AI-Based Image Classification Web Application

TABLE OF CONTENTS: -

Overview

📌 Overview of the Project

- This project is a web-based application that classifies images into one of three animal categories: **cats, dogs, or snakes**. It consists of:
- A trained **TensorFlow/Keras** model.
- A **FastAPI** backend for inference.
- A **Flask** frontend UI.
- Deployment using **Render**.
- Model storage on **Google Drive** due to file size limits on GitHub.

Model Details

- Framework: TensorFlow/Keras
- Input size: (224, 224)
- Classes: `['cats', 'dogs', 'snakes']`
- Exported format: `.h5`
- Model size: ~128MB
- Storage: Google Drive

🔧 *Backend – FastAPI*

Setup

- Handles image upload and classification.
- Downloads the `model.h5` from Google Drive on first run.
- Uses `gdown` to fetch the model.

API Endpoint

`POST /predict`

- Accepts: Image file (JPEG/PNG)
- Returns: Predicted class and confidence %(Probability)

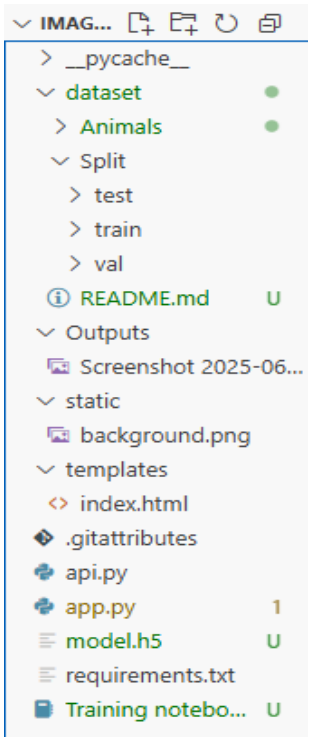
Frontend – Flask

Functionality:-

- Simple UI to upload image.
- Sends request to FastAPI backend.
- Displays the predicted result and confidence score.

Project Structure

- We are building a flask application which needs HTML pages stored in the templates folder and a python script app.py for scripting.

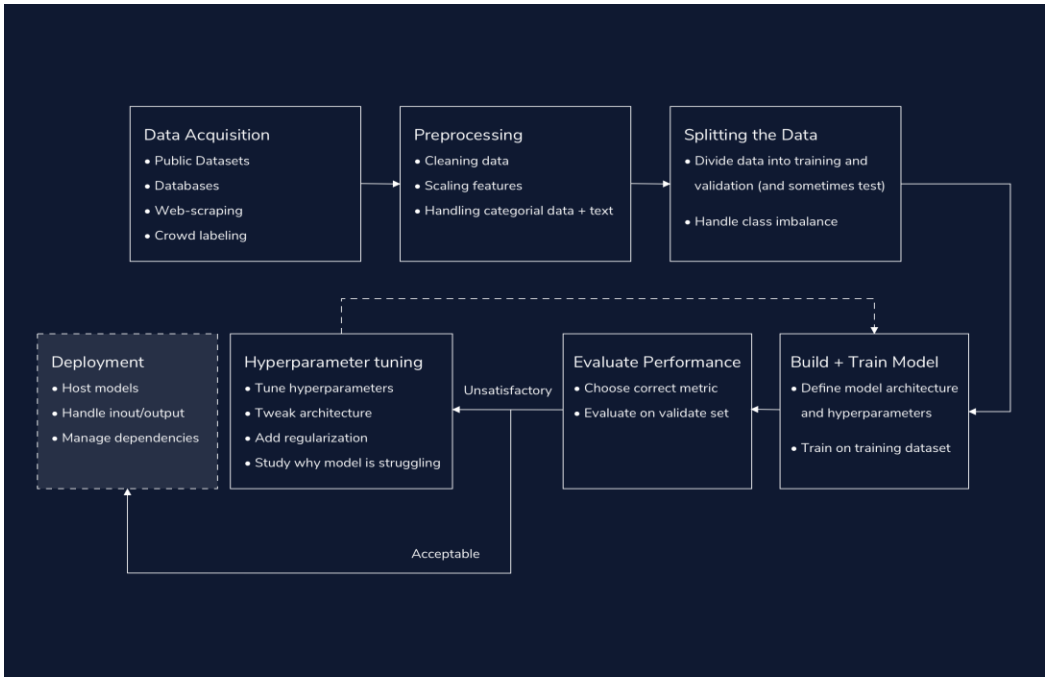


- Model.h5 is our saved model. Further we will use this model.
- Animal folder is the Dataset used

- The Notebook file contains procedure for building the model.

