Ministère de l'Enseignement Supérieur
Et de la Recherche Scientifique
IT Business School

Dog Breed Identification Using Deep Learning

Realized By
Rihab Boutrif

Academic Year 2024–2025

Contents

List of Figures

Chapter 1

Introduction Chapter

Introduction

Dog breed identification refers to the process of determining the breed of a dog based on physical characteristics such as size, coat, color, and facial structure. With hundreds of recognized dog breeds across the world, the task of accurate breed recognition poses a significant challenge. Many breeds share similar features, and crossbreeding further complicates identification. This classification is important in fields like veterinary care, pet adoption, and dog training, where knowing the breed can influence health diagnostics, behavioral expectations, and lifestyle compatibility.

Problem Statement

Despite the importance of identifying dog breeds accurately, doing so manually is time-consuming, subjective, and often inaccurate. Even experts may struggle to differentiate between visually similar breeds. Additionally, not everyone has access to professional opinion or genetic testing. The challenge, therefore, lies in creating a reliable, automated system capable of identifying a dog's breed from a simple image, even when breeds share subtle visual similarities.

Chapter 2

Deep Learning Background

Deep Learning Overview

Deep learning is a subset of machine learning that utilizes artificial neural networks inspired by the human brain. These networks consist of layers of interconnected nodes (neurons) that process data hierarchically. Deep learning models are particularly powerful in handling unstructured data like images, audio, and text, making them ideal for tasks such as image classification, object detection, and natural language processing.

Image Classification with Deep Learning

Image classification is a core task in computer vision where the goal is to assign a label to an input image. In deep learning, this is typically achieved using Convolutional Neural Networks (CNNs), which are highly effective at capturing spatial hierarchies in images. By learning filters at multiple levels, CNNs can detect edges, textures, shapes, and higher-level features relevant for classification.

Image Preprocessing

Before feeding images into a neural network, it is essential to preprocess them to ensure consistency and improve model performance. Common preprocessing steps include:

- **Resizing:** Ensuring all images have the same dimensions.

- **Normalization:** Scaling pixel values (e.g., to a [0, 1] range).
- **Augmentation:** Applying random transformations (rotation, flip, zoom, etc.) to improve generalization.
- **Shuffling:** Mixing the dataset to reduce learning bias.

Convolutional Neural Networks (CNNs)

CNNs are a specialized type of neural network designed for processing grid-like data such as images. A typical CNN architecture includes:

- **Convolutional Layers:** Apply filters to extract features from the input image.
- **Activation Functions:** Usually ReLU, adding non-linearity.
- **Pooling Layers:** Reduce the spatial size of feature maps. Max pooling is the most common type, which selects the maximum value in a region to retain important features while reducing dimensionality.
- **Fully Connected Layers:** Connect all neurons for final classification.
- **Softmax Layer:** Produces a probability distribution across output classes.

Filters in convolutional layers slide over the input image and help extract key patterns such as edges and textures. These learned filters increase in complexity as you go deeper in the network.

Max pooling reduces computation and controls overfitting by downsampling feature maps while preserving the most important features.

Feature maps are the output of the convolution layers, capturing spatial patterns in the image.

