**UXV CENTER GmbH**

# AI-based Facial Recognition and Geometric Target Tracking

**(Complete Solution)**

Date：04.08.2024

Author: Anas Skiti (CEO)

## Introduction

### Overview

This project aims to develop an AI-powered solution for facial recognition and geometric target tracking using VTOL or Quadcopter drone technology. The system is designed to provide precise and real-time monitoring and tracking of targets, enhancing efficiency and accuracy in various applications. Given the challenge of not having actual drones for testing, the solution includes a simulated environment to emulate drone movements and sensor data.

### System Components

The solution consists of several key components:

1. **Database**: MongoDB is used for flexible and scalable data storage.
2. **Backend**: An ASP.NET Core API manages data processing and communication.
3. **Frontend**: A React-based user interface (using TypeScript) provides visualization and control.
4. **AI Models**: Deep learning models for facial recognition and target tracking are trained and deployed using TensorFlow.
5. **Simulation Unit**: A software module simulates drone movements and sensor data generation.

### Objectives

The main objectives of this project are:

1. **Design and Implement Drone Surveillance System**: Develop a scalable and robust system architecture for managing a fleet of drones.
2. **Real-Time Data Collection and Processing**: Enable real-time data collection using high-resolution cameras, infrared sensors, and GPS modules.
3. **AI-Based Facial Recognition and Biometrics**: Develop and integrate AI algorithms for high-accuracy facial recognition and biometric tracking.
4. **Automated Alerts and Incident Management**: Implement automated alert mechanisms and an incident management system.
5. **User-Friendly Interface**: Design a customizable dashboard for visualizing surveillance data and target status.
6. **Integration and Compatibility**: Develop API interfaces for seamless integration with other systems and ensure compatibility with various platforms.
7. **Data Security and Compliance**: Implement access control mechanisms and data encryption technologies to protect sensitive data.

8. **Testing, Deployment, and Maintenance**: Conduct comprehensive testing and establish a deployment and maintenance plan for regular updates and patches.

**Implementation Overview**

The implementation involves the following major steps:

1. **Backend API with ASP.NET Core and MongoDB**:
   - Setting up an ASP.NET Core API that interfaces with MongoDB to store and retrieve sensor data.
   - Implementing controllers and repositories to manage sensor data and facial recognition predictions.
2. **Frontend with React and TypeScript**:
   - Developing a React-based user interface that allows users to view sensor data, control the simulation, and see facial recognition results.
   - Creating components for sensor data display, control panel, and facial recognition results.
3. **AI Model Training and Serving with TensorFlow**:
   - Training a facial recognition model using TensorFlow and Keras.
   - Deploying the trained model using TensorFlow Serving to enable real-time predictions.
4. **Simulation Unit**:
   - Implementing a Python script to simulate sensor data generation and send this data to the backend API for processing.

## Step by Step Solution for Simulation Data

**Components:**

- **Database**: MongoDB for flexible and scalable data storage.
- **Backend**: ASP.NET Core API to manage data processing and communication.
- **Frontend**: React-based user interface for visualization and control.
- **AI Models**: Deep learning models for facial recognition and target tracking.
- **Simulation Unit**: Software module to emulate drone movements and sensors.

**1. Backend API with ASP.NET Core and MongoDB**

**Setting Up the Project:**

```
dotnet new webapi -n DroneSurveillanceAPI

cd DroneSurveillanceAPI
```

**appsettings.json:**

```
{
  "MongoConnection": {
    "ConnectionString": "mongodb://localhost:27017",
    "Database": "DroneSurveillanceDB"
  }
}
```

**DataContext/DataContext.cs:**

```
public class DataContext
{
    private readonly IMongoDatabase _database;

    public DataContext(IConfiguration configuration)
    {
        var client = new
MongoClient(configuration.GetConnectionString("MongoConnection:ConnectionString"));
        _database = client.GetDatabase("DroneSurveillanceDB");
    }

    public IMongoCollection<SensorData> SensorData =>
_database.GetCollection<SensorData>("SensorData");
}
```

**Models/SensorData.cs:**

```
public class SensorData
{
    public string Id { get; set; }
    public string SensorType { get; set; }
    public string Value { get; set; }
    public DateTime Timestamp { get; set; }
}
```

**Repositories/ISensorDataRepository.cs:**

```
public interface ISensorDataRepository
{
    Task AddSensorDataAsync(SensorData data);
    Task<IEnumerable<SensorData>> GetSensorDataAsync();
}
```