Theoretical Analysis

1. Block Diagram:-



2. Software designing

The following is the Software required to complete this project:

- **Google Collab**: Google Collab will serve as the development and execution environment for your predictive modeling, data preprocessing, and model training tasks. It provides a

cloud-based Jupyter Notebook environment with access to Python libraries and hardware acceleration.

- **Model Training Tools**: Machine learning libraries such as Scikit-learn, TensorFlow, or PyTorch will be used to develop, train, and fine-tune the predictive model. Regression or classification models can be considered, depending on the nature of the AQI prediction task.
- **UI Based on Flask Environment**: Flask, a Python web framework, will be used to develop the user interface (UI) for the system. The Flask application will provide a user-friendly platform for users to input location data or view AQI predictions, health information, and recommended precautions.

- **Software requirement:**
  - ➢ Python
  - ➢ HTML
  - ➢ Flask
  - ➢ Anaconda Distribution(For Jupyter Notebook)
  - ➢ git
  - ➢ Render (For Deployment)

Functional Requirements

1. Data Collection: -

- There are many popular open sources for collecting the data.

  Eg: kaggle.com, UCI repository, etc.
- Link:-

  Animal Image Classification Dataset

→ The Animal Image Classification Dataset is a comprehensive collection of images tailored for the development and evaluation of machine learning models in the field of computer vision.

→ It contains 3,000 JPG images, carefully segmented into three classes representing common pets and wildlife: cats, dogs, and snakes.

2. Dataset Preparation: -

Expected Folder Structure:

Organize your dataset into `train`, `val`, and `test` folders:

Split Ratio:

You can use this common split:

- **Train**: 70% (700 per class)
- **Validation**: 15% (150 per class)
- **Test**: 15% (150 per class)

→ We working with a classification dataset with **3 classes** (`cat`, `dog`, `snack`) and **1000 images per class**.

→ To train a reliable machine learning model, it's important to **split** the dataset into:

- **Training set (70%)**: Used to teach the model.
- **Validation set (15%)**: Used to tune the model
- **Test set (15%)**: Used to evaluate how well the final model performs on completely unseen data.

→ Instead of doing this manually, the Python script **automates the splitting** and **organizes** the images into folders.

Folder Creation:-

```
#classes

classes = ['cats', 'dogs', 'snakes']

#Train, val, test split ratio

split_ratio = [0.7, 0.15, 0.15] # 70% train, 15% val, 15% test

#Create folder structure

for split in ['train', 'val', 'test']:

      for class_name in classes:

          os.makedirs(os.path.join(base_dir, split, class_name),
     exist_ok=True)

#Split and copy files

for class_name in classes:

      class_dir = os.path.join(original_dataset_dir, class_name)

      images = os.listdir(class_dir) random.shuffle(images)

train_count = int(split_ratio[0] * len(images))
val_count = int(split_ratio[1] * len(images))

for i, img_name in enumerate(images):
    src_path = os.path.join(class_dir, img_name)

    if i < train_count:
        dst_dir = os.path.join(base_dir, 'train', class_name)
    elif i < train_count + val_count:
        dst_dir = os.path.join(base_dir, 'val', class_name)
    else:
        dst_dir = os.path.join(base_dir, 'test', class_name)

    shutil.copy2(src_path, dst_dir)
```

print("✅ Dataset successfully split into train/val/test!")

