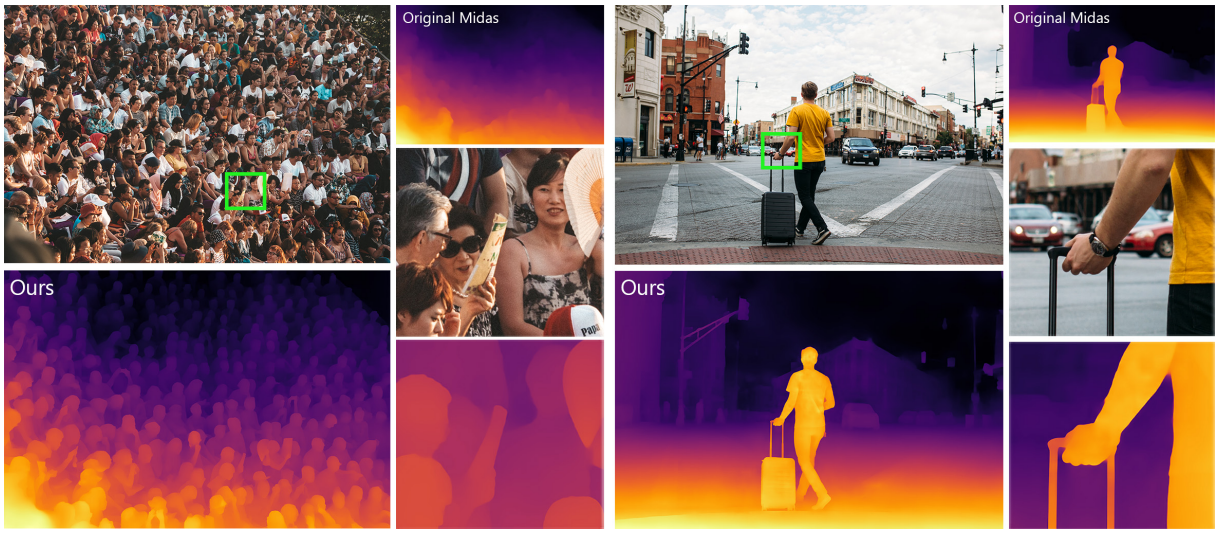


Image Description for Visually Impaired People

Javier Montes Pérez
Emilio Rodrigo Carreira Villalta
Nyasha Eysenck Gandah

Thursday 29th August, 2024

Table of Contents

List of Figures

1. Introduction

This document presents the process of developing a system designed to provide comprehensive and accessible descriptions of images for people with visual impairments. The objective is to build a robust solution that integrates multiple advanced technologies, including YOLO v8 for object detection, MiDaS for generating depth maps, LangChain for structured reasoning, transformers for natural language processing, and a custom model for obtaining weather information from images (based on EfficientNetB3). By combining these technologies, the system is capable of identifying objects, understanding their relationships, generating heat maps, and providing detailed contextual information, such as weather conditions, demonstrating its effectiveness and potential applications in assistive technology and inclusive design.

The project, accessible in this GitHub repository [1], includes various features aimed at achieving this goal. It encompasses the integration of YOLO v8 for object detection, Midas for depth mapping, Langchain for reasoning, transformers for natural language generation, and a model for extracting weather information from images. Additionally, it covers preprocessing and augmentation of training data, scripts for training custom models, and evaluation scripts to assess system performance. Furthermore, the output of the final language model (LLM) is converted into voice using a text-to-speech (TTS) system, making the information accessible through audio, thus enhancing the assistive technology's utility and inclusiveness. To put the icing on the cake, we have also developed a mini-app in Flask to put the whole system into production.

This document details the steps involved in setting up and running the project, explaining the preprocessing techniques used for the training data, the architecture of the integrated models, and the training and evaluation procedures. Each aspect of the implementation is thoroughly described, ensuring that readers can understand both the practical implementation and the underlying concepts.

This work is part of a broader effort to develop advanced assistive technologies and is inspired by the documentation and resources provided by leading experts in the field. The goal is to create a comprehensive guide that not only facilitates the understanding of the techniques used but also encourages experimentation and learning in the fields of computer vision, natural language processing, and inclusive design.

[1] https://github.com/rorro6787/img-desc-visually-impaired

2. Ultralytics, YOLO

Contents

YOLO, which stands for "You Only Look Once," is a popular real-time object detection system designed by ultralytics. It is widely used in computer vision tasks because of its high speed and accuracy in detecting objects within images and videos. Here's a detailed overview of YOLO:

- **Single Neural Network Approach**: Unlike traditional object detection systems that use a pipeline of separate models for region proposal and classification, YOLO uses a single convolutional neural network (CNN) to predict multiple bounding boxes and class probabilities directly from the full images in one evaluation.
- **Real-time Detection**: YOLO is designed to process images extremely quickly, making it suitable for real-time applications. For instance, YOLO can achieve frame rates of up to 45 frames per second (fps) on a modest GPU.

YOLOv8 represents the latest iteration in the YOLO (You Only Look Once) series of object detection models. As a continuation of the YOLO family, YOLOv8 builds upon the advancements made by its predecessors, enhancing both speed and accuracy. Here's a detailed look at what makes YOLOv8 stand out, particularly the yolov8x.pt model.

2.1 Why yolov8x.pt?

The model yolov8x.pt is one of the variants in the YOLOv8 family, and the name conveys specific information about its characteristics:

- **yolov8**: This prefix denotes that the model is part of the YOLO version 8 series. It indicates the architecture and the improvements over earlier YOLO versions.
- **x**: This letter stands for "extra-large" and signifies that the model is designed with the largest capacity in the YOLOv8 family. The "x" variant includes significantly more parameters and layers compared to its smaller counterparts like yolov8n (nano) or yolov8s (small). This results in higher detection accuracy and robustness, though it requires more computational resources and memory.
- **.pt**: This suffix indicates that the model is saved in PyTorch format. PyTorch is a popular deep learning framework that provides flexibility and ease of use for training and deploying models.

Even though yolov8x is an ”extra-large” model, it is designed to maximize performance and accuracy. In fact, yolov8x.pt is an excellent choice when high detection accuracy and robustness are crucial, and there are sufficient computational resources available. Its advanced architecture and increased parameter count make it particularly well-suited for complex object detection tasks. For training a model to detect hand gestures in a Rock-Paper-Scissors application, yolov8x.pt offers a powerful solution by delivering the highest possible accuracy and precision, ensuring reliable recognition of hand gestures even in challenging scenarios. However, it is important to note that this model is more demanding on hardware resources, making it best suited for environments where computational power is not a limiting factor.

