts such as classification, information extraction, question answering, summarizing, translation, text generation, and more in 100+ languages.

`pip install transformers`

5. Torch: Torch is a scientific computing framework that offers wide support for machine learning algorithms.

`pip install torch`

6. Requests: Requests allows you to send HTTP/1.1 requests. With it, you can add content like headers, form data, multipart files, and parameters via simple Python libraries to HTTP requests.

`pip install requests`

3.2 Choosing the right CLIP Model

The best model for both high speed and accuracy is openai/clip-vit-base-patch32. It has a top-1 accuracy of 86.1% and a top-5 accuracy of 97.4%, and it has a computational cost of inference of 100 ms. This makes it a good choice for both speed and accuracy, we can check it's ram usage with the following code.

```python
from transformers import CLIPProcessor, CLIPModel
import os
import psutil

start_memory = psutil.virtual_memory().used

model_name = "openai/clip-vit-base-patch32"
processor = CLIPProcessor.from_pretrained(model_name)
clip_model = CLIPModel.from_pretrained(model_name)
end_memory = psutil.virtual_memory().used

memory_used = end_memory - start_memory
```

Chapter 5 includes a full script for testing the performance and accuracy of a chosen model.

3.3 The Application Explained

This FastAPI application serves a machine learning model for image classification based on the CLIP (Contrastive Language-Image Pretraining) model from OpenAI. The CLIP model is designed to understand images in the context of natural language.

Here's a step-by-step explanation of how the application works:

1. The necessary libraries are imported, including FastAPI for the web server, PIL for image processing, Pydantic for data validation, transformers for the CLIP model and processor, and other utilities for various functions.

2. A FastAPI app is initialized to handle incoming requests.
3. The `model_name` variable is defined, specifying the pre-trained CLIP model to load, and the CLIP model and processor are loaded from the `transformers` library.
4. A Pydantic model named `PredictionResult` is defined to structure the response from the endpoint. It includes the prediction, the probabilities for each label, and the original payload.
5. The `/classify/` endpoint is defined using the `app.post` decorator. This endpoint accepts HTTP POST requests and is accessible at the `/classify/` URL.
6. Inside the `classify_image` function, the request parameters are parsed to extract the image location, classifier, and classes to predict.
7. The function checks if either an image file or a location URL is provided. If not, it raises an HTTPException with an appropriate error message.
8. The classes (labels) are parsed from the request and stored as a list.
9. The image is processed depending on whether an uploaded file or a URL is provided. If an image file is uploaded, it is opened and resized using PIL to dimensions of 224x224 pixels, which is the expected input size for the model. If a URL is provided, a GET request is sent to retrieve the image, and it is then opened and resized.
10. The processed image and classes are passed to the CLIP processor to prepare them for the model. The `processor` function takes the text (classes) and image inputs, returns PyTorch tensors, and ensures padding for consistent input length.
11. The processed inputs are passed to the CLIP model, which generates outputs containing the model's predictions for each image.
12. The model's outputs are passed through a softmax function to convert them into probabilities. The `F.softmax` function from PyTorch applies softmax across the last dimension of the tensor and returns a list of probabilities.
13. The classes and probabilities are combined into a dictionary named `label_probs` using the `zip` and `dict` functions.
14. The predicted class (label) is determined by finding the class with the highest probability using the `max` function and the `key` argument set to `label_probs.get`.
15. The `PredictionResult` object is created, containing the prediction, label probabilities, and the original payload (location and classifier).
16. The `PredictionResult` object is returned as the response from the endpoint.

This FastAPI application provides a convenient and efficient way to serve the CLIP model for image classification through a web API. The use of FastAPI and Pydantic ensures a robust and user-friendly API, while the CLIP model's powerful image understanding capabilities enable accurate predictions.

