Machine Learning Project Documentation

Deployment

1. Overview

The deployment phase focused on turning the trained snake ID models into portable, runtime-ready artifacts that can be executed consistently outside the training environment. The core steps were: exporting the fine-tuned CNN to ONNX with dynamic batch axes (so batch sizes can vary at inference time), optionally quantizing to INT8 for CPU/edge efficiency, and persisting class metadata (artifacts/class_map.json) to preserve label ordering. With these artifacts, the model can be dropped into a thin serving layer—HTTP API, batch job, or edge script—without reloading training code, enabling fast integration into real-world workflows.

2. Model Serialization

After training, the model is moved to CPU and exported to ONNX (artifacts/model_fp32.onnx) with named inputs/outputs and dynamic axes to maintain flexibility (new_training.py). This format is widely supported (Python, C++, mobile/edge runtimes) and reduces environment coupling. A companion INT8 export (model_int8.onnx) is attempted via ONNX Runtime dynamic quantization to shrink size and improve latency on CPUs. The class mapping file (class_map.json) travels with the ONNX file so consumers can convert logits to species names consistently. Storage considerations: keep both FP32 (highest fidelity) and INT8 (small/fast) versions; version artifacts by model hash or timestamp to support rollbacks.

3. Model Serving

Serving is designed around ONNX Runtime: load the ONNX graph, apply the same preprocessing as training (resize to 320, normalize with ImageNet stats), run inference, and return class scores. Platform choices remain flexible: a lightweight FastAPI/Uvicorn container for cloud/on-prem, a CLI/batch script for offline scoring, or ONNX Runtime Mobile for edge devices. Because the model is small (MobileNetV3-Small or EfficientNet-B0) and quantizable, it fits CPU-only deployments without GPU dependence. If higher throughput is needed, the same ONNX can be hosted on an inference server (e.g., Triton or ORT server) with autoscaling.

4. API Integration

A typical API wrapper (not yet implemented) would expose a /predict POST endpoint accepting an image file or base64 input. The server would load the ONNX model (model_fp32.onnx or model_int8.onnx), apply the same preprocessing as training, run inference, and return top-k species with probabilities in JSON. Class mappings shipped with the model allow the API to remain stateless aside from loading artifacts at startup.

5. Security Considerations

For an HTTP deployment, key controls include authentication (API keys or OAuth2 bearer tokens), authorization (role-based access to prediction endpoints), and transport security (TLS termination). Rate limiting and request size limits protect against abuse (e.g., excessively large uploads). If running quantized models on shared infrastructure, ensure artifacts are read-only and separated per environment. Logging should omit sensitive user data; store only minimal request metadata or hashed identifiers to support debugging without leaking images.

6. Monitoring and Logging

Operational monitoring would track request latency, throughput, error rates (4xx/5xx), and model-specific metrics like per-class confidence distributions to detect drift. Application logs should capture inference timings, model version IDs, and basic request metadata (size, format) without storing raw images. Alerting can be set on latency SLO violations, elevated error rates, or unusual confidence patterns (e.g., many low-confidence outputs). For periodic quality checks, batch a sample of predictions against a labeled canary set to compute accuracy or calibration metrics; if accuracy drops, trigger investigation or rollback to a prior ONNX version.