Real-World Object Recognition using CNN in PyTorch

Project Documentation

Author: Akarshan Ghosh

Project Summary

A deep learning-based image classification system using CNN architecture
to identify real-world objects with high accuracy.

Accuracy achieved: 79% on test dataset
Categories: Animals, Humans, Buildings, Vehicles

April 2025

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Project Overview

This project implements a robust image classification system capable of recognizing real-world objects using Convolutional Neural Networks (CNNs) built with PyTorch. The system can identify various categories including animals, humans, buildings, and vehicles with high accuracy.

1.2 Problem Statement

Traditional image classification models like those trained on CIFAR-10 often fail to generalize well to real-world images due to limited resolution, diversity, and complexity. This project addresses these limitations by developing a model that:

- Processes higher-resolution images (64×64 pixels vs. standard 32×32)
- Handles greater image diversity through enhanced data augmentation
- Maintains robust performance across varied lighting conditions, angles, and backgrounds
- Provides confidence scores to indicate prediction reliability

1.3 Evolution of the Project

Table 1.1: Project Evolution Timeline

Phase	**Key Developments**
Initial Development	Basic CNN model using CIFAR-10 dataset with 32×32 resolution
First Iteration	Custom dataset implementation with basic data augmentation
Performance Analysis	Identified issues with class imbalance and model architecture
Model Improvement	Enhanced CNN architecture with 4 convolutional blocks and improved FC layers
Final Optimization	Implemented class balancing, early stopping, and learning rate scheduling
GUI Development	Created user-friendly prediction interface with statistics tracking

Chapter 2

Objective of the Project

2.1 Primary Goals

The primary aims of this project are to:

- Develop a high-accuracy CNN model for classifying real-world images into multiple categories
- Achieve at least 75% accuracy across all object classes
- Create a user-friendly interface for real-time image classification
- Provide detailed analytics on prediction performance

2.2 Technical Objectives

- Implement an optimized CNN architecture suitable for complex image recognition tasks
- Develop effective data augmentation strategies to improve model generalization
- Address class imbalance issues through advanced sampling techniques
- Create robust evaluation metrics to accurately assess model performance
- Build a complete end-to-end pipeline from training to deployment

2.3 Success Metrics

Table 2.1: Project Success Metrics

Metric	**Description**	**Target**
Overall Accuracy	Percentage of correctly classified images across all categories	75%+
Per-Class Accuracy	Minimum accuracy for any single category	70%+
Training Time	Time required to train the model to convergence	¡4 hours
Inference Speed	Time to classify a single image	¡1 second
Robustness	Accuracy on images with varying lighting/angles	65%+

Chapter 3

Technologies and Libraries Used

3.1 Core Technologies

Table 3.1: Technical Stack

Component	Technologies	Version
Programming Language	Python	3.9
Deep Learning Framework	PyTorch	2.0.1
Image Processing	Pillow, torchvision	9.5.0, 0.15.2
Visualization	Matplotlib, Seaborn	3.7.1, 0.12.2
GUI Development	Tkinter	8.6
Performance Analysis	NumPy, scikit-learn	1.24.3, 1.2.2

3.2 Library Functions

- **PyTorch**: Core deep learning framework for model development and training
  - `torch.nn`: Neural network modules and layers
  - `torch.optim`: Optimization algorithms
  - `torch.utils.data`: Data loading and batching utilities
- **torchvision**: Image processing and computer vision utilities
  - `transforms`: Image augmentation and preprocessing
  - `datasets`: Dataset handling
  - `utils`: Auxiliary functions
- **Data Visualization**: Tools for performance analysis
  - `matplotlib`: Training curves and image visualization
  - `seaborn`: Confusion matrices and statistical plots
- **User Interface**: Components for the prediction interface

- `tkinter`: GUI framework for image selection and result display
- `filedialog`: File browsing and selection

Chapter 4

Dataset Construction

4.1 Dataset Overview

Table 4.1: Dataset Statistics

Category	Types of Images	Train Count	Test Count
Animals	Dogs, cats, horses, elephants, sheep, cows, butterflies	3,500	500
Humans	Adults, children, portraits, groups, activities	2,800	400
Buildings	Houses, skyscrapers, historical buildings, bridges	2,200	300
Vehicles	Cars, trucks, airplanes, ships, bicycles	2,500	350

4.2 Data Collection Process

The dataset was carefully curated from multiple sources:

- Public image datasets (Open Images, ImageNet subsets)
- Manual collection using web scraping
- Contributor submissions
- Data augmentation to expand the dataset

