Report for Project Image Classifier

Anshif
Soul AI Assessment Task

March 13, 2025

1 Data preprocessing

1.1 Image Resizing

Images are resized to 150x150 pixels to ensure the consistency with input dimensions for the model

1.2 Dataset Splitting

Training set is split 20% into validation set. Intel_image_dataset provides a separate test set.

1.3 Data Augmentation

Applied to training data to increase diversity and reduce overfitting with:

- Rotation (up to 20 degrees)
- Width and height shifts (up to 20%)
- Shear transformations (up to 20%)
- Zoom variations (up to 20%)
- Horizontal flips
- Nearest-neighbor fill mode for empty pixels

The jupyter code is designed to output some example augmented images.

Figure 1: Augmented Images

1.4 Exploratory Data Analysis (EDA)

Analysis of class distribution and image sample visualisation

1.5 Data Generators and Batch Processing

Data is processed in batches (BATCH_SIZE = 32) to efficiently handle large datasets within memory constraints. Generators apply prepossesses and generate images in real time to optimize memory usage.

2 Model Training & Evaluation

2.1 Architecture Options

The code allows choosing from four model architectures viz MobileNetV2 (default), ResNet50, EfficientNetB0, Custom CNN. The model used for deployment is trained using MobileNet.

2.2 Model Structure

- Base Model Customization: Pre-trained networks with custom classification head
- Feature Extraction: GlobalAveragePooling2D for spatial dimension reduction
- Regularization: Strategic Dropout layers (0.2-0.5 rates) to prevent overfitting
- Dense Layers: 256-512 neuron fully connected layers for feature learning
- Output Layer: Softmax activation for multi-class probability distribution

2.3 Training Techniques

- Early Stopping: Prevents overfitting by monitoring validation loss (patience=5)
- Learning Rate Scheduling: ReduceLROnPlateau adjusts learning rate when performance plateaus
- Model Checkpointing: Saves best performing model based on validation accuracy Two-Phase Training: Initial training followed by optional fine-tuning phase

2.4 Evaluation Features

2.4.1 Performance Metrics

- Accuracy
- Precision
- Recall F1-Score

2.4.2 Visualization

- Confusion Matrix
- Training/Validation Curves: Monitors learning progress and potential overfitting
- Per-Class Metrics: Visualizes precision, recall, and F1-score for each class

2.5 Optimization Approach

- Optimizer: Adam optimizer
- Learning Rate: 0.001 (initial), reduced during training as needed
- Loss Function: Categorical cross-entropy (standard for multi-class classification)
- Metric: Accuracy as primary performance metric

3 Deploy Model Using FastAPI

The backend system for the image classification service was implemented using FastAPI.

3.1 FastAPI Backend Architecture

```
image-classifier-api/
 app/
    __init__.py     # Application initialization and logging setup
    main.py         # API endpoints, middleware, and request handling
    auth.py         # Authentication logic
    model.py        # Image classifier implementation and inference
    utils.py        # Helper functions and custom utilities
 model/              # Directory storing trained models
 logs/               # Log file storage
 Dockerfile          # Container configuration
 requirements.txt    # Dependencies
```

3.2 Core API Implementation

The API exposes three primary endpoints:

- `/`: Root endpoint providing basic health check
- `/health`: Detailed health status including model readiness
- `/predict`: Main endpoint for image classification

The `/predict` endpoint accepts image uploads, processes them through the classifier model, and returns prediction results with confidence scores. The endpoint implementation includes comprehensive validation, error handling, and authentication to ensure reliability and security.

3.3 Application Lifecycle Management

A key architectural feature is the implementation of asynchronous lifespan management using FastAPI's `asynccontextmanager`. This pattern guarantees that the model is loaded during application startup and all resources are properly released during shutdown, preventing memory leaks and ensuring system stability.

3.4 Logging and Monitoring

A comprehensive logging system was implemented to provide visibility into application behavior:

The logging system captures:

- Request metadata (method, path, timing)
- Performance metrics (processing duration)
- Error information (exceptions with stack traces)
- Model operations (loading, prediction results)

Logs are stored in date-formatted files in the `logs/` directory.

3.5 Error Handling and Resilience

The system implements a multi-layered approach to error handling:

1. **Custom Exception Handlers**: Dedicated handlers for HTTP and general exceptions
2. **Request Tracing**: Unique IDs assigned to each request for error correlation
3. **Graceful Degradation**: Appropriate status codes and informative messages when services are unavailable
4. **Descriptive Error Responses**: Clear error details with request context

y and user experience.

3.6 Security Implementation

Security is implemented through HTTP Basic Authentication.

3.7 Deployment Considerations

The application is containerized using Docker, with all dependencies clearly specified in `requirements.txt`.

3.8 Deployment

The API is hosted and publicly accessible at: `https://image-classifier-rngj.onrender.com`

4 Frontend Using Streamlit

4.1 Frontend Implementation

The frontend of our Landscape Classifier application is built using Streamlit.

4.2 Project Structure

```
streamlit-app/
 app.py                   # Main Streamlit application
 .env                     # Environment variables
 .streamlit/              # Streamlit configuration
    config.toml          # Streamlit config settings
    requirements.txt     # Dependencies
 Dockerfile-streamlit     # Docker configuration
```

4.3 Main Application (app.py)

The `app.py` file serves as the core of our Streamlit application. It implements several key features:

- **Modern UI/UX Design**: Custom CSS styling with gradient headers, animated buttons, and responsive layout
- **Image Upload**: File uploader for JPG, JPEG, and PNG formats with preview functionality
- **API Integration**: Secure communication with the backend classification API
- **Results Visualization**: Interactive charts showing classification confidence scores
- **System Status**: Real-time API health monitoring with visual indicators
- **Responsive Layout**: Two-column design that adapts to different screen sizes

The application follows a modular structure with separate functions for:

- API health checking with caching
- Image classification request handling
- Results visualization with Plotly charts
- Temporary file management
- Category and metadata display

4.4 Environment Configuration

The `.env` file contains essential configuration variables:

- API URL endpoints
- Authentication credentials

4.5 Containerization

The `Dockerfile-streamlit` enables containerization of the frontend, ensuring consistent deployment across environments.

4.6 Deployment

The application is deployed and publicly accessible at: `https://image-classifier-tnb86bthtu2vwr streamlit.app/`

This deployment leverages Streamlit Cloud for hosting, providing:

- Automatic scaling
- SSL encryption
- High availability
- Continuous deployment from our GitHub repository

The frontend communicates with our backend API service, which hosts the trained DL model for landscape classification.

5 Optional(Bonus Point) Features Implemented

- Implement logging and error handling in the API.
- Deploy the API on AWS, GCP, Azure, or Render.
- Implement a lightweight frontend using Streamlit for image classification.