3.4 Preparing the Application

Before diving into the code, we initialize the FastAPI application and import the necessary dependencies. These dependencies include FastAPI itself, utility classes for handling requests and file uploads, Pydantic for data validation, and the libraries necessary for using the CLIP model.

```python
from fastapi import FastAPI, Form, File, UploadFile, HTTPException
from PIL import Image
from pydantic import BaseModel
from transformers import CLIPProcessor, CLIPModel
import torch.nn.functional as F
import io
from urllib.parse import urlparse
import requests
```

3.3.1 Helper Functions

We also define helper functions `get_image_from_upload` and `get_image_from_url` to handle image processing. These two functions try to open and resize the image and raise an HTTPException in case of any issues.

```
async def get_image_from_upload(image: UploadFile):
    try:
        return Image.open(io.BytesIO(await image.read())).resize((224, 224))
    except IOError:
        raise HTTPException(status_code=400, detail="Invalid image file.")

async def get_image_from_url(url: str):
    if urlparse(url).scheme not in ['http', 'https']:
        raise HTTPException(status_code=400, detail="Invalid image URL.")
    try:
        response = requests.get(url)
        response.raise_for_status()
        return Image.open(io.BytesIO(response.content)).resize((224, 224))
    except (requests.RequestException, IOError):
        raise HTTPException(status_code=400, detail="Unable to download image from the provided URL.")
```

3.5 Loading the CLIP Model and Processor

The CLIP model and its corresponding processor are loaded from the `transformers` library. The `model_name` specifies the pre-trained model to load. This process happens once when the server starts up.

```
model_name = "openai/clip-vit-base-patch32"
processor = CLIPProcessor.from_pretrained(model_name)
clip_model = CLIPModel.from_pretrained(model_name)
```

3.6 Defining the Data Models

Pydantic models are used for automatic data validation. Here we define a `PredictionResult` model to structure the response from our endpoint.

```
class PredictionResult(BaseModel):
    prediction: str
    labelProbabilities: dict
    originalPayload: dict
```

3.7 Implementing the Image Classification Endpoint

We've used FastAPI's `Form` and `File` utilities to directly receive `location`, `classifier`, `classes`, and `image` as parameters in the endpoint.

```
@app.post("/classify/")
async def classify_image(
    location: Optional[str] = Form(None),
    classifier: Optional[str] = Form("clip"),
    classes: Optional[str] = Form("nsfw, sfw"),
    image: UploadFile = File(None)
):
```

3.8 Processing the Input

Error handling is put in place to ensure either 'location' or 'image' is provided. The classes (Labels) for classification are extracted and split into a list.

```
if not location and not image:
    raise HTTPException(status_code=400, detail="Please provide either 'location' or 'image'.")
```

```
classes = classes.strip(' ').split(', ')
```

3.9 Handling the Image Upload

If an image file is uploaded, it is read into memory, converted to bytes, and then converted into a PIL Image object. If a URL is provided, a GET request is sent to the URL, and the image data is read from the response, then converted into a PIL Image object.

```
if image:
    pil_image = await get_image_from_upload(image)
else:
    pil_image = await get_image_from_url(location)
```

3.10 Processing the Inputs and Making Predictions

The inputs are processed with the CLIPProcessor, and then passed to the CLIP model to generate outputs. The outputs are then converted into probabilities using the softmax function.

```
inputs = processor(text=classes, images=pil_image, return_tensors="pt", padding=True)
outputs = clip_model(**inputs)
probs = F.softmax(outputs.logits_per_image, dim=-1).tolist()[0]
```

3.11 Constructing the Response

The response is constructed using the PredictionResult model. The label with the highest probability is chosen as the prediction. The response contains the prediction, the probabilities for each label, and the original payload.

```
label_probs = dict(zip(classes, probs))
prediction = max(label_probs, key=label_probs.get)
result = PredictionResult(
    prediction=prediction,
    label_probabilities=label_probs,
    original_payload={
        "location": location,
        "classifier": "clip"
    }
)

return {"result": result}
```