3. *Model Training*

#Normalization

```
train_dir = r"C:\Users\vatha\Downloads\dataset\Split\train"
val_dir = r"C:\Users\vatha\Downloads\dataset\Split\val"
test_dir = r"C:\Users\vatha\Downloads\dataset\Split\test"
IMG_SIZE = (224, 224)
BATCH_SIZE = 32


train_datagen = ImageDataGenerator(
      rescale=1./255,
      rotation_range=20,
      width_shift_range=0.1,
      height_shift_range=0.1,
      shear_range=0.1,
      zoom_range=0.2, horizontal_flip=True,
      fill_mode='nearest' )
val_test_datagen = ImageDataGenerator(rescale=1./255)
```

#traing, testing and Validation sets

```
train_generator = train_datagen.flow_from_directory( train_dir,
target_size=IMG_SIZE, batch_size=BATCH_SIZE,
class_mode='categorical' )

val_generator = val_test_datagen.flow_from_directory( val_dir,
target_size=IMG_SIZE, batch_size=BATCH_SIZE,
class_mode='categorical' )

test_generator = val_test_datagen.flow_from_directory( test_dir,
target_size=IMG_SIZE, batch_size=BATCH_SIZE, class_mode='categorical',
shuffle=False )
```

Step	Purpose
ImageDataGenerator	Creates preprocessing pipeline: resize, normalize, augment
flow_from_directory()	Automatically loads and labels images from folders
class_mode='categorical'	Since you have 3 classes (multi-class classification)
rescale=1./255	Normalizes pixel values to range 0–1

#visualization

```
images, labels = next(train_generator)

plt.figure(figsize=(12, 8))
for i in range(9):
    plt.subplot(3, 3, i+1)
    plt.imshow(images[i])
    plt.title(f"Class: {labels[i].argmax()}")
    plt.axis("off")
plt.tight_layout()
plt.show()
```

#CNN Model Building

```
import tensorflow as tf from tensorflow.keras import layers, models

#Get the number of classes from the generator

num_classes = train_generator.num_classes

#Build a simple CNN model

model = models.Sequential([ layers.Input(shape=(224, 224, 3)),

layers.Conv2D(32, (3, 3), activation='relu'),
layers.MaxPooling2D(2, 2),

layers.Conv2D(64, (3, 3), activation='relu'),
layers.MaxPooling2D(2, 2),

layers.Conv2D(128, (3, 3), activation='relu'),
layers.MaxPooling2D(2, 2),

layers.Flatten(),
layers.Dense(128, activation='relu'),
layers.Dropout(0.5),

layers.Dense(num_classes, activation='softmax')  # For multi-class


#Compile the model

model.compile( loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'] )

#Show model summary

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 222, 222, 32)      896

 max_pooling2d (MaxPooling2  (None, 111, 111, 32)      0
 D)

 conv2d_1 (Conv2D)           (None, 109, 109, 64)      18496

 max_pooling2d_1 (MaxPoolin  (None, 54, 54, 64)        0
 g2D)

 conv2d_2 (Conv2D)           (None, 52, 52, 128)       73856

 max_pooling2d_2 (MaxPoolin  (None, 26, 26, 128)       0
 g2D)

 flatten (Flatten)           (None, 86528)             0

 dense (Dense)               (None, 128)               11075712

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 3)                 387

=================================================================
Total params: 11169347 (42.61 MB)
Trainable params: 11169347 (42.61 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

#Train the model

```
history = model.fit( train_generator, epochs=1,
validation_data=val_generator )
```

#Save The Model

```
model.save("model.h5")
```

4. API Development with Fast API (api.py)

#Importing libraries

```
from fastapi import FastAPI, File, UploadFile
from fastapi.responses import JSONResponse
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
```

```
import numpy as np
from PIL import Image
import io
import os
import gdown
from tensorflow.keras.models import load_model
```

#As model.h5 file too big, so I have uploaded in the drive and downloading here.

```
model_dir = "model" model_path = os.path.join(model_dir, "model.h5")
 if not os.path.exists(model_path):
      print("Downloading model...")
       url =
      "https://drive.google.com/uc?id=184B4sZiOg23lW7MhfP11Moq2cuSTYvoz"
      os.makedirs(model_dir, exist_ok=True)
       gdown.download(url, model_path, quiet=False)
```

#Loading the trained model

```
model = load_model(model_path)
labels = ['cats', 'dogs', 'snakes']
```

#Create the fastapi app

```
app = FastAPI(title="Animal Image Classifier API")
```

#actual predition root

```
@app.post("/predict")
async def predict(file: UploadFile = File(...)):
  try:
       # Read and open the image using read()
       contents = await file.read()
       img = Image.open(io.BytesIO(contents)).convert('RGB')
```

