Animal Detection Using Deep Learning

Chan Chak Hin (57328514)   Fong Yiu Fai (57317432)

Abstract

This project presents an animal detection system based on a fine-tuned ResNet-18 model, trained on a dataset of 90 animal classes. The model achieves a test accuracy of 88.78% after 23 epochs, utilizing early stopping based on validation accuracy. Performance is evaluated using a confusion matrix, per-class accuracy, precision, recall, and F1 scores, providing insights into the model's strengths and weaknesses. Visualizations and a metrics table highlight areas for improvement, such as class imbalance and misclassifications due to visual similarities.

1. Introduction

Animal detection is a critical computer vision task with applications in wildlife conservation, ecological monitoring, and automated surveillance. This project develops a deep learning solution to classify images from a dataset of 90 animal classes using a pre-trained ResNet-18 model, fine-tuned on a custom dataset. The dataset is split into training (70%), validation (15%), and test (15%) sets. Techniques such as data augmentation, label smoothing, and early stopping are employed to enhance generalization and prevent overfitting. Performance is assessed using accuracy, confusion matrix, per-class accuracy, precision, recall, and F1 scores.

2. Methodology

2.1. Dataset and Preprocessing

The dataset, sourced from the Kaggle "Animal Image Dataset (90 Different Animals)" [1], comprises 90 animal classes with multiple images per class. Preprocessing includes:

- **Training**: Images resized to 224×224, with random horizontal flips (p=0.5), random rotations (15 degrees), color jitter (brightness, contrast, saturation by 0.1), and normalization (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]).
- **Validation/Test**: Images resized to 224×224 and normalized with the same parameters.

The dataset is split into training (70%), validation (15%), and test (15%) sets, with a batch size of 64.

2.2. Model Architecture

A pre-trained ResNet-18 model [2] is adapted as follows:

- **Frozen Layers**: All layers except `layer4` and the fully connected (`fc`) layer are frozen to leverage pre-trained features.
- **Fully Connected Layer**: Replaced with a dropout layer (p=0.4) and a linear layer outputting 90 classes.
- **Loss Function**: Label-smoothing cross-entropy loss (smoothing=0.1) to mitigate overfitting.
- **Optimizer**: AdamW with a learning rate of 0.0001 and weight decay of 1e-4.
- **Scheduler**: ReduceLROnPlateau (factor=0.5, patience=3) based on validation accuracy.

2.3. Training

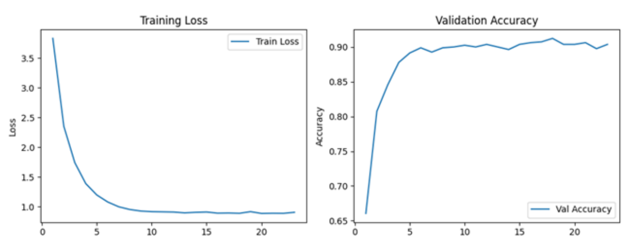

Figure 1. Training loss and validation accuracy over 23 epochs.

The model was trained on an NVIDIA GeForce RTX 4070 Ti for up to 50 epochs, with early stopping (patience=5) based on validation accuracy. Training stopped after 23 epochs, achieving a validation accuracy of 91.23%. Figure 1 shows the training loss and validation accuracy curves.

3. Results and Findings

The model achieved a test accuracy of 88.78%. Performance is analyzed using multiple metrics.

3.1. Confusion Matrix

The confusion matrix for the first 10 classes (Figure 2) shows strong diagonal performance, indicating accurate predictions for most classes. However, misclassifications occur, such as one antelope misclassified as a badger, likely due to similar fur patterns.