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

Figure 2.1: YOLO models features

2.2 Code of the Detection

Our computer vision system processes a given image using three computer vision models. The first of these is the ultralytics yolov8x.pt model, which is responsible for analyzing the image and returning the coordinates of all the bounding boxes that the pre-trained model can detect. This is why we opted for an extra-large pre-trained model, as training a new model would be extremely costly. Instead, we are simply evaluating it. Afterward, we will convert the information generated by the model into a well-structured string, which will later be fed to the LLM along with the data from the other models:

```
def extract_entities_image(sourcePath:str):
    model = YOLO("yolov8x.pt", task="detect")
    cd = os.getcwd()
    information = []

    original_path = os.path.dirname(sourcePath)
    results = model(sourcePath, save = True,
       ↪ project=original_path)
    ...
    ...
    for result in results:
```

```
        # This contains the bounding boxes
        boxes = result.boxes
        for box in boxes:
            # Extract the bounding box coordinates
            x1, y1, x2, y2 = box.xyxy[0].tolist()  # top-left
               ↪ and bottom-right corners

            confidence = box.conf[0].item()  # confidence
               ↪ score
            class_name = model.names[int(class_id)]  # class
               ↪ name

            center_x = int((x1 + x2) / 2)
            center_y = int((y1 + y2) / 2)

            # Print or store the results
            information.append(f"{class_name} at coordinates:
               ↪ [{x1}, {y1}, {x2}, {y2}] with heat deepth of
               ↪ {heat_map_array[center_y][center_x]} in the
               ↪ centre of the image\n")

    return dimensions, max_deepth, min_deepth, weather,
       ↪ information
```

Note that this function returns five elements: the first is the post-processed information from the YOLO model, and the remaining elements are the post-processed data from the other models, which will be discussed further later. We use the YOLO script as the primary source of information for the LLM script, which will rely on it to obtain all necessary details. Within this function, we can retrieve the additional data by simply calling the different scripts as follows:

```
import computer_vision.heat_map as hm
import computer_vision.inference_weather as wm

heat_map_path = hm.heat_map(sourcePath)
heat_map_array = hm.load_npy(heat_map_path)
max_deepth = "Max deepth: " + str(heat_map_array.max()) + "\n"
min_deepth = "Min deepth: " + str(heat_map_array.min()) + "\n"
weather = wm.inference_image(sourcePath)
dimensions = ...
```

2.2.1 Color Detection using K-Means Clustering

In the context of this project, color detection within specific regions of an image is a critical task for generating accurate and descriptive information. For this purpose, K-means clustering has been employed to analyze the colors of subregions corresponding to bounding boxes detected by the YOLO (You Only Look Once) object detection model. This subsection provides an in-depth explanation of how K-means clustering is utilized to determine the predominant color in these subregions.

2.2.1.1 Overview of K-Means Clustering

K-means clustering is a widely used unsupervised learning algorithm that partitions data into k distinct clusters based on feature similarity. Each cluster is represented by its centroid, which is the mean of all data points assigned to that cluster. The algorithm iterates between assigning data points to the nearest centroid and updating centroids based on the mean of the assigned points until convergence is achieved.

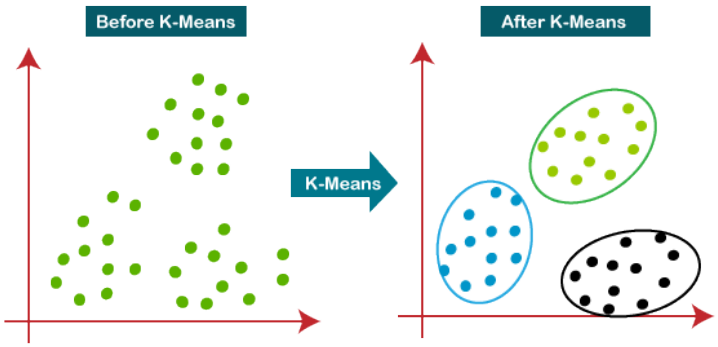


Figure 2.2: K-means procedure

2.2.1.2 Application to Color Detection

In our system, K-means clustering is applied to the RGB color values extracted from image subregions. Here is a step-by-step explanation of the process:

1. Extraction of Color Data After the YOLO model detects bounding boxes within an image, we extract the pixel data from the subregion corresponding to each bounding box. This subregion's pixels are represented in RGB color space. The data is reshaped into a list of RGB tuples, where each tuple represents a single pixel's color.

2. Preprocessing The RGB color values are preprocessed to ensure they are in a suitable format for clustering. This involves normalizing the pixel values if necessary and ensuring that all data points are properly formatted as RGB tuples.

3. Applying K-Means Clustering K-means clustering is performed on the RGB color values. The algorithm is initialized with a predefined number of clusters, k. The choice of k influences the clustering results and is typically determined through experimentation or based on specific requirements. The algorithm then iterates to partition the color data into k clusters and computes the centroid of each cluster.

4. Identifying the Dominant Color After clustering, each cluster represents a group of similar colors. To determine the predominant color in the subregion, we identify the cluster with the highest number of data points, i.e., the most common color cluster. The centroid of this dominant cluster is taken as the representative color of the subregion. This centroid is a vector in RGB space and is converted into an integer tuple representing the color.

5. Handling Multiple Shades In cases where the detected color might have variations (e.g., different shades of red), K-means clustering helps in grouping these variations into a common cluster. By analyzing the size of each cluster and selecting the largest one, the system can effectively identify the primary color even when multiple shades are present.

2.2.1.3 Advantages of Using K-Means Clustering

The use of K-means clustering for color detection offers several advantages:

- **Scalability**: K-means can handle large datasets efficiently, making it suitable for images with many pixels.
- **Simplicity**: The algorithm is relatively simple to implement and understand.
- **Flexibility**: By adjusting the number of clusters, K-means can adapt to different levels of color granularity.

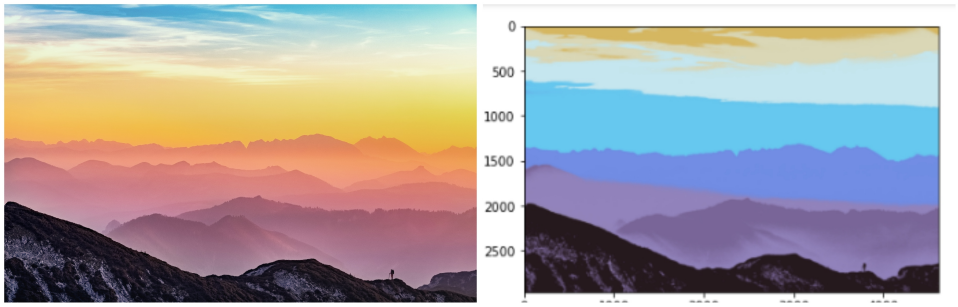


Figure 2.3: K-means procedure for image segmentation

2.2.1.4 Conclusion

K-means clustering provides an effective method for determining the dominant color in image subregions corresponding to bounding boxes. By leveraging this technique, the system can deliver accurate and meaningful color descriptions, which are integral to generating comprehensive image descriptions for visually impaired users.

3. MiDaS

Contents

Monocular Depth Estimation is the task of predicting depth from a single image, which is a challenging problem due to the inherent ambiguity and lack of explicit depth cues in monocular images. MiDaS (Mixed Depth Estimation) is a state-of-the-art model for monocular depth estimation that has shown impressive results on various datasets. This documentation provides a detailed overview of MiDaS, including its architecture, implementation, and usage in our project.

3.1 Background and Motivation

MiDaS computes relative inverse depth from a single image. The repository provides multiple models that cover different use cases ranging from a small, high-speed model to a very large model that provide the highest accuracy. The models have been trained on 10 distinct datasets using multi-objective optimization to ensure high quality on a wide range of inputs.

3.1.1 Monocular Depth Estimation: An Overview

Depth estimation from images is a crucial task in computer vision with applications in autonomous driving, robotics, augmented reality, and more. The goal of depth estimation is to predict the distance from the camera to various points in the scene, which is essential for understanding the 3D structure of the environment.

Traditionally, depth information is obtained using specialized hardware such as Li-

DAR, stereo cameras, or structured light systems. However, these methods can be expensive, complex, and sensitive to environmental conditions like lighting and weather. Monocular depth estimation, on the other hand, aims to infer depth from a single image, offering a more flexible and cost-effective solution.