```
        img = img.resize((224, 224)) # Resize to dimenstion (224x224)


        # Preprocess image
        img_array = image.img_to_array(img)
        img_array = np.expand_dims(img_array, axis=0)
        img_array = img_array / 255.0  #  normalization

        #  predictions
        predictions = model.predict(img_array)
        predicted_index = np.argmax(predictions)
        predicted_class = labels[predicted_index]
        confidence = float(np.max(predictions))


    return {
        "predicted_class": predicted_class,
        "confidence": round(confidence * 100, 2)
    }

except Exception as e:
    return JSONResponse(status_code=500, content={"error": str(e)})


if __name__ == "__main__":

        import uvicorn

     uvicorn.run("api:app", host="0.0.0.0", port=int(os.environ.get("PORT",
8000)))
```

Swagger UI: -

→ Swagger UI is an interactive documentation interface automatically provided by **FastAPI**. It allows developers and users to **interact with the API endpoints** via a web-based interface without needing external tools like Postman.

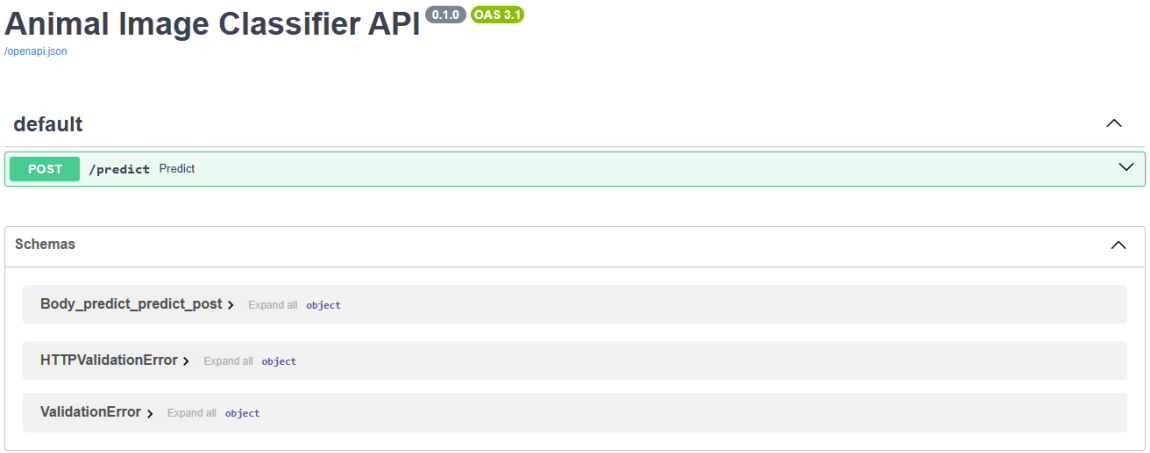
→ Swagger UI is a **web interface** for your API documentation generated from your FastAPI code. It reads the metadata, parameters, responses, and descriptions from your FastAPI endpoints and displays it in a user-friendly, interactive format.

→ Run your app:

```
uvicorn api:app --host 0.0.0.0 --port 8000
```

→ Visit:

http://localhost:8000/docs



5. *Frontend Using Flask*

#Importing libraries

```
from flask import Flask, render_template, request import requests
app = Flask(name)
```

```
api_url = "http://127.0.0.1:8000/predict" # the backend link
@app.route("/", methods=["GET", "POST"])

def index():
    prediction = None
    confidence = None
    if request.method == "POST":
        file = request.files["file"]
        if file:
            files = {"file": (file.filename, file.stream, file.mimetype)}
            response = requests.post(api_url, files=files)
            print(response)
            if response.status_code == 200:
                result = response.json()
                prediction = result.get("predicted_class")
                confidence = result.get("confidence")
            else:
                prediction = "Error from API"
    return render_template("index.html", prediction=prediction,
confidence=confidence)