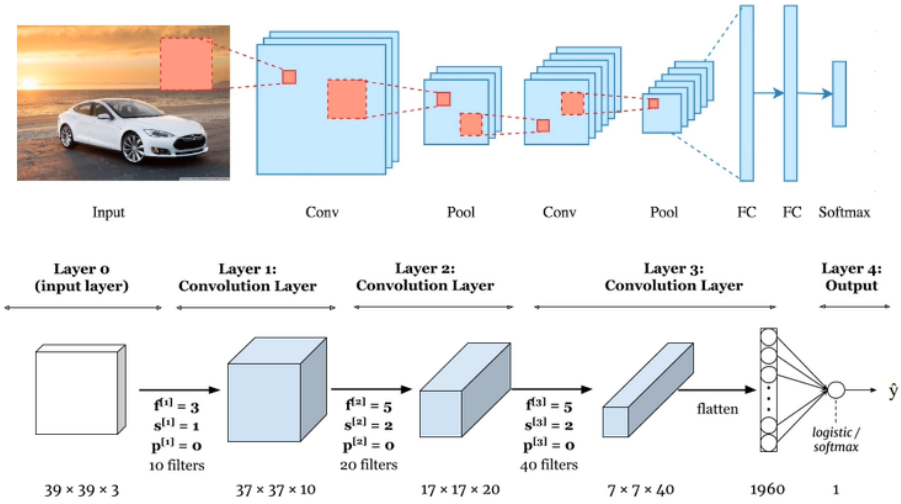


Figure 2.1: CNN architecture

Chapter 3

Dataset and Preprocessing

Dataset

The dataset used for this project was sourced from Kaggle and consists of approximately 10,000 labeled images representing 120 dog breeds. I used 8300 images for training and 839 validation and the remaining 820 for testing.

Figure 3.1: Dataset Overview

Data Preprocessing with Keras ImageDataGenerator

To train a deep learning model effectively, the input data must be preprocessed and formatted correctly. In this project, I used the ImageDataGenerator class from tensorflow.keras.preprocessing.image, which streamlines the process of preparing and loading image data.

Key preprocessing steps:

- **Rescaling:** All image pixel values were rescaled to a [0, 1] range by dividing by 255.
- **Target size:** All images were resized to 224x224 pixels.
- **Batching:** Images were processed in batches of 32 for efficient memory use.
- **Class Mode:** Labels were automatically one-hot encoded using the "categorical" setting.
- **Shuffling:** Training data was shuffled to avoid learning order bias.

```python
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(rescale=1./255)

train_generator = datagen.flow_from_dataframe(
    dataframe=df_train,
    x_col='filename',
    y_col='class',
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',  # Converts integer labels to one-hot automatically
    shuffle=True
)

val_generator = datagen.flow_from_dataframe(
    dataframe=df_val,
    x_col='filename',
    y_col='class',
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    shuffle=False
)
```

Figure 3.2: Dataset Overview

This setup allowed me to efficiently feed training and validation data into my model with minimal code and maximum flexibility.

Chapter 4

Modeling and Training and Evaluation

Model 1: CNN From Scratch

In the initial phase of this project, I built a custom CNN model from scratch. The architecture included multiple convolutional and pooling layers, followed by fully connected layers for classification. The input images were resized to 224x224 pixels and passed through the network.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 224, 224, 16)	448
max_pooling2d (MaxPooling2D)	(None, 112, 112, 16)	0
conv2d_1 (Conv2D)	(None, 110, 110, 32)	4,640
max_pooling2d_1 (MaxPooling2D)	(None, 55, 55, 32)	0
conv2d_2 (Conv2D)	(None, 53, 53, 64)	18,496
max_pooling2d_2 (MaxPooling2D)	(None, 26, 26, 64)	0
conv2d_3 (Conv2D)	(None, 24, 24, 128)	73,856
max_pooling2d_3 (MaxPooling2D)	(None, 12, 12, 128)	0
conv2d_4 (Conv2D)	(None, 10, 10, 64)	73,792
max_pooling2d_4 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_5 (Conv2D)	(None, 3, 3, 64)	36,928
max_pooling2d_5 (MaxPooling2D)	(None, 1, 1, 64)	0
flatten (Flatten)	(None, 64)	0
dense (Dense)	(None, 256)	16,640
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 120)	30,840