Monocular depth estimation is inherently challenging due to the ambiguous nature of interpreting depth from a single 2D projection. Factors like texture, shading, occlusions, and object boundaries contribute to the complexity of accurately inferring depth. These challenges necessitate sophisticated models capable of understanding and interpreting the various cues present in an image.

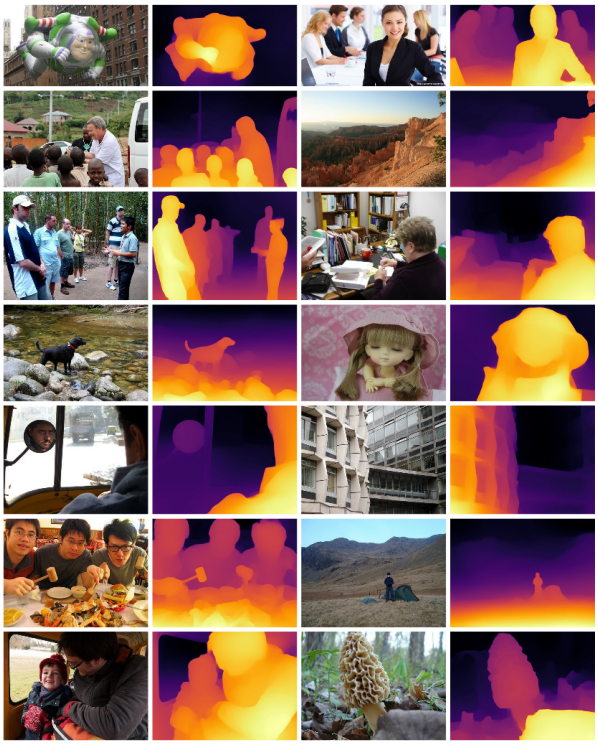


Figure 3.1: MiDaS usage

3.1.2 Challenges in Monocular Depth Estimation

Several key challenges arise in the context of monocular depth estimation:

- **Ambiguity in Depth Cues:** A single image does not inherently contain depth information, making it difficult to discern the relative distances of objects. Without additional information or assumptions, the same 2D image can correspond to multiple 3D scenes.
- **Variety in Scene Structures:** Real-world scenes exhibit a vast variety of structures, ranging from indoor environments to complex outdoor landscapes. A depth estimation model must generalize well across different scene types and contexts.
- **Occlusions:** Objects in a scene often partially obscure each other, creating occluded regions where depth information is less clear. Proper handling of occlusions is crucial for accurate depth estimation.

- **Scale Variation:** Objects of the same type can appear at different scales in an image depending on their distance from the camera. Accurately estimating depth requires the model to recognize and compensate for such scale variations.
- **Lack of Ground Truth Data:** Acquiring ground truth depth data for training is challenging, particularly for large-scale, diverse datasets. Many datasets rely on synthetic or approximate depth information, which may not fully capture real-world depth variations.

3.1.3 Introduction of MiDaS

MiDaS was introduced in the paper titled *"Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-Shot Cross-Dataset Transfer"* by Rene Ranftl*, Katrin Lasinger*, David Hafner, Konrad Schindler, and Vladlen Koltun *et al.*, presented at the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) in 2020. The paper proposed a novel approach to monocular depth estimation by leveraging a mixture of datasets to enhance the model's robustness and generalizability. The main idea was to train a single model on multiple datasets with varying characteristics, thereby enabling the model to perform well across different types of scenes and environmental conditions.

Dataset	Indoor	Outdoor	Dynamic	Video	Metric	Dense	Accuracy	Diversity	Annotation	# Images
NYUDv2 [40]	✓		(✓)	✓	✓	✓	Medium	Low	RGB-D	407K
SUN-RGBD [41]	✓			✓	✓	✓	Medium	Low	RGB-D	10K
ScanNet [7]	✓			✓	✓	✓	Medium	Low	RGB-D	2.5M
Make3D [37]		✓			✓	✓	Low	Low	Laser	534
KITTI LiDAR [14, 32]		✓	✓	✓	✓		Medium	Low	Laser	93K
KITTI Stereo [14, 32]		✓	✓	✓	✓	✓	Medium	Low	Stereo	93K
Cityscapes [6]		✓	✓	✓	✓	✓	Medium	Low	Stereo	25K
DIW [3]	✓	✓	✓				Low	High	User clicks	496K
ETH3D [38]	✓	✓			✓	✓	High	Low	Laser	454
Tanks and Temples [24]	✓	✓			✓	✓	High	Low	Laser	3290
Sintel [1]	✓	✓	✓	✓	(✓)	✓	High	Medium	Synthetic	1064
MegaDepth [28]		✓	(✓)			✓	Medium	Medium	SfM	130K
ReDWeb [45]	✓	✓	✓			✓	Medium	High	Stereo	3600
3D Movies (Ours)	✓	✓	✓	✓		✓	Medium	High	Stereo	∼ 50K/movie

Table 1. Datasets that provide relevant supervision for monocular depth estimation. No single real-world dataset features a large numbe of diverse scenes with dense and accurate ground truth.

Figure 3.2: MiDaS metrics on Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-shot Cross-dataset Transfer research

The authors of the paper highlighted that most previous approaches to monocular depth estimation suffered from limited generalization capabilities, often performing well only on specific datasets they were trained on. This limitation hindered their practical applications, as real-world scenarios often differ significantly from the conditions represented in training datasets.

3.1.4 Motivation for MiDaS

Given the challenges outlined above, there is a clear need for robust and reliable monocular depth estimation models that can generalize across a wide range of scenarios. MiDaS (Mixed Depth Estimation Models) was developed to address these challenges and provide a state-of-the-art solution for depth estimation.

- **Model Generalization:** MiDaS leverages a diverse set of training data and sophisticated network architectures to ensure that it generalizes well across various environments, from indoor scenes to complex outdoor landscapes. This ability to generalize is critical for applications requiring robust performance under varying conditions.
- **Use of Vision Transformers (ViTs):** By incorporating Vision Transformers, MiDaS benefits from the ability to capture long-range dependencies in the image, improving its capacity to understand complex scene structures and relationships between distant objects. This approach enhances depth estimation accuracy, particularly in scenes with intricate details.
- **Efficiency and Scalability:** MiDaS is designed to be computationally efficient, making it suitable for real-time applications and deployment on devices with limited computational resources. Its scalability allows for the handling of high-resolution images, which is important for tasks requiring fine-grained depth information.
- **Integration Capabilities:** The flexibility of MiDaS allows it to be easily integrated into various systems and workflows. This adaptability makes it an attractive option for a wide range of applications, from autonomous vehicles to augmented reality, where depth perception is critical.

3.1.5 Impact on Applications

The ability to accurately estimate depth from a single image opens up numerous possibilities across different fields:

- **Autonomous Vehicles:** Accurate depth estimation is vital for understanding the surrounding environment, detecting obstacles, and making navigation decisions. MiDaS can enhance the perception capabilities of autonomous vehicles, contributing to safer and more efficient operation.
- **Robotics:** Robots equipped with monocular depth estimation can better understand their environment, navigate complex spaces, and interact with objects. MiDaS enables robots to perceive depth accurately without relying on bulky and expensive sensors.
- **Augmented Reality (AR):** Depth estimation is crucial for overlaying virtual objects onto real-world scenes in a realistic manner. MiDaS allows for more immersive and interactive AR experiences by providing accurate depth information from a single camera feed.
- **Human-Computer Interaction (HCI):** Understanding depth can improve user interactions with devices, enabling more intuitive and natural interfaces. MiDaS can contribute to advancements in gesture recognition, virtual meetings, and other HCI applications.