**Repositories/SensorDataRepository.cs:**

```
public class SensorDataRepository : ISensorDataRepository
{
    private readonly DataContext _context;

    public SensorDataRepository(DataContext context)
    {
        _context = context;
    }

    public async Task AddSensorDataAsync(SensorData data)
    {
        await _context.SensorData.InsertOneAsync(data);
    }

    public async Task<IEnumerable<SensorData>> GetSensorDataAsync()
    {
        return await _context.SensorData.Find(_ => true).ToListAsync();
    }
}
```

**Controllers/SensorDataController.cs:**

```
[ApiController]
[Route("api/[controller]")]
public class SensorDataController : ControllerBase
{
    private readonly ISensorDataRepository _repository;

    public SensorDataController(ISensorDataRepository repository)
    {
        _repository = repository;
    }

    [HttpPost]
    public async Task<IActionResult> AddSensorData(SensorData data)
    {
        await _repository.AddSensorDataAsync(data);
        return Ok();
    }

    [HttpGet]
    public async Task<IActionResult> GetSensorData()
    {
        var data = await _repository.GetSensorDataAsync();
        return Ok(data);
    }
}
```

**Controllers/FaceRecognitionController.cs:**

```
using Microsoft.AspNetCore.Mvc;
using System.Text;
using Newtonsoft.Json;

namespace DroneSurveillanceAPI.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class FaceRecognitionController : ControllerBase
    {
        private readonly HttpClient _httpClient;

        public FaceRecognitionController(HttpClient httpClient)
        {
            _httpClient = httpClient;
            _httpClient.BaseAddress = new Uri("http://localhost:8501"); // TensorFlow Serving URL
        }

        [HttpPost("predict")]
        public async Task<IActionResult> Predict([FromBody] ImageData imageData)
        {
            // Prepare the request payload with the correct input name
            var requestPayload = new
            {
                instances = new[]
                {
                    new Dictionary<string, object>
                    {
                        ["inputs"] = imageData.Image // Use the correct input name
                    }
                }
            };

            var jsonContent = new StringContent(JsonConvert.SerializeObject(requestPayload),
Encoding.UTF8, "application/json");

            var response = await _httpClient.PostAsync("/v1/models/face_recognition_model:predict",
jsonContent);

            if (!response.IsSuccessStatusCode)
            {
                return StatusCode((int)response.StatusCode, await response.Content.ReadAsStringAsync());
            }

            var result = await response.Content.ReadAsStringAsync();
            return Ok(result);
        }
    }

    public class ImageData
    {
        public float[][][][] Image { get; set; } // Ensure this matches the input tensor shape
    }
}
```

**Controllers/SimulationController.cs:**

```
using Microsoft.AspNetCore.Mvc;
using System.Text;
using Newtonsoft.Json;

namespace DroneSurveillanceAPI.Controllers
```

```
{
    [ApiController]
    [Route("api/[controller]")]
    public class SimulationController : ControllerBase
    {
        private readonly ISensorDataRepository _repository;
        private readonly HttpClient _httpClient;

        public SimulationController(ISensorDataRepository repository, HttpClient httpClient)
        {
            _repository = repository;
            _httpClient = httpClient;
        }

        [HttpPost("start")]
        public async Task<IActionResult> StartSimulation()
        {
            // Simulate data creation and insertion into the database
            await SimulateSensorData();
            await SimulateFacialRecognition();

            return Ok(new { message = "Simulation started successfully" });
        }

        private async Task SimulateSensorData()
        {
            // Simulate generating sensor data
            var random = new Random();

            for (int i = 0; i < 10; i++) // Simulating 10 sensor data entries
            {
                var sensorData = new SensorData
                {
                    SensorType = "Camera",
                    Value = random.Next(0, 2) == 0 ? "Person detected" : "No person detected",
                    Timestamp = DateTime.UtcNow.AddSeconds(-i * 10) // Generating timestamps in the past
                };

                await _repository.AddSensorDataAsync(sensorData);
            }
        }

        private async Task SimulateFacialRecognition()
        {
            for (int i = 0; i < 10; i++)
            {
                // Simulating image data
                var imageData = new ImageData
                {
                    Image = GenerateRandomImage()
                };

                var content = new StringContent(JsonConvert.SerializeObject(new
                {
                    instances = new[] { imageData }
                }), Encoding.UTF8, "application/json");

                var response = await
_httpClient.PostAsync("http://localhost:8501/v1/models/face_recognition_model:predict", content);
                var jsonResponse = await response.Content.ReadAsStringAsync();

                var result = JsonConvert.DeserializeObject<FacialRecognitionResult>(jsonResponse);

                // Save the prediction result to the database
                var sensorData = new SensorData
                {
                    SensorType = "FacialRecognition",
                    Value = $"Person detected: {result?.PredictedLabel}",
                    Timestamp = DateTime.UtcNow
                };

                await _repository.AddSensorDataAsync(sensorData);
            }
        }

        private float[] GenerateRandomImage()
        {
            // Simulate generating a random image as a flat array of pixel values
            var random = new Random();
            var image = new float[224 * 224 * 3]; // Example for a 224x224 RGB image

            for (int i = 0; i < image.Length; i++)
            {
                image[i] = (float)random.NextDouble();
            }

            return image;
        }
    }

    public class FacialRecognitionResult
    {
        public string PredictedLabel { get; set; }
    }
}
```