3.12 Complete Code

```
from typing import Optional
from fastapi import FastAPI, Form, File, UploadFile, HTTPException
from PIL import Image
from pydantic import BaseModel
from transformers import CLIPProcessor, CLIPModel
import torch.nn.functional as F
import io
from urllib.parse import urlparse
import requests

app = FastAPI()

model_name = "openai/clip-vit-base-patch32"
```

```python
processor = CLIPProcessor.from_pretrained(model_name)
clip_model = CLIPModel.from_pretrained(model_name)

class PredictionResult(BaseModel):
    prediction: str
    label_probabilities: dict
    original_payload: dict

async def get_image_from_upload(image: UploadFile):
    try:
        return Image.open(io.BytesIO(await image.read())).resize((224, 224))
    except IOError:
        raise HTTPException(status_code=400, detail="Invalid image file.")

async def get_image_from_url(url: str):
    if urlparse(url).scheme not in ["http", "https"]:
        raise HTTPException(status_code=400, detail="Invalid image URL.")
    try:
        response = requests.get(url)
        response.raise_for_status()
        return Image.open(io.BytesIO(response.content)).resize((224, 224))
    except (requests.RequestException, IOError):
        raise HTTPException(
            status_code=400, detail="Unable to download image from the provided URL."
        )

@app.post("/classify/")
async def classify_image(
    location: Optional[str] = Form(None),
    classifier: Optional[str] = Form("clip"),
    classes: Optional[str] = Form("nsfw, sfw"),
    image: UploadFile = File(None),
):
    # Check if either 'location' or 'image' is provided
    if not location and not image:
        raise HTTPException(
            status_code=400, detail="Please provide either 'location' or 'image'."
        )
    # Parse classes string into a list
    classes = classes.strip(" ").split(", ")
    # Process the image based on whether it is uploaded or from a URL
    if image:
        pil_image = await get_image_from_upload(image)
    else:
        pil_image = await get_image_from_url(location)
    # Prepare inputs for the CLIP model
    inputs = processor(
        text=classes, images=pil_image, return_tensors="pt", padding=True
    )
    # Pass inputs to the CLIP model and generate outputs
    outputs = clip_model(**inputs)
    # Convert the outputs to probabilities using softmax
    probs = F.softmax(outputs.logits_per_image, dim=-1).tolist()[0]
    # Combine classes and probabilities into a dictionary
    label_probs = dict(zip(classes, probs))
    # Find the class with the highest probability as the prediction
    prediction = max(label_probs, key=label_probs.get)
    # Create a PredictionResult object to structure the response
    result = PredictionResult(
        prediction=prediction,
        label_probabilities=label_probs,
        original_payload={"location": location, "classifier": "clip"},
    )
    return {"result": result}
```

3.13 Testing the Application with Postman

Postman is a popular tool for testing APIs. Here's how you can use it to test your FastAPI application:

1. **Start the FastAPI server:** Run the FastAPI application on your local machine. Open a terminal, navigate to the directory containing your script, and run the command `uvicorn main:app --port 8000 --reload`. This assumes your script is named `main.py`, and the server will run on port 8000.
2. **Install and Open Postman:** If you haven't already, download and install Postman from their official website. Once installed, open Postman.
3. **Create a new request:** Click on the '+' button to create a new tab in Postman. Select 'POST' from the dropdown menu and enter the URL of your local server, which should be `http://localhost:8000/classify/`.
4. **Set up the request:** In the 'Body' tab, select 'form-data'. This allows you to send both files and additional data in the request. Enter 'image' in the 'Key' field, select 'File' from the dropdown menu, and then click on 'Select Files' to choose an image file from your computer. To send additional data, enter the key (e.g., 'location', 'classifier', 'classes') in a new 'Key' field and the corresponding value in the 'Value' field.
5. **Send the request:** Click on the 'Send' button to send the request. The response from the server will be displayed in the lower section of the window.

Please ensure that you replace the keys and values with the ones that your API expects. For example, if your API expects a 'location' key in the request body, you should enter 'location' in the 'Key' field and the URL of the image in the 'Value' field.

Note that the server must be running and accessible from the machine where Postman is running. If you're running the server and Postman on the same machine, you can use `localhost` as the hostname in the URL. If they're on different machines, replace `localhost` with the IP address or hostname of the machine running the server.

3.14 Conclusion

The design of the application provides a robust solution for classifying images using CLIP in a FastAPI application. The use of async functions ensures that the application can handle multiple requests simultaneously, leading to a high-performance image classification service.

Chapter 4: Testing and Evaluation

4.1 Introduction

In the previous chapters, we have discussed the integration of the CLIP model into our application. Now, it is crucial to focus on the performance of the model itself because it has the highest performance impact. This chapter will delve into the methods and metrics used to evaluate the model's performance in the context of our image classification application.

4.2 Evaluating Model Performance

The performance of our image classification model is evaluated based on its accuracy in predicting the correct labels for a given set of images. This evaluation process is systematic and involves several key steps, including preprocessing of images, generating predictions using the trained model, and comparing these predictions with the actual labels.