The development of MiDaS represents a significant step forward in monocular depth estimation, addressing many of the limitations of previous methods. Its ability to provide accurate and reliable depth information from a single image has broad implications for the future of computer vision and its applications.

3.2 Overview of MiDaS

MiDaS is a model developed by Intel Labs that utilizes deep learning techniques to estimate depth from a single image. It was trained on a large and diverse set of datasets, which enables it to generalize well to different types of scenes and environments. Its key features are:

- **State-of-the-art Accuracy:** MiDaS achieves high accuracy across multiple datasets.
- **Generalization:** Trained on a mixture of datasets, MiDaS generalizes well to various environments and object types.
- **Efficiency:** MiDaS provides a good balance between computational efficiency and accuracy, making it suitable for real-time applications.

3.3 Architecture of MiDaS

MiDaS utilizes a deep convolutional neural network (CNN) architecture based on the Vision Transformer (ViT) backbone. The architecture is designed to effectively capture both local and global contextual information from the input image.

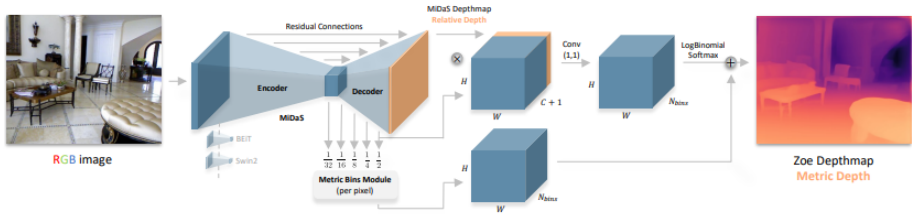


Figure 3.3: MiDaS architecture

3.3.1 Backbone Network

The backbone of MiDaS is based on the Vision Transformer (ViT) architecture, which leverages self-attention mechanisms to capture long-range dependencies. The use of ViT enables MiDaS to process images at multiple scales, improving its ability to estimate depth for objects of varying sizes.

3.3.2 Depth Prediction Head

MiDaS includes a depth prediction head that processes the output of the backbone network to produce a dense depth map. The depth prediction head is designed to output a relative depth map, which is then normalized to provide a consistent depth range.

4. Weather recognition: a comprehensive overview

Contents

This document provides a detailed overview of the weather recognition component of our image analysis project. We utilize the MiDaS algorithm for depth estimation, a deep learning model for weather classification, and various preprocessing techniques to achieve accurate weather recognition in images. The integration of these components is discussed, along with the methodology, experimental setup, and results.

4.1 Introduction

Weather recognition in images involves determining the current weather conditions such as sunny, cloudy, rainy, or snowy from visual data. Accurate weather classification is valuable for applications in autonomous systems, environmental monitoring, and various other fields where environmental context is crucial.

Figure 4.1: Examples of usage in weather recognition

4.2 Weather Classification Model

This section outlines the key components involved in developing and evaluating the weather classification model. It starts with the preparation of the dataset, detailing the steps necessary to ensure that the data is suitable for training a robust model. Following that, the model architecture is described, focusing on the use of the EfficientNetB3 network and its various components. This setup aims to effectively classify images into different weather conditions by leveraging advanced deep learning techniques.

4.2.1 Data Preparation

For weather classification, a diverse dataset of images labeled with different weather conditions is used. The dataset preparation involves several steps:

- **Dataset Collection:** Images are collected from various sources to ensure a wide range of weather conditions and scenarios.
- **Data Splitting:** The dataset is divided into training, validation, and test sets to evaluate the model's performance effectively. Typically, 80% of the data is used for training, 10% for validation, and 10% for testing.
- **Image Preprocessing:** Images are resized to a consistent resolution, normalized, and augmented to enhance model generalization. Techniques such as rotation, flipping, and scaling are applied to increase the robustness of the model.

4.2.2 Model Architecture

The weather classification model is built using a deep learning approach, specifically leveraging the EfficientNetB3 architecture. EfficientNetB3 is known for its efficiency and effectiveness in image classification tasks. The model architecture includes:

- **Base Model:** EfficientNetB3, a pre-trained convolutional neural network (CNN) that extracts high-level features from images.
- **Batch Normalization:** Applied to normalize the activations and gradients, improving training stability and convergence.

- **Dense Layers:** Fully connected layers for classifying the image into predefined weather categories.
- **Dropout Layer:** Used to prevent overfitting by randomly dropping units during training.

4.2.2.1 EfficientNetB3 Architecture

EfficientNetB3 is part of the EfficientNet family, which is renowned for its balance between accuracy and efficiency. EfficientNetB3 is designed to provide high accuracy with relatively lower computational cost compared to traditional architectures.

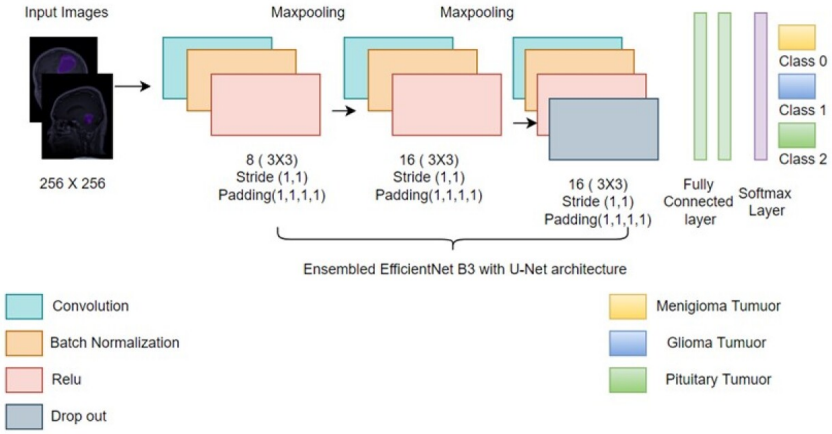


Figure 4.2: EfficientNetB3 network

Here is a detailed breakdown of its internal architecture, including an in-depth examination of the network's various layers, components, and the strategies it employs to achieve its balance of high accuracy and computational efficiency. This analysis will cover the specific design choices, such as the depthwise separable convolutions and compound scaling, that contribute to the model's performance and efficiency.

1. Compound Scaling EfficientNetB3 employs a compound scaling method to optimize the model's performance. This method scales the depth, width, and resolution of the network uniformly based on a set of scaling coefficients. The compound scaling formula used in EfficientNet is:

$$\text{Depth} = \alpha^d \times \text{Baseline Depth}$$

$$\text{Width} = \beta^w \times \text{Baseline Width}$$

$$\text{Resolution} = \gamma^r \times \text{Baseline Resolution}$$

where α, β, and γ are scaling coefficients for depth, width, and resolution, respectively. EfficientNetB3 uses specific values for these coefficients to achieve a good balance of accuracy and efficiency.