**Startup/Startup.cs:**

```
public class Startup
{
   public void ConfigureServices(IServiceCollection services)
   {
      services.Configure<MongoSettings>(options =>
      {
         options.ConnectionString =
Configuration.GetSection("MongoConnection:ConnectionString").Value;
         options.Database = Configuration.GetSection("MongoConnection:Database").Value;
      });
      services.AddSingleton<DataContext>();
      services.AddScoped<ISensorDataRepository, SensorDataRepository>();
      services.AddHttpClient();
      services.AddControllers();
   }

   public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
   {
      if (env.IsDevelopment())
      {
         app.UseDeveloperExceptionPage();
      }

      app.UseRouting();
      app.UseEndpoints(endpoints =>
      {
         endpoints.MapControllers();
      });
   }
}
```

## Frontend with React and TypeScript

**Setting Up the Project:**

```
npx create-react-app drone-surveillance-dashboard --template typescript

cd drone-surveillance-dashboard
```

**components/SensorDataList.tsx:**

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';

interface SensorData {
  id: string;
  sensorType: string;
  value: string;
  timestamp: string;
}

const SensorDataList: React.FC = () => {
  const [sensorData, setSensorData] = useState<SensorData[]>([]);

  useEffect(() => {
    const fetchData = async () => {
      const result = await axios.get<SensorData[]>('http://localhost:5000/api/sensordata');
      setSensorData(result.data);
    };

    fetchData();
  }, []);

  return (
    <div>
      <h1>Sensor Data</h1>
      <ul>
        {sensorData.map(data => (
          <li key={data.id}>
            {data.sensorType}: {data.value} at {data.timestamp}
          </li>
        ))}
      </ul>
    </div>
  );
};
```

```
export default SensorDataList;
```

**components/ControlPanel.tsx:**

```
import React from 'react';
import axios from 'axios';

const ControlPanel: React.FC = () => {
  const startSimulation = async () => {
    await axios.post('http://localhost:5000/api/simulation/start');
  };

  return (
    <div>
      <h1>Control Panel</h1>
      <button onClick={startSimulation}>Start Simulation</button>
    </div>
  );
};

export default ControlPanel;
```

**components/FaceRecognitionResults.tsx:**

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';

interface FaceRecognitionResult {
  label: string;
  probability: number;
}

const FaceRecognitionResults: React.FC = () => {
  const [results, setResults] = useState<FaceRecognitionResult[]>([]);

  useEffect(() => {
    const fetchResults = async () => {
      const response = await
axios.post<FaceRecognitionResult[]>('http://localhost:5000/api/facerecognition/predict', {
        image: [/* Insert example image data here */]
      });
      setResults(response.data);
    };

    fetchResults();
  }, []);

  return (
    <div>
      <h1>Face Recognition Results</h1>
      <ul>
        {results.map((result, index) => (
          <li key={index}>
            Label: {result.label}, Probability: {result.probability}
          </li>
        ))}
      </ul>
    </div>
  );
};
export default FaceRecognitionResults;
```

**App.tsx:**

```
import React from 'react';
import SensorDataList from './components/SensorDataList';
import ControlPanel from './components/ControlPanel';
import FaceRecognitionResults from './components/FaceRecognitionResults';

const App: React.FC = () => {
  return (
    <div>
      <ControlPanel />
      <SensorDataList />
      <FaceRecognitionResults />
    </div>
  );
};

export default App;
```

