Machine Learning Project Documentation

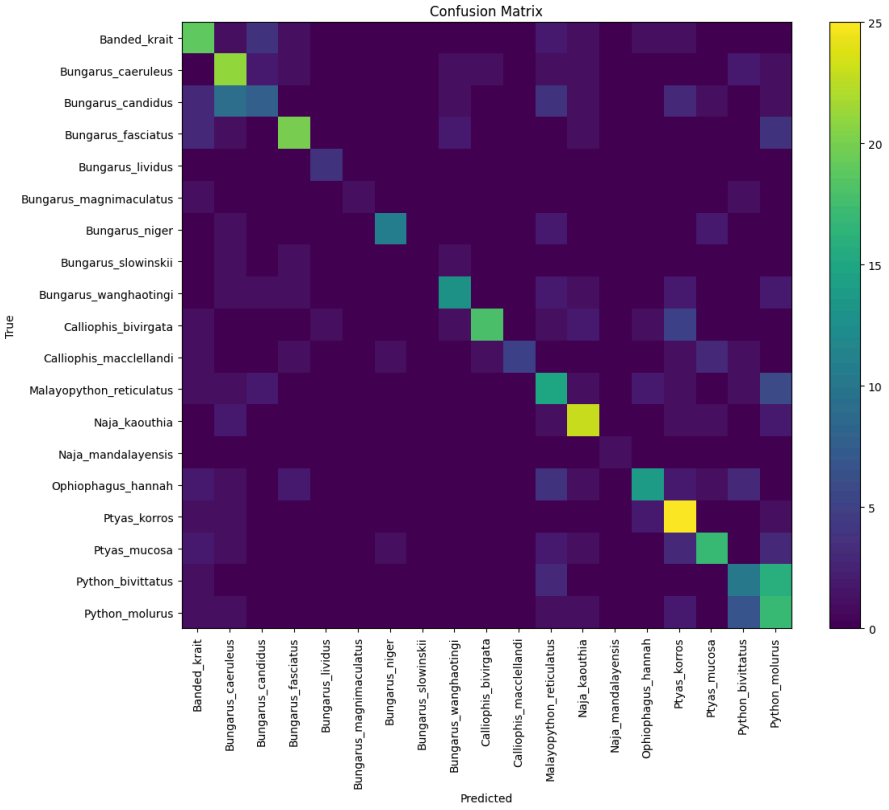
Model Refinement

1. Overview

After baseline transfer learning runs, refinement focused on squeezing more accuracy/robustness without losing the deployment-friendly footprint. The main levers were staged training (head → partial fine-tune), stronger augmentations, and comparing lightweight backbones (EfficientNet-B0 vs MobileNetV3-Small) to balance accuracy vs latency (new_training.py, train_cpu.py).

2. Model Evaluation

The model was evaluated on held-out validation and test sets using accuracy, precision, recall, and F1-score to address class imbalance. It achieved a test accuracy of 56.67%, indicating moderate performance. Species with more training samples showed higher precision and recall, while minority classes were poorly recognized or not predicted, leading to undefined precision for some classes. The macro F1-score (0.58) reflects uneven performance across classes, while the weighted F1-score (0.57) indicates bias toward majority classes. These results highlight the effects of data imbalance and visual similarity among snake species, suggesting that additional data and class-balanced training could improve performance.



3. Refinement Techniques

- Two-stage training: freeze backbone, train classifier head, then unfreeze the last feature block + classifier for targeted fine-tuning at a lower LR.
- Backbone comparison: EfficientNet-B0 for higher accuracy; MobileNetV3-Small for speed/size-sensitive deployments.
- Data/augmentation adjustments: kept ImageNet normalization, resize to 320, added random resized crop, flips, and mild color jitter to improve generalization; SafeImageFolder wrapper skips corrupt samples instead of crashing.

4. Hyperparameter Tuning

Detail any additional hyperparameter tuning performed during the refinement phase. Include insights gained and their impact on the model's performance.

- Learning rate schedule: head at 1e-3, fine-tune at 5e-4; epochs 8–10 with finetune_epochs=max(2, epochs//2).
- Batch size 32–64 tuned for throughput vs memory; num_workers increased (default 8) to speed I/O.
- Unfreezing scope limited to the last feature block to gain accuracy without overfitting or blowing up training time.
- No full grid/random search—manual, targeted tweaks guided by validation accuracy.

5. Cross-Validation

Discuss any changes made to the cross-validation strategy during model refinement and explain the reasoning behind those changes.

No k-fold CV was introduced; the approach stayed with a single hold-out validation (either explicit train/val folders or 80/20 split in train_cpu.py) to keep iteration time low on this image workload.