2. Model Components EfficientNetB3 is composed of several key components:

- **Stem Block:** The model starts with a stem block consisting of a convolutional layer with a 3x3 kernel, followed by batch normalization and a ReLU activation function. This block initializes the feature extraction process.
- **Mobile Inverted Bottleneck Convolutions (MBConv):** EfficientNetB3 employs MBConv blocks, which are efficient building blocks designed to optimize performance and reduce computational cost. Each MBConv block consists of:
  - **Depthwise Separable Convolution:** This convolution reduces computational complexity by separating the spatial and depthwise convolutions.
  - **Expansion Layer:** A layer that expands the number of channels using a 1x1 convolution, followed by a depthwise separable convolution.
  - **Squeeze-and-Excitation (SE) Block:** A mechanism that adaptively recalibrates channel-wise feature responses by applying a global average pooling followed by two fully connected layers.
  - **Residual Connection:** A shortcut that bypasses the block to aid in gradient flow during training.
- **Top Layer:** After processing through multiple MBConv blocks, EfficientNetB3 concludes with a global average pooling layer that reduces the spatial dimensions of the feature maps, followed by a dense layer with a softmax activation function to output class probabilities.

Block N0. (i)	Layer (F_i ( ))	Resolution (H_i x W_i)	No. of Layers (L_i)
1	Conv 3x3	300x300	1
2	MBConv1, 3x3	150x150	2
3	MBConv6, 3x3	150x150	3
4	MBConv6, 5x5	75x75	3
5	MBConv6, 3x3	38x38	5
6	MBConv6, 5x5	19x19	5
7	MBConv6, 5x5	10x10	6
8	MBConv6, 3x3	10x10	2
9	Conv 1x1	10x10	1
10	Global Pooling	10x10	1
11	Dense layer	10x10	1

Figure 4.3: EfficientNetB3 model layers

3. Efficiency and Performance EfficientNetB3 achieves high accuracy with fewer parameters and lower computational cost by effectively balancing the model's width, depth, and resolution. The architecture is optimized using neural architecture search (NAS) techniques to ensure that the trade-offs between accuracy and efficiency are well-managed.

4. Training Strategy EfficientNetB3 is typically trained using the ImageNet dataset with transfer learning. The pre-trained weights are used to initialize the model, which can then be fine-tuned on specific tasks, such as weather classification, to achieve high performance on the target dataset.

4.2.3 Training Procedure

The training process involves:

- **Loss Function:** Categorical cross-entropy is used as the loss function, which is suitable for multi-class classification tasks.
- **Optimizer:** The Adamax optimizer is chosen for its adaptive learning rate capabilities, which helps in achieving better convergence.
- **Metrics:** The model's performance is evaluated using accuracy, precision, recall, and AUC (Area Under the Curve) metrics.
- **Epochs and Batch Size:** The model is trained for a predefined number of epochs with a batch size that balances training efficiency and computational resources.

4.2.4 Evaluation Metrics

The performance of the weather classification model is assessed using the following metrics:

- **Accuracy:** Measures the percentage of correctly classified images out of the total number of images.
- **Precision:** Indicates the proportion of true positive predictions among all positive predictions made by the model.
- **Recall:** Measures the proportion of true positive predictions among all actual positive instances in the dataset.
- **AUC (Area Under the Curve):** Represents the ability of the model to distinguish between classes, with higher values indicating better performance.

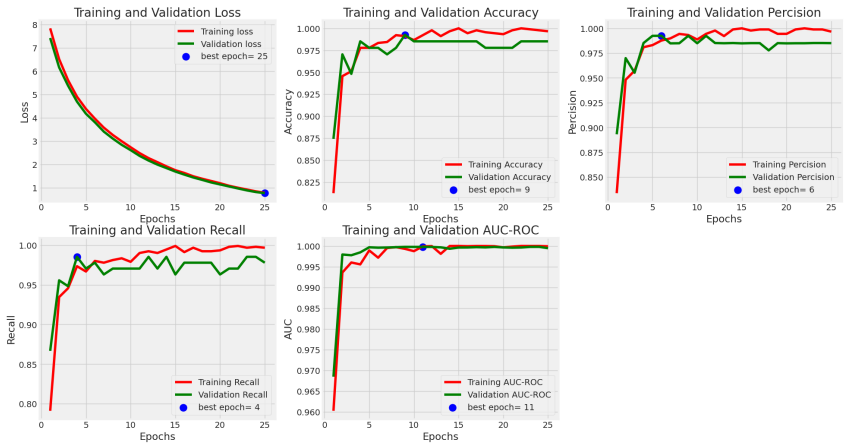


Figure 4.4: Weather recognition model metrics after training

4.3 Integration and Results

This section details the integration of the weather recognition model into the image analysis system and presents an overview of the results obtained from its application. We begin by describing how the model is integrated into the overall system architecture, followed by an evaluation of its performance. The discussion will highlight key findings, compare them to existing methods, and address any challenges encountered during the evaluation process. Finally, we will summarize the model's contributions and propose directions for future research and improvements.

4.3.1 System Integration

The weather recognition model is integrated into the overall image analysis system, where it processes images to predict the current weather conditions. The integration involves feeding images into the model, obtaining predictions, and utilizing these predictions for further analysis or decision-making.

4.3.2 Results and Discussion

The model's performance is evaluated on a test set, and results are compared with baseline methods or existing models. The evaluation metrics provide insights into the model's effectiveness in classifying different weather conditions. Any observed challenges, such as misclassifications or limitations, are discussed along with potential solutions and improvements.

4.4 Conclusion and Future Work

The weather recognition model demonstrates the ability to accurately classify weather conditions from images, contributing to applications requiring environmental context. Future work may involve exploring advanced architectures, incorporating temporal data from video sequences, or expanding the dataset to improve model robustness and accuracy.

5. Langchain and LLMs

Contents

This document provides an in-depth overview of LangChain, a powerful library designed for building applications with large language models (LLMs) like GPT-4. It covers the library's features, architecture, key components, and practical usage examples to demonstrate how LangChain can be employed in various applications.

5.1 Introduction

LangChain is an innovative library aimed at simplifying the development of applications utilizing large language models (LLMs). By offering a high-level interface for integrating LLMs into various applications, LangChain facilitates the creation of sophisticated and interactive systems, leveraging the capabilities of advanced models

like GPT-4. This document provides a comprehensive guide to LangChain, exploring its features, architecture, and practical applications.

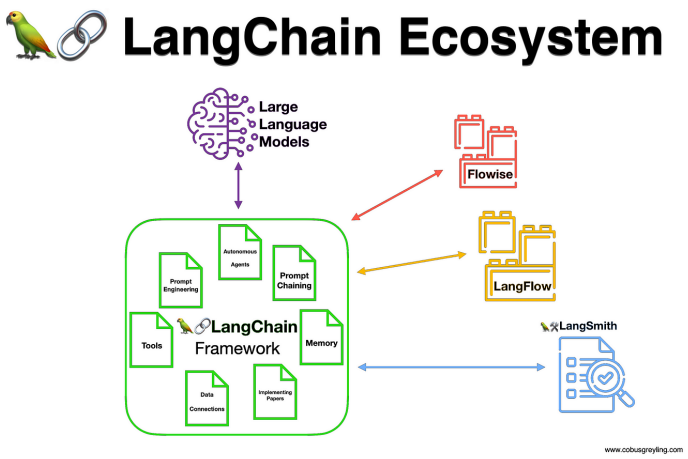


Figure 5.1: LangChain ecosystem architecture

5.2 Key Features of LangChain

LangChain offers a range of features designed to streamline the development of applications using LLMs:

- **Seamless Integration:** LangChain simplifies the integration of LLMs into applications by providing pre-built modules and interfaces.
- **Modular Design:** The library is modular, allowing developers to use only the components they need, such as language models, prompt templates, or output processors.
- **Customizable Pipelines:** LangChain supports the creation of customizable processing pipelines that can be tailored to specific use cases.
- **Pre-trained Models:** The library includes access to a variety of pre-trained LLMs, including GPT-4, facilitating rapid development and experimentation.
- **Advanced Features:** LangChain supports advanced features such as memory management, context-aware responses, and chaining of multiple language models.