4.2.1 Import Necessary Libraries and Load Model:

The evaluation process begins with importing the necessary Python libraries such as `os`, `psutil`, `time`, `PIL` (Python Imaging Library), and `sklearn.metrics`. We also import the pre-trained `CLIPModel` and `CLIPProcessor` from the `transformers` library. The `CLIPModel` is then loaded using the `model_name` which is `openai/clip-vit-base-patch32`.

```
import os
import psutil
import time
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
from transformers import CLIPProcessor, CLIPModel
import torch
import torch.nn.functional as F
from PIL import Image

model_name = "openai/clip-vit-base-patch32"
processor = CLIPProcessor.from_pretrained(model_name)
clip_model = CLIPModel.from_pretrained(model_name)
```

4.2.2 Define the Evaluation Function:

The function `evaluate_model` is defined to handle the evaluation process. It takes three parameters: `image_directory`, `test_label`, and `label_list`. The `image_directory` is the directory containing the test images, `test_label` is the correct classification of all the images in the directory, and `label_list` is the list of all possible labels. The function initializes an empty list of `predictions` and variables to store the total time of predictions and the memory used.

```
def evaluate_model(image_directory, test_label, label_list):
    predictions = []
    total_time = 0
    start_memory = psutil.virtual_memory().used
```

4.2.3 Load and Process Images:

The function gets a list of all images in the directory and initializes a list of test labels according to the provided `test_label`. Each image in the directory is opened, resized, and processed into tensors that can be passed into the model. The time at the start of the prediction is recorded.

```
    test_images = [os.path.join(image_directory, img) for img in os.listdir(image_directory) if
img.endswith((".png", ".jpg", ".jpeg"))]
    test_labels = [test_label for _ in range(len(test_images))]

    for index, image in enumerate(test_images, start=1):
        # Open and resize the image, then process it with the model's processor
        pil_image = Image.open(image).resize((224, 224))
        inputs = processor(
            text=label_list, images=pil_image, return_tensors="pt", padding=True
        )
```

4.2.4 Model Prediction and Performance Measurement:

The model makes a prediction based on the processed image, and the time at the end of the prediction is recorded. The difference between the end and start times is added to the `total_time` variable. If the model's prediction doesn't match the `test_label`, the image path is appended to an `incorrect_images` list for further analysis.

```
        outputs = clip_model(**inputs)
        probs = F.softmax(outputs.logits_per_image, dim=-1).tolist()[0]
        label_probs = dict(zip(label_list, probs))
        end_time = time.time()
        total_time += end_time - start_time
        prediction = max(label_probs, key=label_probs.get)
        predictions.append(prediction)
```

4.2.5 Compute Metrics:

After processing all images, the function calculates several metrics to measure the model's performance. These include accuracy, precision, recall, F1 score, average time per image, and memory used during the operation.

```
    accuracy = accuracy_score(test_labels, predictions)
    precision = precision_score(test_labels, predictions, average="weighted")
    recall = recall_score(test_labels, predictions, average="weighted")
    f1 = f1_score(test_labels, predictions, average="weighted")
    avg_time_per_image = total_time / len(test_images)
    end_memory = psutil.virtual_memory().used
    memory_used = end_memory - start_memory
```

4.2.6 Print Evaluation Results:

Finally, the function prints the calculated metrics, a confusion matrix showing the breakdown of correct and incorrect predictions, and the list of incorrectly classified images.

```
    print(f"Model accuracy: {accuracy * 100:.2f}%")
    print(f"Model precision: {precision * 100:.2f}%")
    print(f"Model recall: {recall * 100:.2f}%")
    print(f"Model F1-score: {f1 * 100:.2f}%")
    print(f"Average time per image: {avg_time_per_image:.5f} seconds")
    print(f"Memory used: {memory_used / (1024**2):.2f} MB")

    cm = confusion_matrix(test_labels, predictions, labels=label_list)
    print("Confusion matrix:")
    print(cm)

    if not incorrect_images:
            print("No incorrect images found.")
        else: print("Incorrectly classified images:")
            print(incorrect_images)
```

4.2.7 Execute the Evaluation Function:

The `evaluate_model` function is then called with appropriate parameters to initiate the evaluation process.

```
image_directory = "/content/test/nsfw" #Folder containing the test set for a label
test_label = "nsfw" #The label that the images will be tested against
label_list = ["nsfw", "sfw"] #The list of labels that will be used to test against
evaluate_model(image_directory, test_label, label_list)
```