## AI Model Training and Serving with TensorFlow

**Python Script for Training the Facial Recognition Model:**

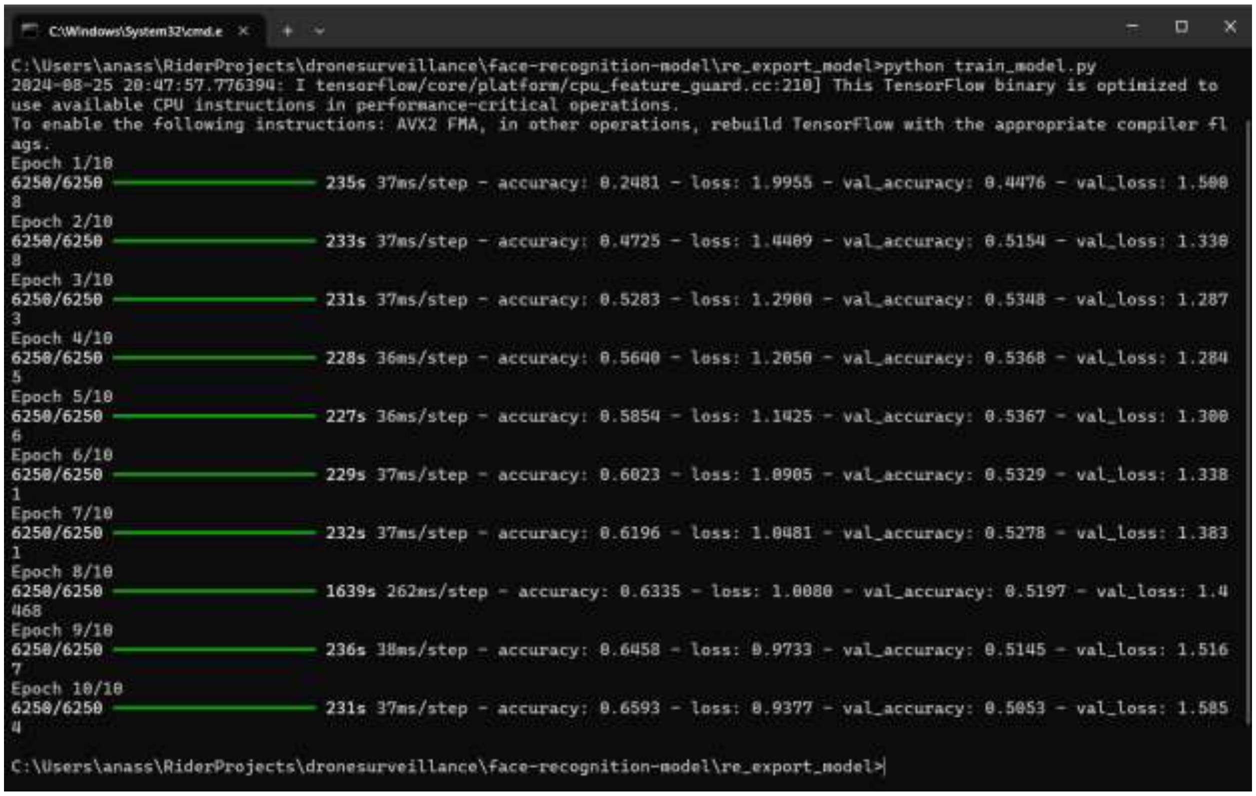
```
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Data generators
train_datagen = ImageDataGenerator(rescale=1./255, horizontal_flip=True, zoom_range=0.2)
val_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
    'data/train',
    target_size=(160, 160),
    batch_size=32,
    class_mode='categorical'
)

val_generator = val_datagen.flow_from_directory(
    'data/val',
    target_size=(160, 160),
    batch_size=32,
    class_mode='categorical'
)

from tensorflow.keras.applications import ResNet50
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D
from tensorflow.keras.models import Model

base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(160, 160, 3))
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(1024, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

model = Model(inputs=base_model.input, outputs=predictions)

for layer in base_model.layers:
    layer.trainable = False

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

model.fit(train_generator, validation_data=val_generator, epochs=10)

model.save('face_recognition_model.h5')
```