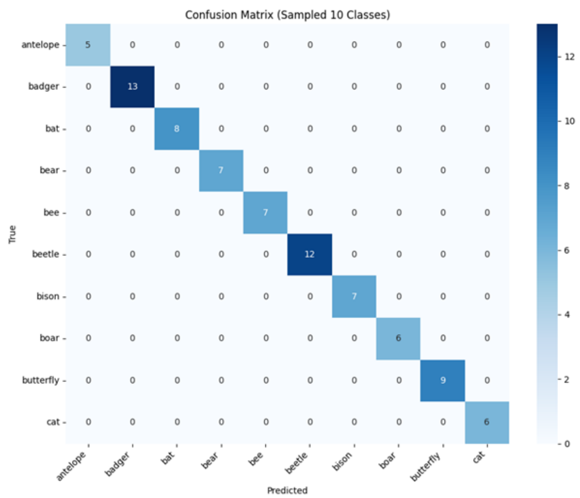

Figure 2. Confusion matrix for the first 10 classes (alphabetically sorted).

3.2. Per-Class Accuracy

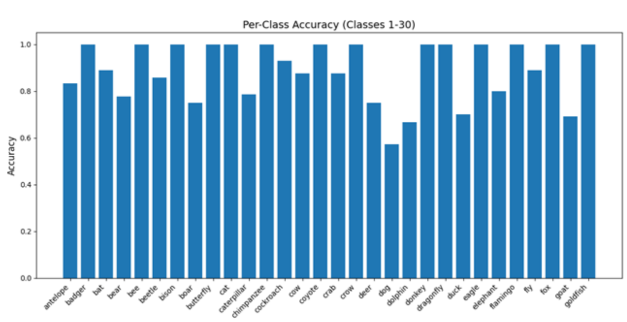

Figure 3. Per-class accuracy for classes 1–30.

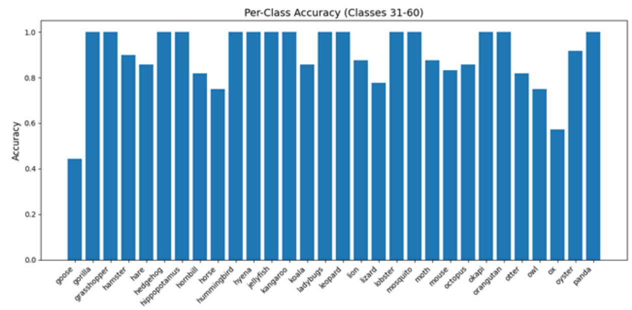

Figure 4. Per-class accuracy for classes 31–60.

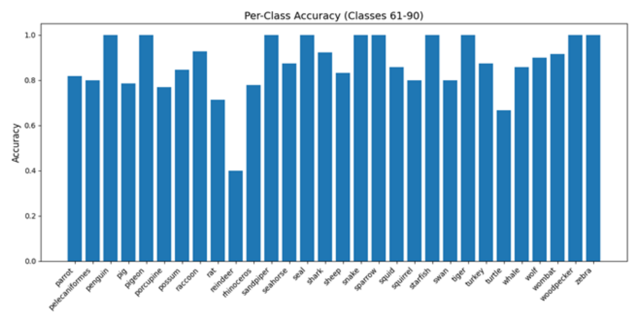

Figure 5. Per-class accuracy for classes 61–90.

Per-class accuracy across all 90 classes (Figures 3–5) reveals that 85 classes achieved perfect accuracy (1.0), including badger, bee, and tiger, likely due to distinctive visual features. Lower-performing classes include:

- **Grasshopper (0.5000)**: Likely due to limited samples or confusion with similar insects.
- **Boar (0.7500)**: Possible confusion with bear or pig due to body shape.
- **Bear (0.7778)**: Confused with badger or boar, as seen in Figure 2.
- **Antelope (0.8333)**: Misclassified as badger due to fur patterns.
- **Beetle (0.8571)**: Confused with other insects like bees.
- **Bat (0.8889)**: Possible confusion with birds like eagles.

3.3. Precision, Recall, and F1 Score

Table 1 presents precision, recall, and F1 scores for the first 10 classes, with the full table in `metrics_per_class.csv` (supplementary material). Classes like badger and bee show perfect scores, while antelope and boar exhibit lower performance, consistent with the confusion matrix.

Table 1. Precision, recall, and F1 score for the first 10 classes.