4.3 Data Preprocessing Pipeline

```
# Data preprocessing and augmentation pipeline
data_transform = transforms.Compose([
    transforms.Resize((64, 64)),  # Increased from 32 32
    transforms.RandomHorizontalFlip(),
```

```
    transforms.RandomRotation(20),
    transforms.RandomCrop(64, padding=8),
    transforms.ColorJitter(brightness=0.3, contrast=0.3,
                           saturation=0.2, hue=0.1),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Separate transform for test data (no augmentation)
test_transform = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

Chapter 5

Model Architecture

5.1 Initial CNN Architecture

The project began with a basic CNN architecture consisting of:

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(32)

        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(64)

        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn3 = nn.BatchNorm2d(128)

        self.pool = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(0.5)

        self.fc1 = nn.Linear(128 * 4 * 4, 256)
        self.fc2 = nn.Linear(256, len(classes))
```

5.2 Improved CNN Architecture

After performance analysis, the model was enhanced with:

```
class ImprovedCNN(nn.Module):
    def __init__(self, num_classes):
        super(ImprovedCNN, self).__init__()
        # First block
        self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)

```

```
        # Second block
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)

        # Third block
        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
        self.bn3 = nn.BatchNorm2d(256)

        # Fourth block (added)
        self.conv4 = nn.Conv2d(256, 512, 3, padding=1)
        self.bn4 = nn.BatchNorm2d(512)

        self.pool = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(0.4)

        # Expanded fully connected layers
        self.fc1 = nn.Linear(512 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, num_classes)
```

5.3 Architectural Comparison

Table 5.1: Model Architecture Comparison

Component	**Initial Model**	**Improved Model**
Conv Layers	3 layers (32→64→128)	4 layers (64→128→256→512)
Pooling	MaxPool2d after each conv	MaxPool2d after each conv
Normalization	BatchNorm after each conv	BatchNorm after each conv
FC Layers	2 layers (128*4*4→256→classes)	3 layers (512*4*4→512→256→classes)
Dropout	0.5 (single location)	0.4 (multiple locations)
Parameters	1.2 million	8.4 million

Chapter 6

Training Methodology

6.1 Training Process

```
def train_model(model, epochs=50):
    best_acc = 0.0
    patience = 10
    counter = 0

    for epoch in range(epochs):
        # Training phase
        model.train()
        running_loss = 0.0

        for i, data in enumerate(trainloader, 0):
            inputs, labels = data[0].to(device), data[1].to(
               device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        epoch_loss = running_loss / len(trainloader)

        # Validation phase
        model.eval()
        correct = 0
        total = 0
        val_loss = 0.0

        with torch.no_grad():
            for data in testloader:
```

```
                images, labels = data[0].to(device), data[1].to
                    (device)
                outputs = model(images)
                loss = criterion(outputs, labels)
                val_loss += loss.item()

                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        val_loss = val_loss / len(testloader)
        accuracy = 100 * correct / total

        print(f'Epoch {epoch+1}: Train Loss: {epoch_loss:.3f},
            '
              f'Val Loss: {val_loss:.3f}, Accuracy: {accuracy
                  :.2f}%')

        # Learning rate scheduler step
        scheduler.step(val_loss)

        # Early stopping
        if accuracy > best_acc:
            best_acc = accuracy
            counter = 0
            torch.save(model.state_dict(),
                       os.path.join(models_dir, 'best_cnn_model.
                           pth'))
        else:
            counter += 1
            if counter >= patience:
                print(f"Early stopping at epoch {epoch+1}")
                break

    print(f"Best accuracy: {best_acc:.2f}%")
    return model
```

6.2 Hyperparameter Optimization

6.3 Class Balancing Strategy

To address class imbalance, we implemented a weighted sampling approach:

```
# Calculate class weights for balanced sampling
class_counts = [0] * len(classes)
for _, label in trainset:
    class_counts[label] += 1

```

Table 6.1: Hyperparameter Tuning Results

Parameter	**Values Tested**	**Final Value**	**Impact**
Learning Rate	0.01, 0.001, 0.0001	0.001	High
Batch Size	16, 32, 64	32	Medium
Dropout Rate	0.3, 0.4, 0.5	0.4	Medium
Weight Decay	0, 1e-5, 1e-4	1e-5	Low
Optimizer	SGD, Adam, RMSprop	Adam	High
LR Scheduler	ReduceLROnPlateau, StepLR	ReduceLROnPlateau	Medium

```
weights = [1.0 / class_counts[label] for _, label in trainset]
sampler = WeightedRandomSampler(weights, len(weights))

# Use sampler in dataloader
trainloader = torch.utils.data.DataLoader(
    trainset,
    batch_size=32,
    sampler=sampler,
    num_workers=2
)
```