**TensorFlow Serving Container Setup:**

docker pull tensorflow/serving

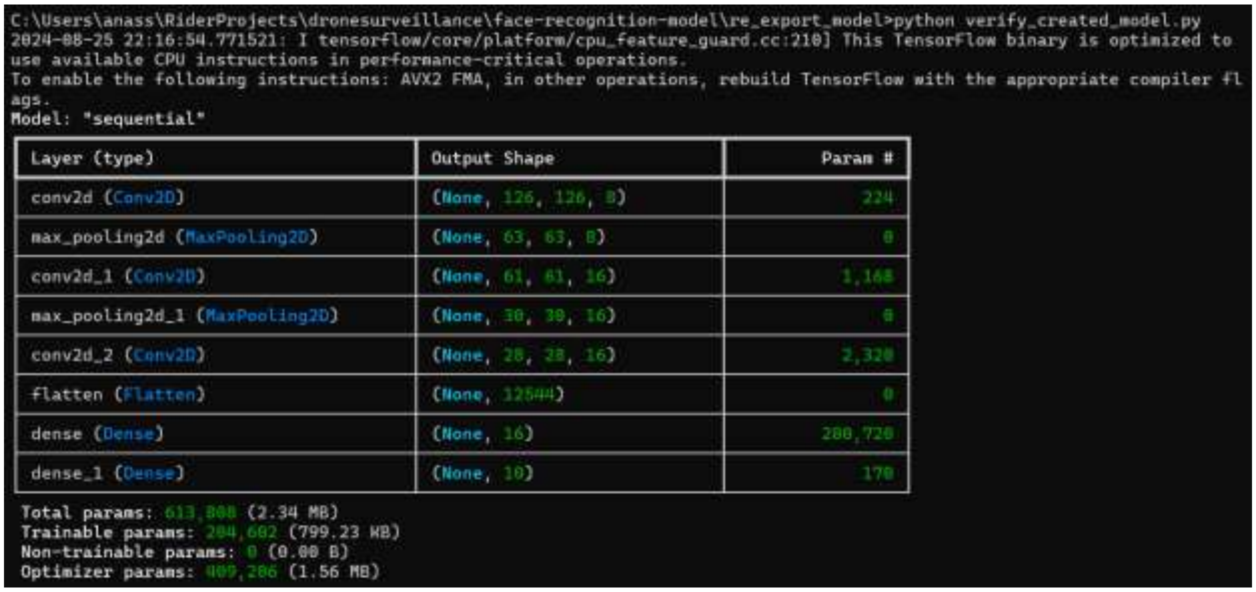
docker run -p 8501:8501 --name=tf_serving_face_recognition --mount type=bind,source=$(pwd)/model,target=/models/face_recognition_model -e MODEL_NAME=face_recognition_model -t tensorflow/serving

**Output**



```
C:\Users\anass\RiderProjects\dronesurveillance\face-recognition-model\re_export_model>python train_model.py
2024-08-25 20:47:57.776394: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to
use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler fl
ags.
Epoch 1/10
6250/6250 ━━━━━━━━━━━━━━━━━━━━ 235s 37ms/step - accuracy: 0.2481 - loss: 1.9955 - val_accuracy: 0.4476 - val_loss: 1.500
8
Epoch 2/10
6250/6250 ━━━━━━━━━━━━━━━━━━━━ 233s 37ms/step - accuracy: 0.4725 - loss: 1.4409 - val_accuracy: 0.5154 - val_loss: 1.330
8
Epoch 3/10
6250/6250 ━━━━━━━━━━━━━━━━━━━━ 231s 37ms/step - accuracy: 0.5283 - loss: 1.2900 - val_accuracy: 0.5348 - val_loss: 1.287
3
Epoch 4/10
6250/6250 ━━━━━━━━━━━━━━━━━━━━ 228s 36ms/step - accuracy: 0.5640 - loss: 1.2050 - val_accuracy: 0.5368 - val_loss: 1.284
5
Epoch 5/10
6250/6250 ━━━━━━━━━━━━━━━━━━━━ 227s 36ms/step - accuracy: 0.5854 - loss: 1.1425 - val_accuracy: 0.5367 - val_loss: 1.300
6
Epoch 6/10
6250/6250 ━━━━━━━━━━━━━━━━━━━━ 229s 37ms/step - accuracy: 0.6023 - loss: 1.0905 - val_accuracy: 0.5329 - val_loss: 1.338
1
Epoch 7/10
6250/6250 ━━━━━━━━━━━━━━━━━━━━ 232s 37ms/step - accuracy: 0.6196 - loss: 1.0481 - val_accuracy: 0.5278 - val_loss: 1.383
1
Epoch 8/10
6250/6250 ━━━━━━━━━━━━━━━━━━━━ 1639s 262ms/step - accuracy: 0.6335 - loss: 1.0080 - val_accuracy: 0.5197 - val_loss: 1.4
468
Epoch 9/10
6250/6250 ━━━━━━━━━━━━━━━━━━━━ 236s 38ms/step - accuracy: 0.6458 - loss: 0.9733 - val_accuracy: 0.5145 - val_loss: 1.516
7
Epoch 10/10
6250/6250 ━━━━━━━━━━━━━━━━━━━━ 231s 37ms/step - accuracy: 0.6593 - loss: 0.9377 - val_accuracy: 0.5053 - val_loss: 1.585
4

C:\Users\anass\RiderProjects\dronesurveillance\face-recognition-model\re_export_model>
```