Figure 4.1: Model Summary

However, during training, the model showed poor learning behavior. The accuracy remained low, and the loss did not improve significantly over epochs.

```
256/256 ━━━━━━━━━━━━━━━━━━━━ 0s 407ms/step - accuracy: 0.0112 - loss: 4.7759
Epoch 15: val_loss improved from 4.77548 to 4.77543, saving model to best_model.keras
256/256 ━━━━━━━━━━━━━━━━━━━━ 109s 425ms/step - accuracy: 0.0112 - loss: 4.7759 - val_accuracy: 0.0122 - val_loss: 4.7754
Epoch 16/25
256/256 ━━━━━━━━━━━━━━━━━━━━ 0s 413ms/step - accuracy: 0.0132 - loss: 4.7792
Epoch 16: val_loss did not improve from 4.77543

Epoch 16: ReduceLROnPlateau reducing learning rate to 4.0000001899898055e-05.
256/256 ━━━━━━━━━━━━━━━━━━━━ 144s 433ms/step - accuracy: 0.0132 - loss: 4.7792 - val_accuracy: 0.0122 - val_loss: 4.7754
Epoch 17/25
256/256 ━━━━━━━━━━━━━━━━━━━━ 0s 438ms/step - accuracy: 0.0131 - loss: 4.7763
Epoch 17: val_loss did not improve from 4.77543
256/256 ━━━━━━━━━━━━━━━━━━━━ 117s 455ms/step - accuracy: 0.0131 - loss: 4.7763 - val_accuracy: 0.0122 - val_loss: 4.7754
Epoch 18/25
256/256 ━━━━━━━━━━━━━━━━━━━━ 0s 418ms/step - accuracy: 0.0120 - loss: 4.7771
Epoch 18: val_loss improved from 4.77543 to 4.77543, saving model to best_model.keras
256/256 ━━━━━━━━━━━━━━━━━━━━ 113s 440ms/step - accuracy: 0.0120 - loss: 4.7771 - val_accuracy: 0.0122 - val_loss: 4.7754
Epoch 19/25
256/256 ━━━━━━━━━━━━━━━━━━━━ 0s 408ms/step - accuracy: 0.0130 - loss: 4.7756
Epoch 19: val_loss improved from 4.77543 to 4.77543, saving model to best_model.keras
```

Figure 4.2: Cnn from scratch trainning phase

Model 2: CNN From Scratch + Data augmentation

In image classification tasks, data augmentation plays a crucial role in improving model performance and robustness. Since image datasets can often be limited in size or variety, augmentation techniques help simulate the diversity found in real-world conditions, preventing the model from overfitting to specific patterns seen in the training set. Common augmentation strategies include geometric transformations such as random flipping, rotation, scaling, cropping, and translation, which teach the model to recognize objects regardless of their orientation or position. Additionally, color-based transformations like brightness adjustment, contrast changes, saturation variation, and color jittering allow the model to become resilient to lighting conditions and color shifts. Injecting noise, applying blurring, sharpening, or randomly erasing parts of the image (Cutout) can further help the model learn to focus on important features rather than memorizing exact pixel patterns.