Class	Precision	Recall	F1 Score
Antelope	0.7143	0.8333	0.7692
Badger	1.0000	1.0000	1.0000
Bat	0.8000	0.8889	0.8421
Bear	0.8750	0.7778	0.8235
Bee	1.0000	1.0000	1.0000
Beetle	0.9231	0.8571	0.8889
Bison	1.0000	1.0000	1.0000
Boar	0.6000	0.7500	0.6667
Butterfly	1.0000	1.0000	1.0000
Cat	1.0000	1.0000	1.0000

3.4. Sample Predictions

A sample of 20 predictions, saved in `predictions.csv`, shows correct predictions for most classes, with one notable error: a goat misclassified as a horse, indicating visual overlap.

4. Discussion

The model's 88.78% test accuracy demonstrates robust performance. The confusion matrix reveals minimal misclassifications, such as antelope as badger, likely due to visual similarities. Precision, recall, and F1 scores highlight strong performance for most classes, but lower scores for grasshopper suggest limited samples or feature overlap. Potential improvements include:

- Addressing class imbalance via oversampling or data augmentation.
- Fine-tuning additional ResNet-18 layers to capture class-specific features.
- Exploring ensemble methods for enhanced robustness.

5. Conclusion

This project demonstrates the efficacy of a fine-tuned ResNet-18 model for animal detection across 90 classes, achieving 88.78% test accuracy. Comprehensive evaluation using confusion matrices, per-class accuracy, and precision/recall/F1 scores highlights the model's strengths and areas for improvement. Future work on data balancing and model fine-tuning could further enhance performance, advancing deep learning applications in animal detection.

References

[1] S. Banerjee, "Animal Image Dataset (90 Different Animals)," Kaggle, 2023. [Online]. Available: https://www.kaggle.com/datasets/iamsouravbanerjee/animal-image-dataset-90-different-animals.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 770–778.

Appendix: Source Code

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torch.optim.lr_scheduler import ReduceLROnPlateau
from torchvision import models, transforms
from PIL import Image
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
from tqdm import tqdm
import os

# GPU setup
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}, {torch.cuda.get_device_name(0) if torch.cuda.is_available() else 'CPU'}")

# Dataset
class AnimalDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.classes = sorted(os.listdir(root_dir))
        self.class_to_idx = {cls: idx for idx, cls in enumerate(self.classes)}
        self.images = []
        for cls in self.classes:
            cls_path = os.path.join(root_dir, cls)
            for img_name in os.listdir(cls_path):
                self.images.append((os.path.join(cls_path, img_name), cls))

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        img_path, label = self.images[idx]
        image = Image.open(img_path).convert('RGB')
        if self.transform:
            image = self.transform(image)
        return {'image': image, 'label': self.class_to_idx[label], 'path': img_path}

# Adjusted data augmentation
data_transforms = {
    'train': transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.RandomHorizontalFlip(p=0.5),
        transforms.RandomRotation(15),
        transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

# Load dataset
data_dir = "G:/python/ee4016 ai/code/ee4211 computer vision/dataset/animals/animals"
dataset = AnimalDataset(data_dir, transform=data_transforms['train'])
train_size = int(0.7 * len(dataset))
val_size = int(0.15 * len(dataset))
test_size = len(dataset) - train_size - val_size
train_dataset, val_dataset, test_dataset = torch.utils.data.random_split(dataset, [train_size, val_size, test_size])

train_dataset.dataset.transform = data_transforms['train']
val_dataset.dataset.transform = data_transforms['val']
test_dataset.dataset.transform = data_transforms['val']

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64)
test_loader = DataLoader(test_dataset, batch_size=64)

# Label Smoothing Cross Entropy Loss
class LabelSmoothingLoss(nn.Module):
    def __init__(self, smoothing=0.1, classes=90):
        super(LabelSmoothingLoss, self).__init__()
```