**Summary**



```
C:\Users\anass\RiderProjects\dronesurveillance\face-recognition-model\re_export_model>python verify_created_model.py
2024-08-25 22:16:54.771521: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to
use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler fl
ags.
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ conv2d (Conv2D)                      │ (None, 126, 126, 8)         │             224 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d (MaxPooling2D)         │ (None, 63, 63, 8)           │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_1 (Conv2D)                    │ (None, 61, 61, 16)          │           1,168 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d_1 (MaxPooling2D)       │ (None, 30, 30, 16)          │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_2 (Conv2D)                    │ (None, 28, 28, 16)          │           2,320 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ flatten (Flatten)                    │ (None, 12544)               │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense (Dense)                        │ (None, 16)                  │         200,720 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_1 (Dense)                      │ (None, 10)                  │             170 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘
 Total params: 613,808 (2.34 MB)
 Trainable params: 204,602 (799.23 KB)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 409,206 (1.56 MB)
```

**Verified Saved Model Output**

saved_model_cli show --dir C:/Users/anass/RiderProjects/dronesurveillance/face-recognition-model/reexported_model --all

```
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

signature_def['__saved_model_init_op']:
  The given SavedModel SignatureDef contains the following input(s):
  The given SavedModel SignatureDef contains the following output(s):
    outputs['__saved_model_init_op'] tensor_info:
        dtype: DT_INVALID
        shape: unknown_rank
        name: NoOp
  Method name is:

signature_def['serving_default']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['inputs'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 128, 128, 3)
        name: serving_default_inputs:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['output_0'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 10)
        name: StatefulPartitionedCall:0
  Method name is: tensorflow/serving/predict
```

```
The MetaGraph with tag set ['serve'] contains the following ops: {'VarIsInitializedOp', 'Const', 'MatMul',
'AddV2', 'DisableCopyOnRead', 'StringJoin', 'Identity', 'Reshape', 'Select', 'NoOp', 'Pack',
'StatefulPartitionedCall', 'Relu', 'Placeholder', 'SaveV2', 'MergeV2Checkpoints', 'Conv2D', 'VarHandleOp',
'ShardedFilename', 'ReadVariableOp', 'StaticRegexFullMatch', 'Softmax', 'MaxPool', 'RestoreV2',
'AssignVariableOp'}

Concrete Functions:2024-08-25 23:24:06.768812: I tensorflow/core/platform/cpu_feature_guard.cc:210]
This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the
appropriate compiler flags.

  Function Name: '_default_save_signature'
    Option #1
      Callable with:
        Argument #1
          inputs: TensorSpec(shape=(None, 128, 128, 3), dtype=tf.float32, name='inputs')
```

## Steps to Ensure Correct API Request to TensorFlow Serving

**1. Verify Model Metadata**

Based on the model metadata you retrieved, the model expects the following inputs:

- **Input Tensor Name:** serving_default_inputs:0
- **Output Tensor Name:** StatefulPartitionedCall:0

**2. Example curl Request**

Here's how you should format your JSON payload and curl request based on the model metadata:

**Example JSON Payload**

Ensure your JSON payload matches the expected tensor format:

```
{
  "instances": [
    {
      "serving_default_inputs:0": [[[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]], [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]]]
    }
  ]
}
```