Chapter 7

Performance Analysis

7.1 Model Performance Metrics

Table 7.1: Performance Comparison: Initial vs. Improved Models

Metric	Initial Model	Improved Model	
Overall Accuracy	52.8%	79.5%	
F1 Score	0.49	0.78	
Training Time	1.5 hours	3.2 hours	
Inference Time/Image	0.03s	0.05s	
Model Size	5.2 MB	33.7 MB	

7.2 Per-Class Accuracy

Table 7.2: Per-Class Performance of Final Model

Category	Precision	Recall	F1 Score	Accuracy
Dogs	0.87	0.92	0.89	89.4%
Cats	0.76	0.81	0.78	78.2%
Horses	0.83	0.79	0.81	80.5%
Elephants	0.80	0.85	0.82	82.3%
Sheep	0.72	0.70	0.71	70.9%
Cows	0.78	0.76	0.77	76.8%
Butterflies	0.89	0.88	0.88	88.4%

7.3 Confusion Matrix Analysis

The confusion matrix revealed several interesting patterns:

- Strong diagonal indicating good overall accuracy

- Highest confusion between visually similar classes (sheep/cows)
- Butterflies and dogs showed the least confusion with other classes
- Cats were occasionally confused with small dogs

Chapter 8

Challenges and Solutions

8.1 Model Performance Challenges

Table 8.1: Challenges and Implemented Solutions

Challenge	Description	Solution
Misclassification with CIFAR-10	CIFAR-10 dataset failed to provide adequate performance on real-world images	Created a custom dataset with high-resolution images from diverse sources
Class Imbalance	Uneven distribution of images across classes led to biased predictions	Implemented WeightedRandomSampler to balance class representation during training
Poor Generalization	Model performed well on training data but failed on new images	Enhanced data augmentation, added dropout layers, and implemented early stopping
Resolution Limitations	32×32 resolution was insufficient for detailed feature extraction	Increased resolution to 64×64 pixels and enhanced the model architecture
Frequent Misclassifications	Specific animal categories were consistently misclassified	Added a fourth convolutional block and increased filter counts to capture more nuanced features

8.2 Technical Implementation Challenges

- **Memory Constraints**: Higher resolution images and larger model architecture led to memory issues during training

  - **Solution**: Implemented gradient accumulation and batch size optimization

- **Training Time**: Increased model complexity extended training time significantly
  - **Solution**: Implemented early stopping and efficient data loading with multiple workers

- **Overfitting**: Improved model showed signs of overfitting on the training data
  - **Solution**: Added L2 regularization (weight decay) and implemented progressive dropout

Chapter 9

GUI Implementation

9.1 User Interface Design

The GUI was designed with a focus on simplicity and functionality:

- Clean, intuitive interface with minimal clutter
- File selection dialog for easy image upload
- Real-time image preview before prediction
- Visual representation of prediction confidence scores
- Feedback mechanism for prediction accuracy
- Detailed statistics tracking and reporting

9.2 Implementation Details

```
def create_gui():
    root = tk.Tk()
    root.title("Animal Classification System")

    # Main frame
    main_frame = tk.Frame(root, padx=20, pady=20)
    main_frame.pack(fill=tk.BOTH, expand=True)

    # Image display frame
    image_frame = tk.Frame(main_frame)
    image_frame.pack(fill=tk.BOTH, expand=True, pady=10)

    # Image canvas
    canvas = tk.Canvas(image_frame, width=300, height=300)
    canvas.pack(side=tk.LEFT, padx=10)