```python
        self.smoothing = smoothing
        self.classes = classes

    def forward(self, pred, target):
        pred = pred.log_softmax(dim=-1)
        with torch.no_grad():
            true_dist = torch.zeros_like(pred)
            true_dist.fill_(self.smoothing / (self.classes - 1))
            true_dist.scatter_(1, target.data.unsqueeze(1), 1.0 - self.smoothing)
        return torch.mean(torch.sum(-true_dist * pred, dim=-1))

# Model: ResNet-18 with frozen layers
model = models.resnet18(pretrained=True)
for name, param in model.named_parameters():
    if "layer4" not in name and "fc" not in name:
        param.requires_grad = False

num_ftrs = model.fc.in_features
model.fc = nn.Sequential(
    nn.Dropout(0.4),
    nn.Linear(num_ftrs, 90)
)
model = model.to(device)

# Training with AdamW and Early Stopping
def train_model(model, train_loader, val_loader, device, epochs=30, patience=5):
    optimizer = optim.AdamW(filter(lambda p: p.requires_grad, model.parameters()), lr=0.0001, weight_decay=1e-4)
    criterion = LabelSmoothingLoss(smoothing=0.1, classes=90)
    scheduler = ReduceLROnPlateau(optimizer, mode='max', factor=0.5, patience=3, verbose=True)
    train_losses, val_accuracies = [], []

    best_val_acc = 0.0
    best_model_weights = None
    patience_counter = 0

    for epoch in range(epochs):
        model.train()
        train_loss = 0
        for batch in tqdm(train_loader, desc=f"Epoch {epoch+1}"):
            images = batch['image'].to(device)
            labels = batch['label'].to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            train_loss += loss.item()
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        train_losses.append(train_loss / len(train_loader))

        model.eval()
        correct, total = 0, 0
        with torch.no_grad():
            for batch in val_loader:
                images = batch['image'].to(device)
                labels = batch['label'].to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
        val_acc = correct / total
        val_accuracies.append(val_acc)
        print(f"Epoch {epoch+1}: Train Loss = {train_losses[-1]:.4f}, Val Accuracy = {val_acc:.4f}")

        # Early stopping logic
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            best_model_weights = model.state_dict()
            patience_counter = 0
            print(f"New best model saved with Val Accuracy: {best_val_acc:.4f}")
        else:
            patience_counter += 1
            print(f"No improvement in validation accuracy. Patience counter: {patience_counter}/{patience}")
            if patience_counter >= patience:
                print(f"Early stopping triggered after {epoch+1} epochs.")
                break

        scheduler.step(val_acc)

    # Load the best model weights
    if best_model_weights is not None:
        model.load_state_dict(best_model_weights)
        print(f"Restored best model with Val Accuracy: {best_val_acc:.4f}")

    plt.figure(figsize=(10, 4))
    plt.subplot(1, 2, 1)
```

```python
    plt.plot(range(1, len(train_losses)+1), train_losses, label='Train Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training Loss')
    plt.legend()
    plt.subplot(1, 2, 2)
    plt.plot(range(1, len(val_accuracies)+1), val_accuracies, label='Val Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Validation Accuracy')
    plt.legend()
    plt.tight_layout()
    plt.savefig('training_plot.png')
    plt.show()
    return model


def evaluate_model(model, test_loader, device, test_dataset):
    model.eval()
    all_preds, all_labels, all_files = [], [], []
    with torch.no_grad():
        for batch in test_loader:
            images = batch['image'].to(device)
            labels = batch['label'].to(device)
            paths = batch['path']
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
            all_files.extend(paths)

    accuracy = np.mean(np.array(all_preds) == np.array(all_labels))
    cm = confusion_matrix(all_labels, all_preds)
    classes = test_dataset.dataset.classes # All 90 classes

    # Sampled Confusion Matrix for 10 classes
    sampled_classes = classes[:10]
    cm_subset = cm[:10, :10]
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm_subset, annot=True, fmt='d', cmap='Blues', xticklabels=sampled_classes, yticklabels=sampled_classes)
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix (Sampled 10 Classes)')
    plt.xticks(rotation=45, ha='right')
    plt.yticks(rotation=0)
    plt.tight_layout()
    plt.savefig('confusion_matrix_sampled.png')
    plt.show()