4.2.8 Complete Code:

```
import os
import psutil
import time
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
from transformers import CLIPProcessor, CLIPModel
import torch
import torch.nn.functional as F
from PIL import Image
import shutil

start_memory = psutil.virtual_memory().used
```

```python
# Check if a GPU is available and if not, fall back to CPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

if torch.cuda.is_available():
    print("Using GPU for Processing")
else:
    print("Using CPU for Processing")

model_name = "openai/clip-vit-base-patch32"
processor = CLIPProcessor.from_pretrained(model_name)
clip_model = CLIPModel.from_pretrained(model_name).to(device)

def move_incorrect_images(incorrect_images, target_directory):
    if not os.path.exists(target_directory):
        os.makedirs(target_directory)
    for image in incorrect_images:
        shutil.move(image, target_directory)

def evaluate_model(image_directory, test_label, label_list, threshold):
    # Initialize arrays for predictions and incorrectly classified images
    predictions = []
    total_time = 0
    incorrect_images = []
    unknown_images = []

    # Retrieve all images from the image directory
    test_images = [
        os.path.join(image_directory, img)
        for img in os.listdir(image_directory)
        if img.endswith((".png", ".jpg", ".jpeg"))
    ]

    # Set the labels for all test images to be the test_label
    test_labels = [test_label for _ in range(len(test_images))]

    for index, image in enumerate(test_images, start=1):
        # Open and resize the image, then process it with the model's processor
        pil_image = Image.open(image).resize((224, 224))
        inputs = processor(
            text=label_list, images=pil_image, return_tensors="pt", padding=True
        )

        # Move inputs to the selected device
        inputs = {name: tensor.to(device) for name, tensor in inputs.items()}

        # Track the start time
        start_time = time.time()

        # Generate model outputs for the inputs
        outputs = clip_model(**inputs)
        # Convert outputs to probabilities
        probs = F.softmax(outputs.logits_per_image, dim=-1).tolist()[0]
        # Create a dictionary of labels and their corresponding probabilities
        label_probs = dict(zip(label_list, probs))

        # Track the end time
        end_time = time.time()
        # Calculate total processing time
        total_time += end_time - start_time

        # Determine the prediction with the highest probability
        prediction = max(label_probs, key=label_probs.get)
        # Add the prediction to the predictions array
        predictions.append(prediction)

        # Display the image being processed
        print(f"Processed image {index} of {len(test_images)}: {image} : Prediction {prediction}")

        # Check if the prediction matches the test label; if not, check the threshold
                if prediction != test_label:
                        incorrect_images.append(image)
                        continue

        if label_probs[prediction] < threshold:
            unknown_images.append(image)

    # Calculate evaluation metrics
    accuracy = accuracy_score(test_labels, predictions)
    precision = precision_score(test_labels, predictions, average="weighted")
    recall = recall_score(test_labels, predictions, average="weighted")
    f1 = f1_score(test_labels, predictions, average="weighted")
    avg_time_per_image = total_time / len(test_images)
    end_memory = psutil.virtual_memory().used
    memory_used = end_memory - start_memory

    # Display the evaluation metrics and memory usage
    print(f"Model accuracy: {accuracy * 100:.2f}%")
    print(f"Model precision: {precision * 100:.2f}%")
    print(f"Model recall: {recall * 100:.2f}%")
    print(f"Model F1-score: {f1 * 100:.2f}%")
    print(f"Average time per image: {avg_time_per_image:.5f} seconds")
    print(f"Memory used: {memory_used / (1024**2):.2f} MB")

    # Display the confusion matrix
    cm = confusion_matrix(test_labels, predictions, labels=label_list)
    print("Confusion matrix:")
    print(cm)

    # Display the incorrectly classified images
    if not incorrect_images:
        print("No incorrect images found.")
    else:
        print("Incorrectly classified images:")
        print(incorrect_images)

    # Display the unknown images
    if not unknown_images:
        print("No unknown images found.")
    else:
        print("Unknown images:")
        print(unknown_images)

    # Move incorrectly classified images to a separate directory
    target_directory = image_directory + "/Incorrect"
    if incorrect_images:
        move_incorrect_images(incorrect_images, target_directory)

    # Move unknown images to a separate directory
    target_directory = image_directory + "/Unknown"
    if unknown_images:
        move_incorrect_images(unknown_images, target_directory)


image_directory = "./temp/Incorrect"
```

```
test_label = "nsfw"
label_list = ["nsfw", "sfw"]

evaluate_model(image_directory, test_label, label_list, 0.48)
```