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    zoom_range=0.15,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    fill_mode="nearest"
)
```

Figure 4.3: Data Augmentation

4.0.1 Result after Data Augmentation

Despite using data augmentation, my model failed to learn effectively. The augmentations didn't help it extract meaningful patterns, and it remained stuck with poor accuracy or erratic loss behavior. This suggests that the model lacked sufficient capacity or prior knowledge to handle the complexity of the task. Given these challenges, switching to a pretrained model becomes a logical next step. Pretrained models come with a rich set of learned features from large datasets, making it easier to fine-tune them for your specific task and achieve better learning dynamics.

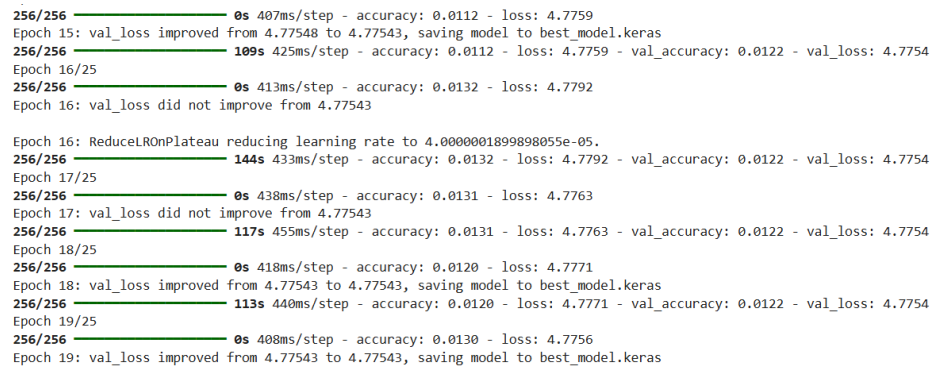


Figure 4.4: Cnn from scratch trainning phase + data augmentation

Pretrained Models and Transfer Learning

To address the limitations of training from scratch, I turned to transfer learning using pre-trained models. These models are trained on large datasets like ImageNet and have already learned useful feature representations. By reusing their convolutional base and fine-tuning it for my task, I could leverage their performance while saving time and computational resources.

ResNet50V2 Architecture

Residual Network (ResNet) is a specific type of neural network which is used for many computer vision problems. ResNet contains convolutional, pooling, activation and fully-connected layers stacked one of the other. A convolutional neural network is a type of deep neural network, which is used for image processing and its classification. As the name suggests, Convolutional Network helps for classifying complex images by multiplying pixel value with weights and then summing them.

These layers of ResNet are pre-trained on more than a million of images from the ImageNet database. Due to many layers, ResNet solves complex problems and increases model accuracy and performance.

Every ResNet uses an initial filter or kernel of 3×3 and 7×7 size with a stride of 2. There are many versions of ResNet. In this project, we will be using Resnet50V2 (version 2) which is 50 layers deep and applies Batch Normalization, RELU activation function before the input is multiplied by

convolutional operations(weight matrix).

Model 2: Dog Breed Identification with ResNet50V2

For the second model, I used ResNet50V2 as the base, excluding the top classification layer and adding a custom dense layer suited to my dog breed classification task. I froze the pretrained layers initially, then fine-tuned some of the deeper layers.

```python
from tensorflow.keras import layers, Model
from tensorflow.keras.applications import ResNet50V2

# Load base model
resnet50_v2 = ResNet50V2(
    include_top=False,
    input_shape=(224, 224, 3),
    weights='imagenet'
)

def build_resnet_model(model_name='ResNet50V2', print_summary=True):
    # Freeze all base model layers
    for layer in resnet50_v2.layers:
        layer.trainable = False

    # Input layer
    model_input = resnet50_v2.input

    # Extract last layer from base model
    hidden = resnet50_v2.output

    # Flatten + classification head
    hidden = layers.Flatten()(hidden)
    output = layers.Dense(units=120, activation='softmax')(hidden)  # adjust units as needed