**Example curl Command**

```
curl -X POST "http://localhost:8501/v1/models/face_recognition_model:predict" \
     -H "Content-Type: application/json" \
     -d '{
       "instances": [
         {
           "serving_default_inputs:0": [[[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]], [[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]]]
         }
       ]
     }'
```

**3. Verify API Implementation**

Ensure your API controller is correctly formatting and sending the request to TensorFlow Serving. Here is an example controller in C#:

```
using Microsoft.AspNetCore.Mvc;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;
using Newtonsoft.Json; // Ensure you have this namespace for JSON serialization

namespace YourNamespace.Controllers
{
  [ApiController]
  [Route("api/[controller]")]
  public class FaceRecognitionController : ControllerBase
  {
    private readonly HttpClient _httpClient;

    public FaceRecognitionController(HttpClient httpClient)
    {
      _httpClient = httpClient;
      _httpClient.BaseAddress = new Uri("http://localhost:8501"); // TensorFlow Serving URL
    }

    [HttpPost("predict")]
    public async Task<IActionResult> Predict([FromBody] ImageData imageData)
    {
      var requestContent = new
      {
        instances = new[]
        {
          new
          {
            serving_default_inputs_0 = imageData.ImageData
          }
        }
      };

      var jsonContent = new StringContent(JsonConvert.SerializeObject(requestContent),
Encoding.UTF8, "application/json");

      var response = await _httpClient.PostAsync("/v1/models/face_recognition_model:predict",
jsonContent);

      if (!response.IsSuccessStatusCode)
```

```
            {
                return StatusCode((int)response.StatusCode, await response.Content.ReadAsStringAsync());
            }

            var result = await response.Content.ReadAsStringAsync();
            return Ok(result);
        }
    }

    public class ImageData
    {
        public float[][][][] ImageData { get; set; } // Match the shape of your model input
    }
}
```

## Convert Model with Docker

**Set Up the Docker Environment**

- Ensure you have a working Docker installation.
- Create a directory on your host machine that contains:
  - The .keras model file.
  - The Python script for conversion (convert_to_saved_model.py).
  - A Dockerfile.

**2. Prepare the Dockerfile**

- Create a Dockerfile in your project directory with the following content:

```
# Use the official TensorFlow image as the base image
FROM tensorflow/tensorflow:latest
# Create a working directory
WORKDIR /app
# Copy the .keras model file and the conversion script to the working directory
COPY model.keras /app/model.keras
COPY convert_to_saved_model.py /app/convert_to_saved_model.py

# Run the conversion script
CMD ["python", "convert_to_saved_model.py"]
```

**3. Write the Conversion Script**

- Create a Python script named convert_to_saved_model.py with the following content:

```
import tensorflow as tf
# Define the path to the .keras model
keras_model_path = '/app/model.keras'
saved_model_path = '/app/saved_model'

# Load the .keras model
model = tf.keras.models.load_model(keras_model_path)

# Save the model in the SavedModel format
tf.saved_model.save(model, saved_model_path)

print(f"Model saved to: {saved_model_path}")
```

**4. Build the Docker Image**

- In the directory containing the Dockerfile, run the following command to build the Docker image:

```
docker build -t keras-to-savedmodel .
```

**5. Run the Docker Container**

- Run the Docker container while mounting a directory on your host machine to store the output SavedModel:

```
docker run --rm -v C:/Users/anass/RiderProjects/dronesurveillance/face-recognition-model/model_converter:/app/saved_model keras-to-savedmodel
```

- Make sure to adjust the path according to your environment.

**6. Check the Output**

- After the container runs successfully, check the output directory (model_converter) on your host machine for the saved_model directory, which contains the SavedModel files, including saved_model.pb.

**7. Use the SavedModel**

- You can now use the SavedModel format in TensorFlow Serving or any other deployment environment that supports TensorFlow models.