4.3 Results

The Image Moderation Testing for openai/clip-vit-base-patch32 was done on a Intel Core i5-1135G7 CPU, GPU acceleration was not used, The images are web scraped from NSFW websites, they contain Safe and Non Safe for work images and were manually reviewed, these are the results from these images:

NSFW (5,422 Images)

Metric	**Result**
Model accuracy	96.62%
Model precision	100.00%
Model recall	96.62%
Model F1-score	98.28%
Average time per image	0.10913 seconds
Memory used	470.63 MB

SFW (444 Images)

Metric	**Result**
Model accuracy	77.25%
Model precision	100.00%
Model recall	77.25%
Model F1-score	87.17%
Average time per image	0.10956 seconds
Memory used	674.94 MB

- SFW images had a lot of edge cases where a human eye can't determine without clear rules if the image is SFW or NSFW, this results in a human error that could impact the study.
- This Testing set is limited in size and a bigger test set is needed but isn't available so web scraping had to be used, both NSFW and SFW are from the same domains so the study can be more concise.
- Using a bigger model like openai/clip-vit-large-patch14 will give better results, but slowness of inference and bigger memory usage are drawbacks.
- Images under 244X244 are more prone to misclassification.
- The choice of labels impact the results so the choice of labels for production must be made with care and after testing.

4.4 Conclusion

Through evaluation, we can measure the accuracy of our image classification and the performance of our application. The results of these processes provide confidence in the functionality and effectiveness of our application, and can guide future improvements and optimizations.

Chapter 5: Future Directions and Enhancements

5.1 Considerations and Future Directions

Utilizing the CLIP Classifier for image moderation offers an immediate, albeit temporary, solution to an enduring issue. This readily available model alleviates the need for retraining initially (It can be fine turned with LORA for example), offering flexibility as the user base and training set expand. Several future additions promise to enhance system efficacy and include:

1. **Batch and Video Processing**: Enabling simultaneous upload and classification of multiple images and extending functionality to incorporate video classification, by evaluating frames extracted from video content.
2. **Model Customization and Augmentation**: Allowing users to upload their training data for personalized, precision results, and including different models and plans for selection.
3. **Interface and Preprocessing Enhancements**: Refining the web interface for seamless user interaction and implementing advanced image preprocessing techniques for optimal input image quality.
4. **Inter-service Integration and Real-time Classification**: Facilitating connectivity with cloud storage services, content management systems, or social media platforms for direct image classification, and incorporating live-stream or immediate classification capabilities.
5. **Feedback, Scalability, and Security Measures**: Establishing a user feedback mechanism for correction of misclassifications, thereby contributing to model refinement. Optimizing the application for improved scalability through load balancing, distributed processing, or model optimization, alongside secure file handling, authentication, access controls, and regular security audits.

5.2 Conclusion

Our FastAPI and CLIP-based image classifier development remains a work in progress. By maintaining a focus on continuous improvement and adapting to the ever-evolving technological landscape, the application promises to remain a formidable tool for image classification tasks. The enhancements proposed in this chapter outline potential for further refining the application's functionality and utility.

References

1. LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition.↩
2. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. Advances in neural information processing systems.↩
3. Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition.↩
4. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions.↩
5. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition.↩
6. Vaswani, A., et al. (2017). Attention is all you need. In Advances in neural information processing systems.↩
7. Chen, T., Isola, P., & Zhu, J. Y. (2021). DeiT: Data-efficient image transformers.↩
8. Dosovitskiy, A., et al. (2020). An image is worth 16x16 words: Transformers for image recognition at scale. ↩↩
9. Radford, A., et al. (2020). Improving language understanding by generative pre-training.↩
10. Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. "Language models are unsupervised multitask learners." OpenAI Blog 11.3 (2021): 8.↩↩↩
11. Dosovitskiy, Alexey, Lucas Beyer, Georg Heigold, Lucas Mnih, Andrew Brock, Aäron van den Oord, et al. "An image transformer." arXiv preprint arXiv:2010.11929 (2020).↩↩
12. Radford, Alec, et al. "Learning transferable visual models from natural language supervision." Proceedings of the European Conference on Computer Vision (ECCV). 2020.↩↩↩
13. Google AI. "Imagen: A Text-to-Image Diffusion Model with High Fidelity and Control." Google AI Blog (2022).↩
14. https://fastapi.tiangolo.com/benchmarks/↩