5.3 Architecture of LangChain

LangChain is designed with a modular architecture that allows for flexibility and extensibility. This section provides a detailed explanation of the internal architecture of LangChain, including its core components and how they interact to deliver functionality.

5.3.1 Core Components

LangChain is composed of several core components that work together to provide a robust framework for building applications with LLMs:

- **Language Models:** The backbone of LangChain, providing the ability to generate and comprehend text. It supports various models such as GPT-4.

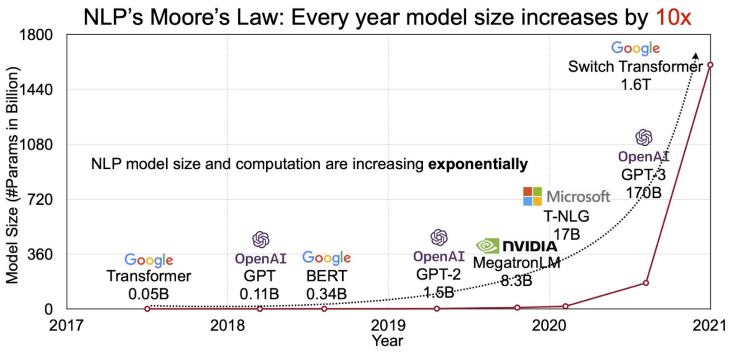


Figure 5.2: LLMs sizes increasing

- **Prompt Templates:** Used to standardize the input prompts sent to the language models. These templates help ensure consistency and quality in the generated responses.
- **Output Processors:** Components that handle the post-processing of outputs from language models, including formatting and additional transformations.
- **Memory Management:** Mechanisms for storing and utilizing contextual information across interactions to maintain continuity in conversations.
- **Pipeline Management:** Tools for creating and managing sequences of processing steps, including interactions with different language models and processing units.

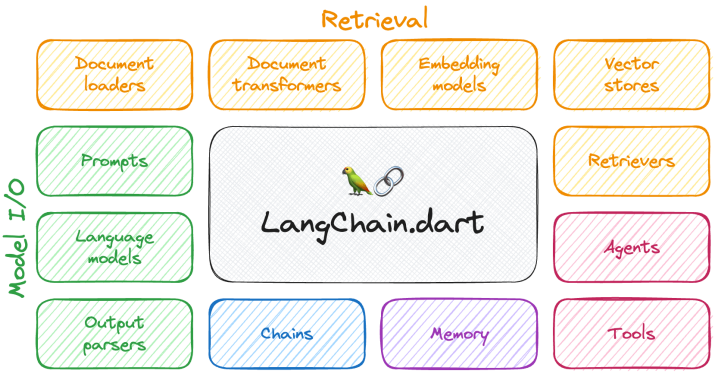


Figure 5.3: LangChain ecosystem components

5.3.2 Detailed Architecture

Here is a brief overview of the comprehensive architecture offered by the LangChain framework:

5.3.2.1 Language Models

The language models in LangChain are modular components responsible for generating text based on the provided prompts. These models include:

- **Model Initialization:** Models are initialized with specific configurations, including pre-trained weights, model size, and other parameters.
- **Model Invocation:** When a prompt is provided, the model processes it to generate a response. This involves tokenizing the input, passing it through multiple layers, and decoding the output.
- **Model Fine-Tuning:** Some applications may involve fine-tuning the pre-trained models on specific datasets to improve performance for particular tasks.

5.3.2.2 Prompt Templates

Prompt templates are designed to structure inputs consistently:

- **Template Definition:** Templates define the format of the input prompt, including placeholders for dynamic content.
- **Template Filling:** When a prompt is generated, the template is filled with specific values to create a final prompt for the language model.
- **Validation and Testing:** Templates are validated to ensure they produce effective and coherent prompts for the intended use cases.

5.3.2.3 Output Processors

Output processors manage the results from the language models:

- **Output Formatting:** Results are formatted according to application needs, including text formatting and structuring.
- **Post-Processing:** Additional transformations may be applied to the outputs, such as extracting key information or reformatting for presentation.
- **Error Handling:** Processors manage potential errors or inconsistencies in the outputs to ensure reliable results.

5.3.2.4 Memory Management

Memory management involves maintaining and utilizing contextual information:

- **Contextual Memory:** Stores information from previous interactions to provide context for current responses. This is crucial for applications like chatbots.
- **Persistent Memory:** Retains information across sessions, allowing the system to remember user preferences or important details.
- **Memory Retrieval:** Mechanisms for retrieving and utilizing stored memory to enhance interactions and responses.

5.3.2.5 Pipeline Management

Pipeline management facilitates complex processing tasks:

- **Pipeline Definition:** Pipelines consist of multiple processing steps, including interactions with different models and processors.
- **Pipeline Execution:** Executes the defined pipeline, ensuring that each step is performed in the correct sequence.
- **Pipeline Optimization:** Optimizes the performance of pipelines by managing resource utilization and processing efficiency.

5.4 Usage and Examples

LangChain provides a user-friendly interface for building applications with LLMs. Below are examples demonstrating how to use the library for different tasks.

5.4.1 Setting Up LangChain

To start using LangChain, you need to install the library and configure the necessary components:

```
# Install LangChain library
!pip install langchain

# Import necessary modules
import langchain
from langchain import LLM, PromptTemplate, OutputProcessor

# Initialize the language model
model = LLM('gpt-4')

# Define a prompt template
template = PromptTemplate('Translate␣the␣following␣English␣
   ↪ text␣to␣French:␣{text}')

# Initialize an output processor
processor = OutputProcessor()
```

5.4.2 Example 1: Simple Text Generation

In this example, we demonstrate how to generate text using a pre-trained language model:

```
# Define a prompt
prompt = "What␣is␣the␣capital␣of␣France?"

# Generate a response using the language model
response = model.generate(prompt)
```

```
print("Response:", response)
```

5.4.3 Example 2: Using Prompt Templates

This example shows how to use prompt templates to structure input for the language model:

Listing 5.1: Prompt Template Example

```
# Define a prompt template
template = PromptTemplate("Write a short story about {topic}")

# Fill the template with specific values
filled_prompt = template.fill(topic="a brave knight")

# Generate text based on the filled prompt
story = model.generate(filled_prompt)

print("Story:", story)
```

5.4.4 Example 3: Chaining Models

LangChain allows for chaining multiple models to achieve complex processing tasks:

```
# Define multiple language models
model1 = LLM('gpt-4')
model2 = LLM('gpt-3.5')

# Define a prompt
prompt = "Summarize the following text: {text}"

# Generate a summary using the first model
summary = model1.generate(prompt.format(text="LangChain is a
    ↪ library for LLMs..."))

# Refine the summary using the second model
refined_summary = model2.generate("Refine the following
    ↪ summary: " + summary)

print("Refined Summary:", refined_summary)
```

5.5 Using GPT-3.5 with LangChain

In our project, we will specifically use GPT-3.5, a model developed by OpenAI, integrated within LangChain. This section provides a detailed explanation of GPT-3.5 and its role in the context of LangChain.