if name == "main":
       app.run(debug=True)
```

Notes:-

→ **`flask_app.py`** serves the UI on `localhost:5000`

→ **`index.html`** is loaded by Flask to show the form and prediction

→ When you upload an image, Flask sends it to **FastAPI (`api.py`)**

→ **FastAPI** loads the model (`animal_classifier_model.h5`) and returns a prediction

Deployment

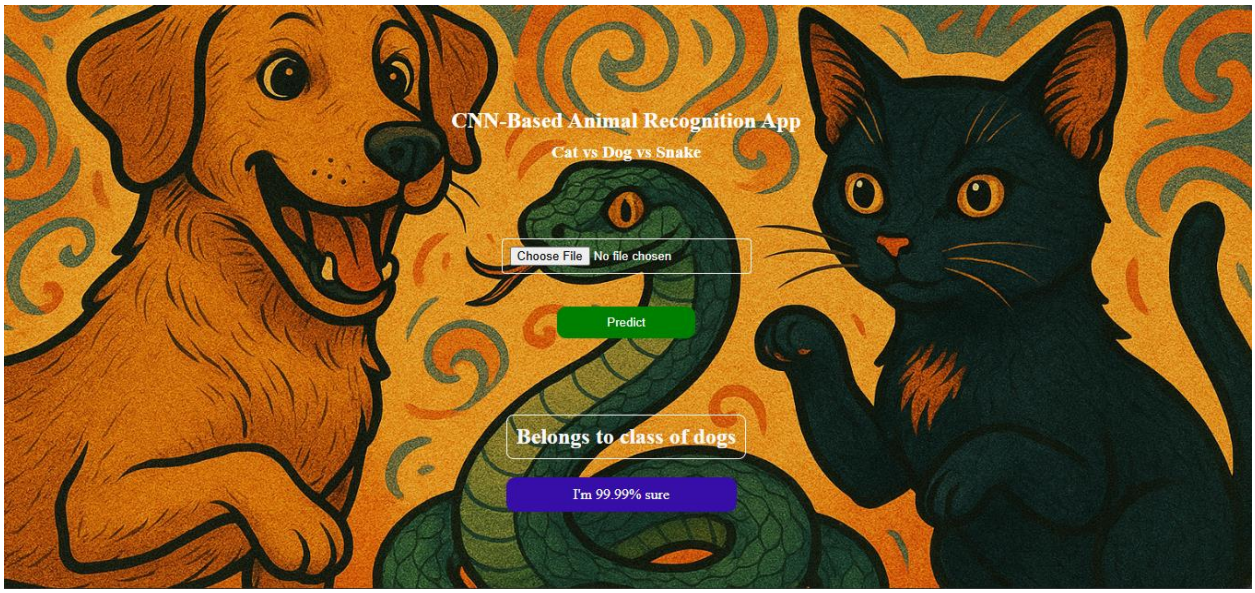
Deploy Backend on Render

1. Go to https://render.com
2. Click **"New > Web Service"**
3. Connect your **GitHub repo** containing backend
4. Set:
    a. **Environment**: Python
    b. **Build Command**: pip install -r requirements.txt
    c. **Start Command**: uvicorn api:app

Deploy Frontend on Render

1. Again, create **New Web Service** on Render
2. Choose frontend GitHub repo
3. Set:
    a. **Environment**: Python
    b. **Build Command**: pip install -r requirements.txt
    c. **Start Command**: gunicorn app:app
4. Done! Flask app will be deployed to another public URL.

The Output: -



Conclusion: -

This project successfully demonstrates the practical application of **Machine Learning (ML)** and particularly Convolutional Neural Networks (CNNs), for solving a real-world image classification problem. By training a CNN model on a dataset of animal images (cats, dogs, and snakes), the system can accurately predict the class of an uploaded image.

The key steps included:

- Preprocessing and augmenting the image data
- Building and training a CNN model using TensorFlow/Keras
- Normalizing input data to enhance model performance
- Evaluating the model on validation and test datasets
- Saving the trained model (model.h5) for reuse in deployment

To make the solution accessible and interactive:

- A FastAPI-based backend was developed to serve the model via a REST API
- A simple Flask frontend allows users to upload images and view predictions in real time

- The entire application was successfully deployed on Render, and the large model file was integrated using Google Drive

This project showcases a complete end-to-end pipeline—from data preparation and model training to full-stack deployment. It highlights how modern deep learning techniques can be effectively combined with web technologies to create scalable and user-friendly AI applications.