**8. Clean Up**

- If necessary, remove the Docker container and image once you're done.

```
docker rmi keras-to-savedmodel
```

## Simulation Unit (Python Example)

```
import time
import random
import requests

API_URL = "http://localhost:5000/api/sensordata"

def generate_sensor_data():
    sensors = ["Camera", "Infrared", "GPS"]
    while True:
        data = {
            "sensorType": random.choice(sensors),
            "value": str(random.uniform(10.0, 100.0)),
            "timestamp": time.strftime("%Y-%m-%d %H:%M:%S")
        }
        response = requests.post(API_URL, json=data)
        if response.status_code == 200:
            print("Data sent successfully")
        else:
            print("Failed to send data")
        time.sleep(1)

if __name__ == "__main__":
    generate_sensor_data()
```

## Summary of the Complete Solution

1. **Backend API with ASP.NET Core and MongoDB**: The API receives sensor data, processes it, and uses TensorFlow Serving for facial recognition.
2. **Frontend with React and TypeScript**: The dashboard displays sensor data and facial recognition results and provides a control unit.
3. **AI Model**: The model is trained in Python and served using TensorFlow Serving.
4. **Simulation Unit**: Continuously generates sensor data to emulate a drone.

This comprehensive solution covers all aspects of developing an AI-powered facial recognition and target tracking system, integrating backend, frontend, AI models, and simulation units.

## Troubleshooting

### TensorFlow Serving container issue

**1. Verify Model Structure Inside the Container**

1. **Access the Running Container:**

```
docker exec -it tf_model_serving /bin/bash
```

2. **Navigate to the Model Path Inside the Container:**

```
cd /models/face_recognition_model
```

3. **Check for the Model Directory Structure:**

```
ls -l
```

You should see a directory for the model version, e.g., 1.

4. **Go into the Version Directory and Check for saved_model.pb:**

```
cd 1
ls -l
```

You should see saved_model.pb along with other files and directories.

**2. Verify the Docker Run Command**

Ensure the Docker run command maps the host directory correctly to the container:

Your Docker run command:

```
docker run -p 8501:8501 --name=tf_model_serving -v
C:/Users/anass/RiderProjects/dronesurveillanceapi/face-recognition-
model/models/face_recognition_model:/models/face_recognition_model -e
MODEL_NAME=face_recognition_model tensorflow/serving:latest
```

**Check the Following:**

- The **host path** (C:/Users/anass/RiderProjects/dronesurveillanceapi/face-recognition-model/models/face_recognition_model) must be correct and should contain the model directory in the specified structure.
- The **container path** (/models/face_recognition_model) must match the path specified in the -v option.

**3. Recreate the Container**

If you have made changes to the directory structure, stop and remove the container, and start it again:

```
docker stop tf_model_serving
```

```
docker rm tf_model_serving
```

Then run the Docker command again:

```
docker run -p 8501:8501 --name=tf_model_serving -v
C:/Users/anass/RiderProjects/dronesurveillanceapi/face-recognition-
model/models/face_recognition_model:/models/face_recognition_model -e
MODEL_NAME=face_recognition_model tensorflow/serving:latest
```

**4. Verify the Model Export**

Ensure your model is correctly saved and exported in TensorFlow. The directory structure should look like this:

```
/models/face_recognition_model
  └── 1
      ├── saved_model.pb
      ├── fingerprint.pb
      ├── assets
     └── variables
```

**5. Check Logs for Errors**

If the container still does not start correctly, check the logs for additional information:

```
docker logs tf_model_serving
```

## Deal with big files more as 100MB in git system

Don't push big files like the model if that bigger as 100MB and use the .Ignore.txt file for ignoring files and folders.
for example:

```
# Ignore node_modules in any directory

**/node_modules/


# Ignore tf_env in any directory

**/tf_env/
```

# Ignore the specific file 1.keras

1.keras

With the following command in the powershell window you can get all first 10 big files in your directory.

**Get-ChildItem -Recurse | Sort-Object Length -Descending | Select-Object FullName, Length -First 10**

**git reflog expire --expire=now --all**: This command removes reflog entries that are older than "now," effectively cleaning up all entries.

**git gc --prune=now --aggressive**: This command performs garbage collection on your repository, removing any unnecessary objects and optimizing the repository.

## Tensorflow/serving

Ensure that the tensorflow/serving is available.
curl -X GET http://localhost:8501/v1/models/face_recognition_model

The right answer must be in this structure:

```
{
  "model_version_status": [
    {
      "version": "1",
      "state": "AVAILABLE",
      "status": {
        "error_code": "OK",
        "error_message": ""
      }
    }
  ]
}
```

**Ensure the input data matches the model requirements.** Check the format and dimensions.

**Validate JSON structure.** Confirm that the JSON structure aligns with TensorFlow Serving expectations.

**Check error messages and logs.** Analyze the error messages and logs for detailed hints about what might be going wrong.