6. Feature Selection

Explicit feature selection techniques were not applied in this project. Instead, the model relies on **automatic feature extraction** through transfer learning using pre-trained convolutional neural networks (EfficientNet-B0/MobileNetV3-Small).

The pre-trained backbone networks extract hierarchical visual features such as edges, textures, patterns, and shapes directly from input images. During training, only the classification head was initially trained, followed by fine-tuning of the final feature extraction layers. This approach allows the model to adapt high-level features to the snake identification task without manual feature engineering.

Data augmentation techniques (random cropping, flipping, and color jittering) were applied to improve feature robustness and generalization, particularly for visually similar snake species.

Test Submission

1. Overview

The test submission phase evaluates the trained snake identification model on a held-out test dataset to assess its generalization performance. The final trained model checkpoint (`model_final.pt`) was loaded along with the corresponding class mapping generated during training. The evaluation pipeline applies consistent preprocessing and computes standard classification metrics to quantify model performance

2. Data Preparation for Testing

The test dataset was organized using a directory-based structure compatible with PyTorch's `ImageFolder` format. Images were resized and normalized using the same mean and standard deviation values applied during training and validation to ensure consistency.

To maintain label alignment with the trained model, the test dataset's class-to-index mapping was explicitly synchronized with the training class map. Test samples belonging to classes not present in the training set were excluded from evaluation to prevent label mismatch and invalid predictions.

3. Model Application

The trained MobileNetV3-Small model was loaded and set to evaluation mode. Test images were processed in batches using a DataLoader, and predictions were generated by selecting the class with the highest output score.

```
outputs = model(images)
preds = torch.argmax(outputs, dim=1)
```

All predictions and ground-truth labels were collected for subsequent performance analysis.

4. Test Metrics

Model performance on the test dataset was evaluated using **accuracy, precision, recall, F1-score, and a confusion matrix**. The model achieved a **test accuracy of 56.67%**, indicating moderate classification performance across 19 snake species.

Macro-averaged metrics were used to assess performance across all classes equally, revealing reduced effectiveness on minority classes. Weighted-averaged metrics were closer to overall accuracy, indicating better performance on majority classes. The confusion matrix further illustrates common misclassifications between visually similar snake species.

Compared to training and validation results, the test performance shows a generalization gap, highlighting the impact of class imbalance and the challenges of fine-grained snake species identification.

```
Test Accuracy: 0.5667

Classification Report:
                          precision    recall  f1-score   support

           Banded_krait       0.53      0.63      0.58        30
      Bungarus_caeruleus       0.50      0.68      0.58        31
       Bungarus_candidus       0.47      0.26      0.33        31
      Bungarus_fasciatus       0.74      0.65      0.69        31
        Bungarus_lividus       0.80      1.00      0.89         4
 Bungarus_magnimaculatus       1.00      0.33      0.50         3
         Bungarus_niger       0.85      0.69      0.76        16
     Bungarus_slowinskii       0.00      0.00      0.00         3
   Bungarus_wanghaotingi       0.68      0.57      0.62        23
    Calliophis_bivirgata       0.90      0.60      0.72        30
 Calliophis_macclellandi       1.00      0.36      0.53        14
Malayopython_reticulatus       0.39      0.50      0.44        30
          Naja_kaouthia       0.68      0.77      0.72        30
      Naja_mandalayensis       1.00      1.00      1.00         1
      Ophiophagus_hannah       0.70      0.47      0.56        30
            Ptyas_korros       0.54      0.83      0.66        30
            Ptyas_mucosa       0.68      0.57      0.62        30
       Python_bivittatus       0.40      0.33      0.36        30
          Python_molurus       0.32      0.57      0.41        30
...
                accuracy                           0.57       427
               macro avg       0.64      0.57      0.58       427
            weighted avg       0.60      0.57      0.57       427
```

5. Model Deployment

Although full production deployment was not completed, several deployment-ready components were implemented. The fine-tuned CNN is exported to ONNX with dynamic batch dimensions, enabling portable inference across Python, C++, and edge devices without retraining. Optional INT8 quantization via ONNX Runtime optimizes CPU inference, reducing model size and latency for resource-constrained devices.Class metadata is saved alongside the model to map outputs to species names consistently. Existing scripts and a dashboard interface

can leverage the ONNX model for batch or interactive evaluation. Standardized preprocessing (resize to 320, ImageNet normalization) ensures parity between training and inference. While a serving layer is not fully implemented, these artifacts establish a reliable pipeline ready for future deployment.