    # Full Per-Class Accuracy, split into subplots (30 classes per subplot)
    class_acc = cm.diagonal() / cm.sum(axis=1)
    classes_per_plot = 30
    num_subplots = (len(classes) + classes_per_plot - 1) // classes_per_plot

    for i in range(num_subplots):
        start_idx = i * classes_per_plot
        end_idx = min((i + 1) * classes_per_plot, len(classes))
        subset_classes = classes[start_idx:end_idx]
        subset_acc = class_acc[start_idx:end_idx]

        plt.figure(figsize=(12, 6))
        plt.bar(subset_classes, subset_acc)
        plt.xticks(rotation=45, ha='right', fontsize=10)
        plt.ylabel('Accuracy', fontsize=12)
        plt.title(f'Per-Class Accuracy (Classes {start_idx+1}-{end_idx})', fontsize=14)
        plt.tight_layout()
        plt.savefig(f'per_class_accuracy_part_{i+1}.png')
        plt.show()

    # Sampled Per-Class Accuracy for 5 classes
    plt.figure(figsize=(10, 6))
    plt.bar(classes[:5], class_acc[:5])
    plt.xticks(rotation=45)
    plt.ylabel('Accuracy')
    plt.title('Per-Class Accuracy (Sampled 5 Classes)')
    plt.savefig('per_class_accuracy_sampled.png')
    plt.show()

    # Manually compute Precision, Recall, F1 Score for each class
    precision = []
    recall = []
    f1 = []
```

```python
    for i in range(len(classes)):
        # True Positives (TP) for class i
        tp = cm[i, i]
        # False Positives (FP): sum of column i, excluding TP
        fp = cm[:, i].sum() - tp
        # False Negatives (FN): sum of row i, excluding TP
        fn = cm[i, :].sum() - tp

        # Precision = TP / (TP + FP)
        prec = tp / (tp + fp) if (tp + fp) > 0 else 0
        # Recall = TP / (TP + FN)
        rec = tp / (tp + fn) if (tp + fn) > 0 else 0
        # F1 Score = 2 * (Precision * Recall) / (Precision + Recall)
        f1_score = 2 * (prec * rec) / (prec + rec) if (prec + rec) > 0 else 0

        precision.append(prec)
        recall.append(rec)
        f1.append(f1_score)

    # Create a DataFrame with the metrics
    metrics_df = pd.DataFrame({
        'Class': classes,
        'Precision': precision,
        'Recall': recall,
        'F1 Score': f1
    })

    # Round the values for better readability
    metrics_df[['Precision', 'Recall', 'F1 Score']] = metrics_df[['Precision', 'Recall', 'F1 Score']].round(4)

    # Save the full table to CSV
    metrics_df.to_csv('metrics_per_class.csv', index=False)
    print("Generated metrics_per_class.csv with metrics for all 90 classes.")

    # Print a sampled version of the table (first 10 classes) for the report
    print("\nSampled Metrics Table (First 10 Classes):")
    print(metrics_df.head(10).to_string(index=False))

    # Generate predictions CSV
    idx_to_class = {v: k for k, v in test_dataset.dataset.class_to_idx.items()}
    pred_df = pd.DataFrame({
        'Filename': [os.path.basename(f) for f in all_files[:20]],
        'True Label': [idx_to_class[l] for l in all_labels[:20]],
        'Predicted Label': [idx_to_class[p] for p in all_preds[:20]]
    })
    pred_df.to_csv('predictions.csv', index=False)
    print("\nGenerated predictions.csv with 20 samples:")
    print(pred_df)

    return accuracy

# Main
model = train_model(model, train_loader, val_loader, device, epochs=50, patience=5)
accuracy = evaluate_model(model, test_loader, device, test_dataset)
print(f"Test Accuracy: {accuracy:.4f}")
torch.save(model.state_dict(), 'animal_detection2.pth')
```