5.5.1 Overview of GPT-3.5

GPT-3.5, or Generative Pre-trained Transformer 3.5, is a state-of-the-art language model developed by OpenAI. It builds on the advancements of its predecessors, particularly GPT-3, with improvements in performance, coherence, and understanding.

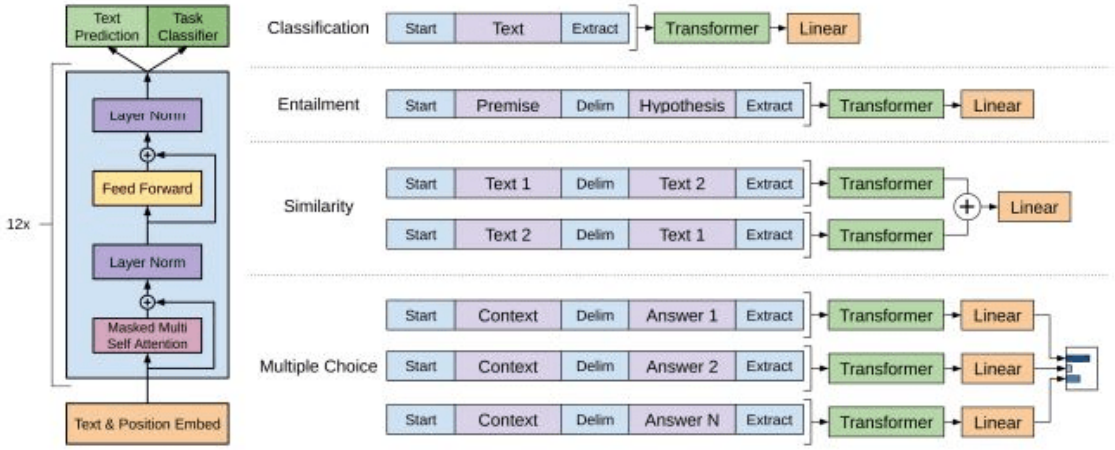
- **Architecture:** GPT-3.5 utilizes the Transformer architecture, which is based on self-attention mechanisms. This architecture allows the model to process and generate text with a deep understanding of context and semantics.
- **Training Data:** The model was trained on diverse datasets comprising books, articles, and other text sources to develop a broad understanding of human language.
- **Capabilities:** GPT-3.5 excels at various natural language processing tasks, including text generation, question answering, translation, and summarization. Its improved fine-tuning capabilities allow for more accurate and context-aware responses.

5.5.2 Internal Architecture of GPT-3.5

5.5.2.1 Transformer Architecture

GPT-3.5 leverages the Transformer architecture, which consists of several key components:

- **Self-Attention Mechanism:** This mechanism allows the model to weigh the importance of different words in a sequence, enabling it to understand and generate contextually relevant text.
- **Positional Encoding:** Since the Transformer architecture does not inherently understand word order, positional encodings are added to the input embeddings to provide information about the position of each word in the sequence.
- **Multi-Head Attention:** Multiple attention heads are used to capture different aspects of the relationships between words. This allows the model to process information from various perspectives simultaneously.
- **Feed-Forward Networks:** Each layer of the Transformer includes a feed-forward network that applies non-linear transformations to the output of the attention mechanism, enhancing the model's capacity to learn complex patterns.



(left) Transformer architecture and training objectives used in this work. **(right)** Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

Figure 5.4: GPT3.5 architecture

5.5.2.2 Model Training and Fine-Tuning

GPT-3.5 underwent extensive pre-training on a large corpus of text data. During this phase:

- **Unsupervised Learning:** The model learned to predict the next word in a sequence given the preceding context, allowing it to develop a deep understanding of language patterns.
- **Fine-Tuning:** Post pre-training, GPT-3.5 can be fine-tuned on specific datasets to adapt it for particular tasks or domains, enhancing its performance in targeted applications.

Model	Parameters	Tokens to Train to Chinchilla Point (B)	Cerebras Model Studio CS-2 Day to Train	Cerebras Model Studio Price to Train
GPT3-XL	1.3	26	0.4	$2,500
GPT-J	6	120	8	$45,000
GPT-3 6.7B	6.7	134	11	$40,000
T-5 11B	11	34*	9	$60,000
GPT-3 13B	13	260	39	$150,000
GPT NeoX	20	400	47	$525,000
GPT 70B	70	1,400	85	$2,500,000
GPT 175B	175	3,500	Contact For Quote	Contact For Quote

* - T5 tokens to train from the original T5 paper. Chinchilla scaling laws not applicable.

Figure 5.5: Some LLMs training procedures cost

5.5.2.3 Capabilities and Applications

GPT-3.5's capabilities make it suitable for a wide range of applications, including:

- **Text Generation:** Produces coherent and contextually relevant text based on the provided prompts.
- **Question Answering:** Provides accurate answers to questions by understanding and analyzing the context.
- **Translation and Summarization:** Translates text between languages and summarizes long documents effectively.
- **Interactive Conversations:** Powers chatbots and virtual assistants with context-aware responses and natural interactions.

5.6 Integration with Other Tools

LangChain integrates with a variety of tools and services to enhance its capabilities:

5.6.1 APIs and Web Services

LangChain can interact with external APIs and web services to fetch additional data or perform actions based on the model's output.

5.6.2 Databases

Integration with databases allows LangChain to store and retrieve information, enabling applications that require persistent data storage.

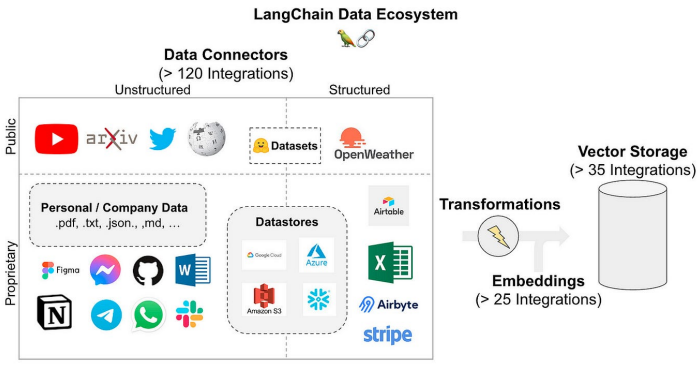


Figure 5.6: LangChain integration

5.7 Conclusion

LangChain provides a robust framework for building applications with large language models. Its modular design, advanced features, and ease of integration make it a valuable tool for developers working with LLMs. By leveraging LangChain, developers can create sophisticated applications that harness the power of modern language models to achieve a wide range of tasks.

6. Docker for Deployment

As the final step for our project, we want to deploy it. However, due to hardware and software limitations, instead of using a hosted AWS service or Microsoft Azure, we will use Docker. This way, users who download the repository can simply follow two instructions to access a virtual machine with all the required software pre-installed, ensuring it works perfectly regardless of their system. Below is the Dockerfile that will allow us to achieve this (Docker must be installed on your system):

```
# Use an official Python runtime as a parent image
FROM python:3.10-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at
   ↪ /app
COPY requirements.txt .

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Copy the current directory contents into the container at
   ↪ /app
COPY . .

# Make port 8000 available to the world outside this container
EXPOSE 5000

# Run app.py when the container launches
CMD ["python", "app.py"]
```

With that, the user will need to run the following two commands to use the application. They also have the option to download the repository and set up a virtual environment to install all the dependencies:

```
docker build -t prueba .
docker run -it prueba
```

7. Using Flask to deploy our system

Contents

Flask is a micro web framework for Python, designed to make web application development simple and efficient. Unlike full-stack frameworks, Flask is lightweight and provides the essentials to get an application up and running. This makes it an excellent choice for building and deploying small to medium-sized web applications and APIs. Flask was developed by Armin Ronacher as part of the Pallets Project and has gained widespread popularity due to its simplicity and flexibility.