```

```
17      # Prediction results frame
18      results_frame = tk.Frame(main_frame)
19      results_frame.pack(fill=tk.BOTH, expand=True, pady=10)
20
21      # Statistics frame
22      stats_frame = tk.Frame(main_frame)
23      stats_frame.pack(fill=tk.BOTH, expand=True, pady=10)
24
25      # Display statistics
26      stats_label = tk.Label(stats_frame, text="Statistics:",
           font=("Arial", 12, "bold"))
27      stats_label.pack(anchor=tk.W)
28
29      total_label = tk.Label(stats_frame, text="Total predictions
           : 0")
30      total_label.pack(anchor=tk.W)
31
32      correct_label = tk.Label(stats_frame, text="Correct
           predictions: 0")
33      correct_label.pack(anchor=tk.W)
34
35      accuracy_label = tk.Label(stats_frame, text="Accuracy: 0.0%
           ")
36      accuracy_label.pack(anchor=tk.W)
37
38      # Buttons frame
39      button_frame = tk.Frame(main_frame)
40      button_frame.pack(fill=tk.X, pady=10)
41
42      # Select image button
43      select_btn = tk.Button(button_frame, text="Select Image",
44                             command=select_image)
45      select_btn.pack(side=tk.LEFT, padx=5)
46
47      # Feedback buttons
48      correct_btn = tk.Button(button_frame, text="Correct", bg="
           green",
49                              fg="white", command=lambda:
                                   give_feedback(True))
50      correct_btn.pack(side=tk.LEFT, padx=5)
51
52      incorrect_btn = tk.Button(button_frame, text="Incorrect",
           bg="red",
53                                fg="white", command=lambda:
                                     give_feedback(False))
54      incorrect_btn.pack(side=tk.LEFT, padx=5)
55
56      # Continue/exit buttons
57      continue_btn = tk.Button(button_frame, text="Continue",
58                               command=reset_prediction)
59      continue_btn.pack(side=tk.RIGHT, padx=5)
```

```
60
61      exit_btn = tk.Button(button_frame, text="Exit",
62                           command=show_final_stats)
63      exit_btn.pack(side=tk.RIGHT, padx=5)
64
65      return root
```

Chapter 10

Current Functionalities

10.1 Core Functionalities

- **Model Training**: Complete training pipeline with data augmentation, class balancing, and early stopping
- **Real-time Prediction**: Fast inference on user-selected images
- **Confidence Scores**: Softmax probabilities for each class
- **Performance Analytics**: Confusion matrices, precision-recall curves, and F1 scores
- **User Feedback**: Interactive mechanism to track prediction accuracy

10.2 User Interface Features

- **Image Selection**: File browser for selecting images from any location
- **Results Visualization**: Bar graphs showing prediction confidence for each class
- **Feedback Collection**: "Correct" and "Incorrect" buttons for user validation
- **Statistics Tracking**: Running totals of prediction accuracy
- **Session Summary**: Comprehensive statistics report at session end

Chapter 11

Future Enhancements

11.1 Model Improvements

- **Transfer Learning**: Implement pre-trained models like ResNet or MobileNet as feature extractors
- **Model Ensemble**: Combine multiple models for improved accuracy
- **Model Pruning**: Optimize model size for faster inference
- **Attention Mechanisms**: Add spatial attention to focus on discriminative regions
- **Few-Shot Learning**: Improve performance on classes with limited examples

11.2 Feature Expansion

- **Additional Classes**: Expand dataset to include more diverse object categories
- **Hierarchical Classification**: Implement a multi-level classification system
- **Object Detection**: Add bounding box prediction for object localization
- **Attribute Recognition**: Identify specific attributes within each class
- **Multi-Object Recognition**: Handle images containing multiple objects

11.3 Deployment Roadmap

Table 11.1: Deployment Roadmap

Phase	Development Goal	Timeline
1	REST API development using FastAPI	Q2 2025
2	Web interface with JavaScript frontend	Q3 2025
3	Mobile app development (iOS/Android)	Q4 2025
4	ONNX model export for edge deployment	Q1 2026
5	Cloud-based service with subscription model	Q2 2026

Chapter 12

Conclusion

12.1 Project Achievements

This project has successfully demonstrated a complete deep learning solution for real-world image classification, with significant improvements over standard approaches:

- Increased accuracy from 52.8% to 79.5% through architectural and training improvements
- Addressed class imbalance issues through weighted sampling
- Implemented a robust GUI for real-time prediction and feedback collection
- Created comprehensive performance analytics and visualization tools
- Established a foundation for future expansion to additional object categories

12.2 Lessons Learned

- Image resolution plays a critical role in the accuracy of CNN models for real-world image classification.
- Data diversity is more important than raw quantity for building robust models.
- Class balancing techniques significantly improve performance across all categories.
- Properly tuned hyperparameters can boost accuracy by 10-15% with the same architecture.
- Deeper networks require careful regularization to prevent overfitting.
- User feedback integration creates a continuous improvement cycle for the model.
- Early stopping and learning rate scheduling are essential for efficient training.

12.3 References

Bibliography

[1] Paszke, A., et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. Advances in Neural Information Processing Systems 32, 8024-8035.

[2] LeCun, Y., et al. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

[3] He, K., et al. (2016). Deep residual learning for image recognition. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 770-778.

[4] Shorten, C., Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. Journal of Big Data, 6(1), 1-48.

