Machine Learning Project Documentation: Deployment

Group Members:

- Abenezer Tesfaye
- Shewanek Zewdu
- Emran Kamil
- Hewan
- Biniyam Berga

1. Overview

The deployment phase for this project involves making the trained plant disease detection model accessible to users through a web application. The model, which is capable of classifying images of plants to detect diseases, is deployed locally using Streamlit, a Python library for creating interactive web applications for machine learning and data science. Users can upload an image of a plant, and the application will display the predicted disease class. This approach allows for a user-friendly interface where individuals can directly interact with the model without needing to run code or understand the underlying complexities. The Streamlit application handles image preprocessing, model prediction, and result display.

2. Model Serialization

The trained machine learning model is serialized to enable its storage and subsequent loading for deployment. In this project:

- **Serialization Format:** The model is saved in the HDF5 format, indicated by the `.h5` file extension (`disease_detector_v1.h5`). Keras (part of TensorFlow) provides functionality to save and load models in this format, which stores the model architecture, weights, and training configuration (including the optimizer state).
- **Process:** The line `model = tf.keras.models.load_model(model_path)` demonstrates the loading of the pre-serialized model. The serialization would have occurred after the model training was complete, using a command like `model.save(model_path)`.
- **Considerations for Efficient Storage:** HDF5 is generally efficient for storing large numerical data, such as the weights of a neural network. While not explicitly detailed in the provided deployment script, considerations during the initial saving of the model might have included whether to save the full optimizer state (useful for resuming

training but larger) or just the weights (smaller, sufficient for inference). For this deployment, loading the full model is appropriate.

3. Model Serving

The serialized model is served for making predictions through the Streamlit web application.

- **Deployment Platform:** The chosen deployment platform is Streamlit. Streamlit runs a local web server that hosts the application. When the user interacts with the application (e.g., uploads an image and clicks "Classify"), Streamlit executes the Python script on the server-side.
- **Serving Mechanism:**
    1. The application starts, and the serialized Keras model (disease_detector_v1.h5) is loaded into memory using tf.keras.models.load_model().
    2. The class_indices.json file, which maps prediction indices to human-readable class names, is also loaded.
    3. When a user uploads an image, it is processed by the load_and_preprocess_image function.
    4. The predict_image_class function then calls model.predict() on the preprocessed image. This is where the loaded model performs inference.
    5. The prediction result is returned to the Streamlit interface and displayed to the user.
- **Solutions:** This is an on-premises solution in the sense that the Streamlit application and the model run on the machine where the script is executed. To make it accessible more broadly, it would need to be deployed on a server with a public IP address or hosted on a cloud platform that supports Streamlit applications (e.g., Streamlit Sharing, Heroku, AWS, GCP, Azure).

4. API Integration

In this specific deployment, the model is not integrated via a separate REST API (like Flask or FastAPI) in the traditional sense. Instead, Streamlit provides the user interface and handles the interaction with the model directly.

- **API Endpoints:** There are no explicit HTTP API endpoints (e.g., /predict) defined in the code for other services to call.
- **Input Format:** The input is an image file (JPG, JPEG, PNG) uploaded by the user through the Streamlit file uploader widget (st.file_uploader). The load_and_preprocess_image

function then converts this uploaded file into the numerical array format (shape (1, 224, 224, 3), scaled to [0, 1]) expected by the model.

- **Response Format:** The response is a string representing the predicted class name, which is displayed directly in the web application's UI using st.success(f'Prediction: {str(prediction)}').

If a dedicated API were required (e.g., for programmatic access by other applications), one would typically wrap the model prediction logic in a web framework like Flask or FastAPI, define an endpoint, and specify JSON or other formats for input (e.g., image as base64 string or URL) and output (e.g., JSON with prediction and confidence).

5. Security Considerations

The provided Streamlit script does not explicitly detail advanced security measures. For a locally run Streamlit application, security considerations might include:

- **Local Network Security:** If the application is only intended for use on a local machine or a trusted internal network, the primary security might rely on the security of that network.
- **File Uploads:** Streamlit handles file uploads. While it restricts file types (type=["jpg", "jpeg", "png"]), in a production environment with broader access, further validation and sanitization of uploaded files would be crucial to prevent security vulnerabilities (e.g., uploading malicious files disguised as images).
- **Dependencies:** Ensuring all Python libraries (Streamlit, TensorFlow, Pillow, NumPy) are kept up-to-date is important to patch known vulnerabilities.

If this application were to be deployed to a public-facing server, more robust security measures would be essential:

- **Authentication & Authorization:** Implementing mechanisms to control who can access the application and what actions they can perform.
- **HTTPS:** Using HTTPS to encrypt data in transit between the user's browser and the server.
- **Input Validation:** More rigorous validation of all inputs.
- **Resource Limits:** Setting limits on file sizes, request rates, etc., to prevent abuse.
- **Secrets Management:** If the application used API keys or database credentials, these would need to be stored securely, not hardcoded.

For the current script, the security context is primarily that of a local development or demonstration tool.

6. Monitoring and Logging

Mechanisms for monitoring the deployed model's performance or for detailed logging beyond what Streamlit or a hosting platform might offer by default.

- **Metrics Tracked:** No specific performance metrics (e.g., prediction latency, error rates over time, drift in input data distribution) are being actively tracked within the script.
- **Logging:**
    - Streamlit applications output logs to the console where they are run. This typically includes information about server startup, user connections, and any errors or exceptions encountered during script execution.
    - TensorFlow might also produce its own logs or warnings.
- **Alerting Mechanisms:** There are no alerting mechanisms defined in the script to notify administrators of issues (e.g., high error rates, application crashes).

For a production deployment, a more comprehensive monitoring and logging strategy would be necessary:

- **Performance Monitoring:** Logging prediction times, CPU/memory usage.
- **Prediction Logging:** Logging input features (or their characteristics) and model predictions to enable later analysis, model retraining, and debugging.
- **Error Tracking:** Using tools (e.g., Sentry, LogRocket) to capture and aggregate errors.
- **Dashboarding:** Visualizing key metrics on dashboards (e.g., using Grafana, Kibana, or cloud provider tools).
- **Alerts:** Setting up alerts for critical issues like application downtime, spikes in errors, or significant drops in model performance.
- **Model Drift Detection:** Implementing checks to detect if the statistical properties of the input data or the relationship between inputs and outputs have changed over time, which might indicate the model needs retraining.

In its current form, monitoring relies on observing the Streamlit console output and manually assessing the application's behavior.