    # Final model
    model = Model(inputs=model_input, outputs=output, name=model_name)
```

Figure 4.5: Resnet model building

This model showed a significant improvement over the custom CNN. Accuracy increased steadily, and validation loss decreased, indicating better generalization.

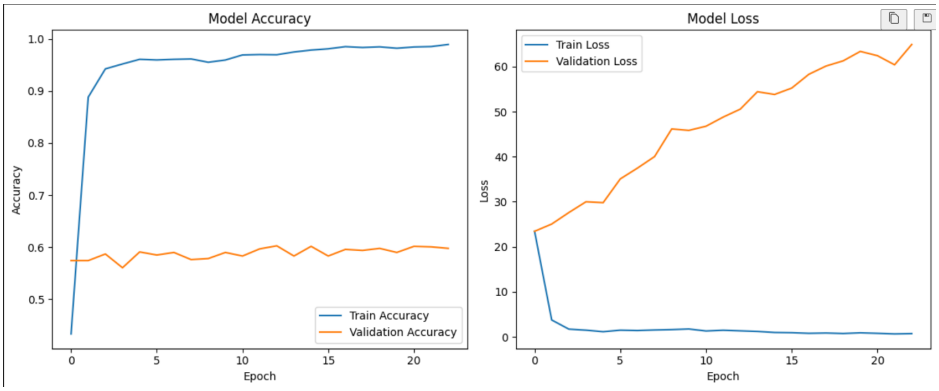


Figure 4.6: Training Results

Model Evaluation

Training and Validation Curves Analysis

The training and validation curves clearly demonstrate severe overfitting in the model. Here is a detailed breakdown of the plots:

- **Accuracy Plot (Left)**
  - **Training Accuracy:** Increases rapidly and reaches approximately 99% early in the training process.
  - **Validation Accuracy:** Remains low, around 58–60%, and relatively flat throughout the training epochs.
- **Loss Plot (Right)**
  - **Training Loss:** Decreases sharply and remains consistently low across epochs.
  - **Validation Loss:** Increases steadily and significantly over time.

Evaluation Summary

The model performs extremely well on the training set but fails to generalize to unseen validation data. This behavior indicates that the model has likely memorized the training data, leading to overfitting, rather than learning generalizable patterns that would perform well on new, unseen data.

Prediction Results

Figure 4.7: Dataset Overview

	precision	recall	f1-score	support
affenpinscher	0.54	0.88	0.67	8
afghan_hound	0.44	1.00	0.62	12
african_hunting_dog	1.00	0.89	0.94	9
airedale	0.50	0.73	0.59	11
american_staffordshire_terrier	0.00	0.00	0.00	7
appenzeller	0.33	0.50	0.40	8
australian_terrier	0.41	0.70	0.52	10
basenji	0.42	0.73	0.53	11
basset	0.67	0.75	0.71	8
beagle	0.83	0.45	0.59	11
bedlington_terrier	0.58	0.78	0.67	9
bernese_mountain_dog	0.91	0.91	0.91	11
black-and-tan_coonhound	0.56	0.62	0.59	8
blenheim_spaniel	0.73	0.80	0.76	10
bloodhound	1.00	0.62	0.77	8
bluetick	0.86	0.75	0.80	8
border_collie	0.44	1.00	0.61	7
border_terrier	0.70	0.78	0.74	9
borzoi	1.00	0.71	0.83	7
boston_bull	0.80	0.89	0.84	9
bouvier_des_flandres	0.50	0.44	0.47	9
boxer	1.00	0.57	0.73	7
brabancon_griffon	0.50	0.86	0.63	7
...				
accuracy			0.60	1023
macro avg	0.66	0.59	0.58	1023

Figure 4.8: Classification Report

Conclusion

This project highlights the power of deep learning and transfer learning in solving complex image classification problems such as dog breed identification. While building a CNN from scratch helped in understanding the fundamentals, it lacked the performance needed for this task. Utilizing a pretrained model like ResNet50V2 provided a more robust solution, achieving higher accuracy and better generalization. now let's deploy the solution as a web application for real-world use.

Chapter 5

Dog Breed Identification Application

5.1 Introduction

