Machine Learning Project Documentation

Deployment

Cervical Cancer Detection Documentation

1. Overview

The deployment phase of the model is a representation of the transition from the development usages of the cervical cancer lesion detection model to real-world usability. This phase will afford the user an avenue to upload medical images for some detailed predictions-inclusive visualizations and metrics that can be integrated seamlessly into the user's workflow. A hybrid approach is considered for this work in model deployment: hosting the model on a FastAPI server and connecting it to a Django backend that provides an API for easy access. The model was deployed on Hugging Face Spaces, ensuring scalability and accessibility.

2. Model Serialization

The trained model was serialized in PyTorch application format with '.pth' extension for appropriate compatibility and efficient storage as follows:

Serialization Process: Weights of the model along with architecture configuration were saved by using the torch.save() function.
Ensured the compatibility of the training and deployment environments with respect to Python and PyTorch versions.
Optimization: The serialized file was kept compact for saving on storage space and enabling faster loading during inference.
Reproducibility: The configuration file, mask_rcnn_config.yaml, was included to guarantee seamless model initialization.

3. Model Serving

The serialized model is served through a FastAPI application that takes an input image for processing and returns predictions. The key aspects include:

Platform: The server is deployed on Hugging Face Spaces with high availability and low latency for a FastAPI server.

Process: The server starts, followed by loading the model weights using DefaultPredictor from Detectron2. Then, medical images uploaded by the user are fed into the model for inference. The server returns bounding boxes, masks, class labels, confidence scores, and visualizations.

Communication: Django backend will talk to the FastAPI server by sending bytes of raw images for inference and process the response back for client consumption.

4. API Integration

The Django backend acts as a proxy, exposing a simple API interface and offloading model inference to the FastAPI server.

Endpoint

/predict/ - Django: This is the endpoint for client requests.

/predict - FastAPI: This handles inference requests.

Request Format

At the client side, send a POST request with an image file uploaded as form-data under the key 'file'.

Response Format

Visualization: A base64-encoded image displaying the predictions.

Metrics: JSON data containing:

Lesion class distribution

Detection scores

Bounding box areas.

This integration ensures that all data preprocessing and post-processing are performed in the Django backend, and model inferences are made on the FastAPI server.

5. Security Considerations

Security considerations put in place that ensured integrity and confidentiality of data included:

Authentication: Credentials or API keys are used to restrict access to the prediction endpoints.

Encryption: Communication from the client to the Django Backend and on to the FastAPI server is encrypted in HTTPS.

Input Validation: The back-end does ensure that the format of images is of a valid type before actually transmitting to the FastAPI server. Resource Isolation: Separation of deployment

regarding the backend and model server; hence, keeping possible breaches confined so that not much harm could be done.

6. Monitoring and Logging

Following metrics performance about the deployment has been defined for system reliability:

Performance Metrics:

Response Latency: To assure that the response time shall remain low.

Request Throughput: Counts the number of requests being processed; Model performance: Accuracy and consistency check of models deployed during inference.

Application Logs: Logs by Django backend and requests and responses logged by the FastAPI server for debugging purposes and auditing. Alerting: Set alerts on high latency or a great number of failed requests. Usage Analytics: It monitors how frequently APIs are called, the kind of input images, and what the results of the predictions were all to improve model performance with time.