6. Code Implementation

The backbone's parameters were frozen to prevent updates during training, while gradients were enabled only for the classifier head, allowing the model to quickly learn class mappings from pre-trained features and stabilize transfer learning before full fine-tuning.

```
# Freeze backbone, train only the classifier head first (stabilizes transfer learning)
for n, p in model.named_parameters():
    p.requires_grad = False
for p in model.classifier.parameters():
    p.requires_grad = True

opt = torch.optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()), lr=args.lr)
ce = nn.CrossEntropyLoss()
for epoch in range(args.epochs):
    model.train()
    for x, y in tqdm(train_loader, desc=f"Epoch {epoch+1}/{args.epochs} [head]"):
        x, y = x.to(device), y.to(device)
        opt.zero_grad()
        loss = ce(model(x), y)
        loss.backward()
        opt.step()
    # ... validation + save best_head.pt ...
```

```
# Light fine-tuning: unfreeze the last feature block + classifier, lower LR to avoid overfitting
for n, p in model.named_parameters():
    p.requires_grad = False
for p in model.features[-1].parameters():
    p.requires_grad = True
for p in model.classifier.parameters():
    p.requires_grad = True

opt = torch.optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()), lr=args.lr * 0.5)
finetune_epochs = max(2, args.epochs // 2)
for epoch in range(finetune_epochs):
    model.train()
    for x, y in tqdm(train_loader, desc=f"Finetune {epoch+1}/{finetune_epochs}"):
        x, y = x.to(device), y.to(device)
        opt.zero_grad()
        loss = ce(model(x), y)
        loss.backward()
        opt.step()
```

These snippets demonstrate a two-stage transfer learning process: first, the backbone is frozen and only the classifier head is trained to stabilize transfer, then the last backbone block and classifier are fine-tuned with a lower learning rate to adapt high-level features while avoiding overfitting.

```python
# Export for deployment: ONNX with dynamic batch axes; optional INT8 quantization
dummy = torch.randn(1, 3, args.img_size, args.img_size, device="cpu")
model.cpu(); model.eval()
onnx_path = os.path.join(args.out_dir, "model_fp32.onnx")
torch.onnx.export(model, dummy, onnx_path,
                  input_names=["input"], output_names=["logits"], opset_version=17,
                  dynamic_axes={"input": {0: "batch"}, "logits": {0: "batch"}})

try:
    from onnxruntime.quantization import quantize_dynamic, QuantType
    int8_path = os.path.join(args.out_dir, "model_int8.onnx")
    quantize_dynamic(model_input=onnx_path, model_output=int8_path, weight_type=QuantType.QInt8)
    print("Quantized to INT8:", int8_path)
except Exception as e:
    print(f"Quantization failed: {e}")
```

Conclusion

The refinement phase established a lean but practical transfer-learning pipeline: head training followed by selective fine-tuning on lightweight backbones (EfficientNet-B0 for accuracy, MobileNetV3-Small for speed). Data robustness was improved via strong augmentations, SafeImageFolder corruption handling, and consistent normalization, which collectively aimed to reduce class confusion observed in earlier validation runs. Hyperparameters were tuned manually (LR schedule 1e-3 → 5e-4, modest epochs, batch 32–64) to balance convergence and overfitting risk on a small dataset. ONNX export and optional INT8 quantization were added to ensure the trained model could move smoothly into real-world inference contexts without retraining or heavyweight dependencies.

Challenges: Full evaluation has not yet been finalized; confusion matrices and accuracy outputs exist in script form but require execution on the held-out test set. No k-fold cross-validation or extensive hyperparameter search was performed due to time and computational constraints. Deployment integration (API/UI) remains to be implemented, despite the availability of portable artifacts. Nevertheless, the pipeline now reliably produces exportable artifacts (model_final.pt, model_fp32.onnx, optional INT8) with class mappings and a documented workflow: load the ONNX model, apply consistent preprocessing, and serve predictions. With a completed test run and a lightweight inference wrapper, the project can be productionized with predictable performance on resource-constrained hardware.

References

List any external resources, libraries, or papers that were referenced during the project.

- Libraries and docs: PyTorch (torch, torchvision), PIL/Pillow, NumPy, scikit-learn metrics, Matplotlib/Seaborn for viz, TQDM for progress, Requests/urllib for data fetches, FastAPI/Uvicorn for potential serving, ONNX/ONNX Runtime/onnxruntime-tools for export and quantization (requirements.txt).
- Model architectures: EfficientNet-B0 (Tan & Le, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”), MobileNetV3 (Howard et al., “Searching for MobileNetV3”), both via torchvision pretrained weights.
- Data sources/APIs: iNaturalist observations API (used in tools/download_inat.py).
- Practices: ImageNet normalization and transfer learning conventions for CNN finetuning.