This chapter describes the cloud deployment architecture and technologies used to develop and deploy our deep learning application for dog breed identification. The application uses a deep learning model integrated with a **Streamlit** web interface and was deployed on **Google Cloud Platform (GCP)** using **Docker** and **Google Kubernetes Engine (GKE)**.

5.2 Docker Overview

Docker is a platform that enables developers to build, share, and run applications within containers.

Containers are lightweight, portable, and self-sufficient units that package software along with all its dependencies, libraries, and configuration files. This ensures that the application behaves the same across all environments, eliminating the common "it works on my machine" problem.

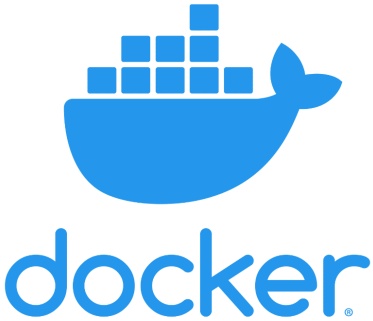


Figure 5.1: docker

Why use Docker?

- **Portability**: Containers can run on any platform that supports Docker.
- **Consistency**: Ensures identical environments for development, testing, and production.
- **Isolation**: Applications are isolated from each other and the underlying host system.
- **Efficiency**: Containers are lightweight compared to traditional virtual machines.

5.3 Kubernetes Overview

Kubernetes is an open-source system for automating the deployment, scaling, and management of containerized applications.

Originally developed by Google, Kubernetes allows users to manage clusters of containers across multiple machines, ensuring high availability, scalability, and efficient resource utilization.

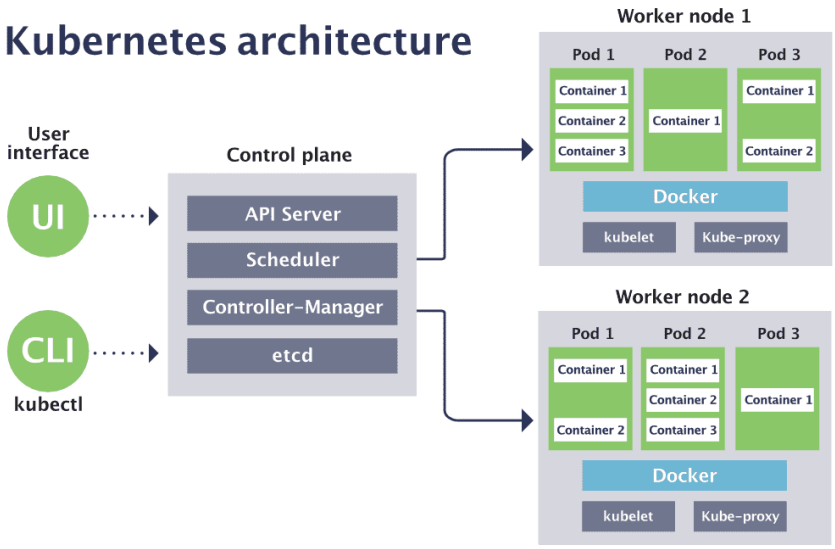


Figure 5.2: Kubernetes Architechture

Why use Kubernetes?

- **Orchestration**: Automates deployment and scaling of containers.
- **Load balancing**: Distributes network traffic to maintain application stability.
- **Self-healing**: Automatically replaces failed containers.
- **Declarative configuration**: Infrastructure is described with YAML/JSON files.

5.4 Google Cloud Platform (GCP)

Google Cloud Platform (GCP) is a suite of cloud computing services running on the same infrastructure Google uses internally.

GCP offers services in computing, storage, networking, machine learning, and security. It enables businesses and developers to build, deploy, and scale applications efficiently and securely.

Key GCP features used:

- **Google Kubernetes Engine (GKE)**: Managed Kubernetes clusters.
- **Cloud Storage**: Persistent storage for files and data.
- **Container Registry / DockerHub**: Managing Docker images.

Figure 5.3: GKE

5.5 Application Architecture: Dog Breed Identification

5.5.1 Streamlit Web Application

The front-end of the application was built using **Streamlit**, a Python-based framework that enables the quick development of web applications for data science and machine learning projects.