Figure 7.1: Flask framework logo

7.1 Key Features of Flask

Flask provides several features that make it suitable for web application development:

- **Lightweight and Modular:** Flask is a micro-framework, meaning it has a minimalistic core, making it easy to extend with additional modules and libraries.
- **Flexible and Easy to Use:** It allows developers to build web applications quickly with minimal boilerplate code.
- **Built-in Development Server:** Flask includes a development server for testing and debugging, making it easy to develop and test applications locally.
- **Jinja2 Templating:** Flask integrates Jinja2, a powerful templating engine, to help create dynamic HTML pages.
- **RESTful Request Handling:** Flask supports RESTful URL routing, which is essential for building APIs.
- **Extensive Documentation:** Flask has comprehensive documentation and a large community, which helps developers find solutions and support easily.

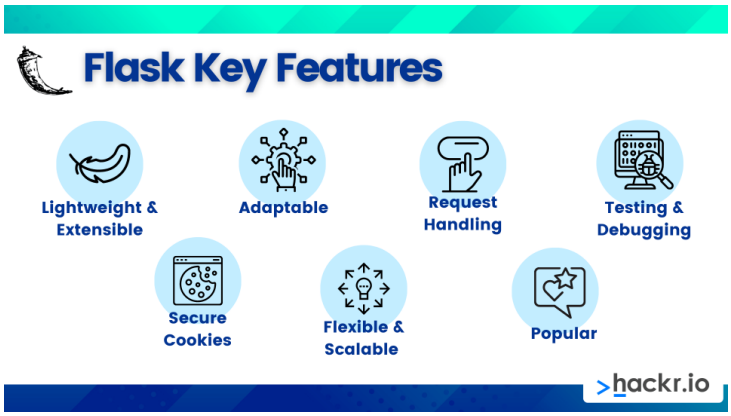


Figure 7.2: Flask framework main features

7.2 Rationale for Using Flask

In the context of our project, Flask was chosen for several reasons:

- **Simplicity:** The lightweight nature of Flask allows for a straightforward and clean architecture, which is ideal for our image description system.
- **Flexibility:** Flask's modular design means we can easily integrate additional features as needed, such as authentication, database connectivity, and API endpoints.
- **Rapid Development:** Flask's minimal setup and ease of use enable quick prototyping and development, which aligns well with our agile development approach.
- **Scalability:** While being simple, Flask can be scaled with the use of extensions and third-party libraries, making it suitable for scaling the image description system if needed.

7.3 Setting Up a Flask Application

Setting up a Flask application is straightforward. The following steps outline the basic setup for a Flask application:

7.3.1 Installation

First, Flask needs to be installed. This can be done using pip, the Python package manager:

```
pip install Flask
```

7.3.2 Creating a Basic Flask Application

Below is a simple example of a Flask application that serves a basic web page:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return 'Welcome to the Image Description System!'

if __name__ == '__main__':
    app.run(debug=True)
```

This script sets up a basic Flask application that responds with a welcome message when accessed at the root URL. The `app.run(debug=True)` line runs the application in debug mode, which is useful during development.

7.4 Deploying the Image Description System with Flask

In our project, Flask is used to deploy an image description system. The system is designed to process images and provide detailed descriptions, aiding visually impaired users by describing the content of images.

7.4.1 System Architecture

The image description system integrates several components:

- **Object Detection:** Utilizing YOLO v8 to detect objects in images.
- **Depth Mapping:** Using MiDaS to generate depth maps.
- **Language Processing:** Employing LangChain and transformers for natural language processing.
- **Weather Information:** Extracting weather details from images using a custom model.

- **Text to Speech Conversion:** Converting the final text output to speech to assist visually impaired users.

7.4.2 Flask Application Structure

The Flask application serves as the backbone of the system, handling image uploads, processing requests, and returning descriptions. Below is an overview of the Flask application structure:

- **Routes:** Defines different endpoints for uploading images, processing images, and returning descriptions.
- **Templates:** Uses Jinja2 for rendering HTML templates to display results.
- **Static Files:** Manages CSS, JavaScript, and images used in the web interface.
- **Error Handling:** Includes mechanisms to handle exceptions and provide meaningful error messages.

7.4.3 Implementation Example

Here is a snippet demonstrating how images are uploaded and processed using Flask:

```
from flask import Flask, request, render_template, redirect,
   ↪ url_for
from werkzeug.utils import secure_filename
import os

app = Flask(__name__)
UPLOAD_FOLDER = 'uploads'
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER

@app.route('/', methods=['GET', 'POST'])
def upload_image():
    if request.method == 'POST':
        if 'file' not in request.files:
            return redirect(request.url)
        file = request.files['file']
        if file.filename == '':
            return redirect(request.url)
        if file:
            filename = secure_filename(file.filename)
            filepath =
               ↪ os.path.join(app.config['UPLOAD_FOLDER'],
               ↪ filename)
            file.save(filepath)
            # Process the image and return the description
            description = process_image(filepath)
            return render_template('result.html',
               ↪ description=description)
    return render_template('upload.html')
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

In this example, the image is uploaded via a web form, saved in the `uploads` folder, and then processed by a function named `process_image()`.

7.5 Showcase

Whether you installed the virtual environment on your computer or created the Docker image, once everything is set up, you will be able to test the application on your localhost. To begin, you will see the following screen:

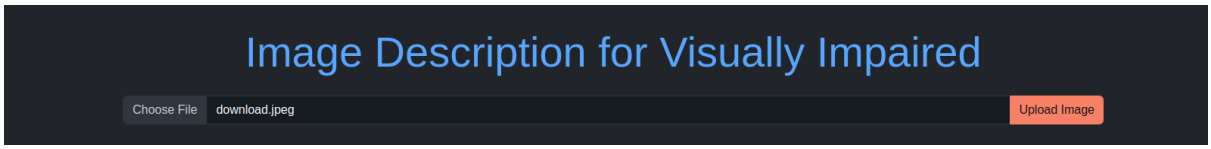


Figure 7.3: First screen

If you choose any picture and press 'Upload Image,' after about half a minute, you will see a screen displaying the image you selected, a depth heat map, and an image generated by the YOLO model with all detected entities. It also shows how long the app took to process everything. Additionally, you will receive a description generated by the LLM, summarizing the processed information. Finally, there is a button that allows the generated text to be played through the microphone, so blind users can hear the description:

Figure 7.4: Result screen

8. Bibliography

[1] **Ultralytics Documentation**, *Ultralytics Documentation*, Aug 2024, https://docs.ultralytics.com/ pages

[2] **MiDaS**, *MiDaS official repository*, Aug 2024, https://github.com/isl-org/MiDaS pages

[3] **MiDaS**, *MiDaS official torch documentation*, Aug 2024, https://pytorch.org/hub/intelisl_midas_v2/ pages

[4] **MiDaS**, *MiDaS official first research*, Aug 2024, https://arxiv.org/abs/1907.01341 pages

[5] **Ultralytics Documentation for metrics**, *Ultralytics Documentation*, Aug 2024, https://docs.ultralytics.com/guides/yolo-performance-metrics/#introduction pages

[6] **Langchain**, *Langchain official repository*, Aug 2024, https://github.com/langchain-ai/langchain pages