Figure 5.4: streamlit logo

The Streamlit app:

- Accepts user-uploaded images.
- Processes the image and passes it through the deep learning model.
- Displays the predicted dog breed.

The model was trained, optimized, and saved in Keras format (`model.keras`).

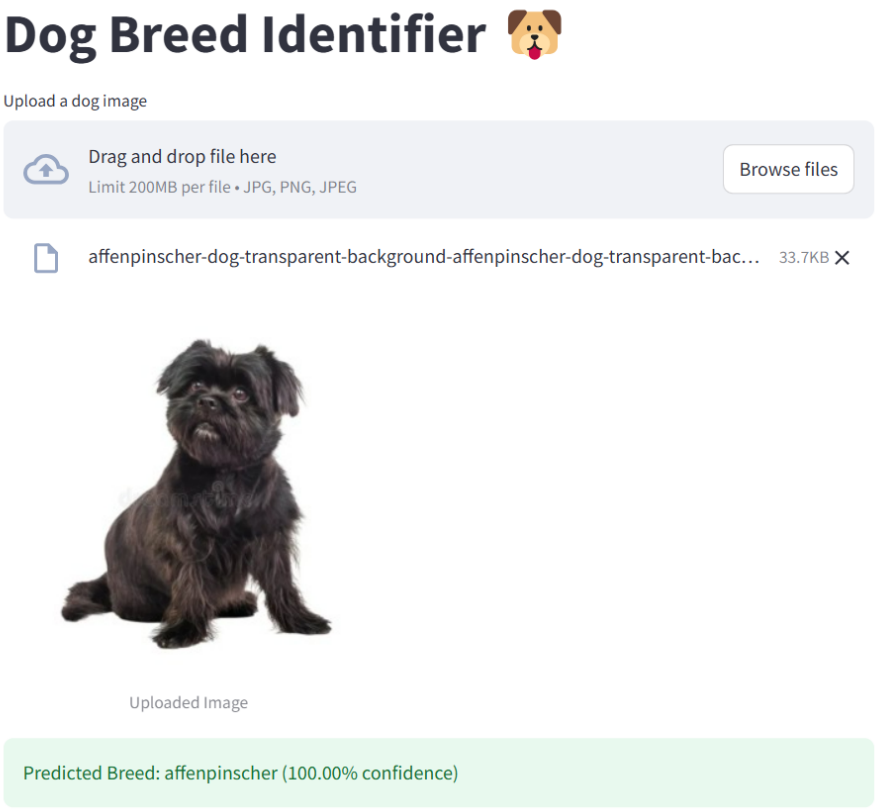


Figure 5.5: application

5.5.2 Docker Images: dev, test, prod

To prepare the application for deployment, three separate Docker images were created: Development (dev), Testing (test), and Production (prod), each with a specific purpose. In both dev and test environments, I worked extensively on the model loading and image preprocessing components. The dev image includes the full application stack (main.py, the app server, and all dependencies), making it ideal for local development and feature integration. This setup allows real-time testing of model integration and user interaction. The test image, on the other hand, is lightweight and tailored for running automated tests using tools like pytest to validate model behavior and preprocessing logic independently of the app interface. In the prod environment,

I focused on stability, performance, and scalability. The production image is optimized for serving predictions efficiently, using a rolling update strategy to minimize downtime during deployments and ensure high availability for users.

These images were then pushed to **DockerHub** to be pulled by the Kubernetes cluster.

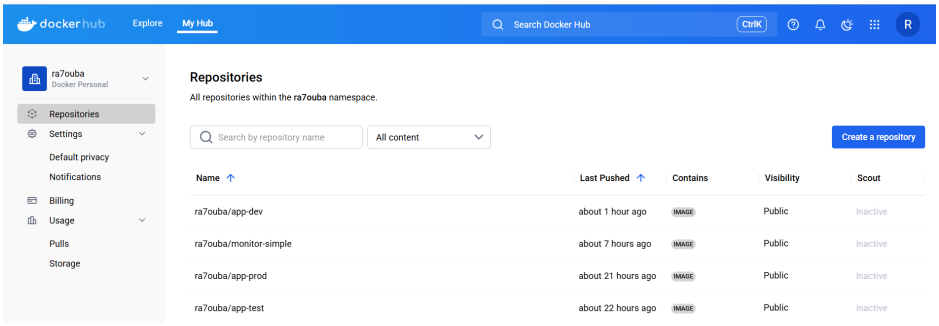


Figure 5.6: pushing images to dockerhub

5.5.3 Namespace Strategy

In Kubernetes, we created three separate namespaces:

- `dev`
- `test`
- `prod`

Importance of namespaces:

- Isolation between development, testing, and production environments.
- Better organization and easier access control.
- Allows different resource quotas and limits per environment.

5.5.4 Persistent Volume (PV) and Persistent Volume Claims (PVC)

In Kubernetes, Persistent Volumes (PV) and Persistent Volume Claims (PVC) provide durable storage solutions independent of the lifecycle of individual pods.

We created:

- **model-pv.yaml**: Defines the Persistent Volume.
- **model-pvc-dev.yaml**, **model-pvc-test.yaml**, **model-pvc-prod.yaml**: Define Persistent Volume Claims for each environment.

Importance of PV and PVC:

- Containers are ephemeral; data saved inside a container is lost after pod restarts or crashes.
- PV/PVC ensures that user-uploaded files (such as images for prediction) or application-generated data (saved in `/mnt/data`) are preserved.
- Enables the application to handle user inputs and persist temporary responses during runtime.

5.5.5 Monitoring Containers

Along with the main Streamlit container, each pod included a secondary **monitoring container** based on a basic Flask application.

This container exposed endpoints:

- `/health`: For liveness and readiness probes.
- `/predict`: to perform basic checks on prediction API.

Purpose of monitoring container:

- Detect application failures early.
- Allow Kubernetes to automatically restart unhealthy pods.
- Simplify monitoring and debugging.

5.5.6 Deployment Files

We created three deployment configurations:

- `deployment-dev.yaml`
- `deployment-test.yaml`
- `deployment-prod.yaml`

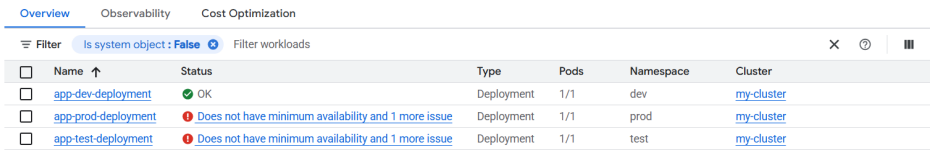


Figure 5.7: deployments

Each deployment:

- Pulled the corresponding Docker image.
- Mounted the model through the PVC.

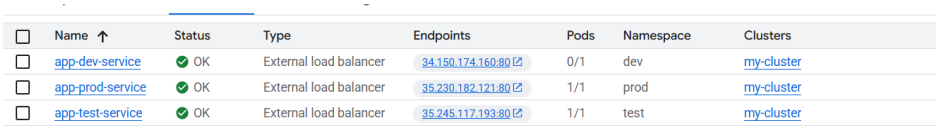
In my Kubernetes setup, I used the Recreate strategy for the development (dev) environment because it's simple and allows for a clean restart of the application with every update. Since downtime is acceptable in development, replacing all Pods at once avoids version conflicts and ensures the latest changes are fully applied. On the other hand, for the production (prod) environment, I use the RollingUpdate strategy to ensure high availability. It updates Pods gradually, which allows the application to remain available during the deployment process and reduces the risk of service interruption for users.

5.5.7 Kubernetes Services

To expose the application, we created Kubernetes **Services**:

- `service-dev.yaml`
- `service-test.yaml`

- `service-prod.yaml`



Name	Status	Type	Endpoints	Pods	Namespace	Clusters
app-dev-service	OK	External load balancer	34.150.174.160:80	0/1	dev	my-cluster
app-prod-service	OK	External load balancer	35.230.182.121:80	1/1	prod	my-cluster
app-test-service	OK	External load balancer	35.245.117.193:80	1/1	test	my-cluster

Figure 5.8: services

These Services provided external LoadBalancers to expose the app publicly.

5.6 Perspectives and Lessons Learned

This project offered many important lessons:

- **Docker** simplified application portability and environment replication.
- **Kubernetes** automated deployment, scaling, and management.
- **Persistent storage** was critical for machine learning models.
- **Namespaces** improved organization and environment separation.
- **Monitoring containers** increased reliability and visibility into application health.

Managing applications on GCP using GKE demonstrated the power and flexibility of cloud-native architectures.

5.7 Conclusion

Building and deploying the Dog Breed Identification app on Google Cloud demonstrated how modern cloud-native technologies (Docker, Kubernetes, GCP) can greatly simplify and accelerate application development and deployment.

This architecture ensures scalability, maintainability, and reliability, providing a solid foundation for future machine learning application deployments.