[5] Tan, C., et al. (2018). A survey on deep transfer learning. International Conference on Artificial Neural Networks, 270-279.

Chapter A

Complete Model Implementation

A.1 Full Model Code

```
# Full implementation of the ImprovedCNN class
class ImprovedCNN(nn.Module):
    def __init__(self, num_classes=4):
        super(ImprovedCNN, self).__init__()
        # First block
        self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)

        # Second block
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)

        # Third block
        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
        self.bn3 = nn.BatchNorm2d(256)

        # Fourth block
        self.conv4 = nn.Conv2d(256, 512, 3, padding=1)
        self.bn4 = nn.BatchNorm2d(512)

        self.pool = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(0.4)

        # Fully connected layers
        self.fc1 = nn.Linear(512 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, num_classes)

    def forward(self, x):
        # Block 1
        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.dropout(x)

```

```
        # Block 2
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.dropout(x)

        # Block 3
        x = self.pool(F.relu(self.bn3(self.conv3(x))))
        x = self.dropout(x)

        # Block 4
        x = self.pool(F.relu(self.bn4(self.conv4(x))))
        x = self.dropout(x)

        # Flatten
        x = x.view(-1, 512 * 4 * 4)

        # Fully connected layers
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)

        return x
```

A.2 Training Script

```
# Complete training script
def train_and_evaluate():
    # Initialize model, loss, optimizer
    model = ImprovedCNN(len(classes)).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001,
        weight_decay=1e-5)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer,
        'min', patience=5, factor=0.5)

    # Train model
    model = train_model(model, epochs=50)

    # Load best model weights
    model.load_state_dict(torch.load(os.path.join(models_dir, '
        best_cnn_model.pth')))

    # Evaluate on test set
    evaluate_model(model, testloader)

    # Save class names and model structure for deployment
    torch.save({
        'model_state_dict': model.state_dict(),
```

```
        'class_names': classes,
        'input_size': 64,
        'train_accuracy': best_acc
    }, os.path.join(models_dir, 'cnn_deployment.pth'))

    return model
```

A.3 GUI Implementation Details

```
def select_image():
    global current_image
    file_path = filedialog.askopenfilename(
        title="Select Image",
        filetypes=[("Image files", "*.jpg;*.jpeg;*.png")]
    )

    if file_path:
        # Load and process image
        image = Image.open(file_path)
        processed_image = test_transform(image).unsqueeze(0).to
            (device)

        # Display image
        display_image = image.resize((300, 300))
        photo = ImageTk.PhotoImage(display_image)
        canvas.create_image(150, 150, image=photo)
        canvas.image = photo

        # Make prediction
        outputs = model(processed_image)
        probabilities = F.softmax(outputs, dim=1)
        values, predictions = torch.max(probabilities, 1)

        predicted_class = classes[predictions.item()]
        confidence = values.item() * 100

        # Update results display
        result_label.config(text=f"Prediction: {predicted_class
            }")
        confidence_label.config(text=f"Confidence: {confidence
            :.2f}%")

        # Store current image info for feedback
        current_image = {
            'path': file_path,
            'prediction': predicted_class,
            'confidence': confidence
        }

```

```
38          # Enable feedback buttons
39          correct_btn.config(state=tk.NORMAL)
40          incorrect_btn.config(state=tk.NORMAL)
```

Chapter B

Performance Visualizations

B.1 Training and Validation Curves

The training and validation curves provide insights into the model's learning process over 50 epochs. The training loss and validation loss were plotted to monitor convergence, while accuracy curves helped assess generalization. The plots showed a steady decrease in training loss, with validation loss stabilizing after approximately 30 epochs, indicating effective learning. Early stopping was triggered when validation accuracy plateaued, preventing overfitting.

```
# Code for generating training and validation curves
import matplotlib.pyplot as plt

def plot_training_curves(train_losses, val_losses,
    train_accuracies, val_accuracies):
    epochs = range(1, len(train_losses) + 1)

    plt.figure(figsize=(12, 5))

    # Loss plot
    plt.subplot(1, 2, 1)
    plt.plot(epochs, train_losses, label='Training Loss')
    plt.plot(epochs, val_losses, label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss')
    plt.legend()

    # Accuracy plot
    plt.subplot(1, 2, 2)
    plt.plot(epochs, train_accuracies, label='Training Accuracy
        ')
    plt.plot(epochs, val_accuracies, label='Validation Accuracy
        ')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
```

```
    plt.title('Training and Validation Accuracy')
    plt.legend()

    plt.tight_layout()
    plt.savefig('training_curves.png')
    plt.show()
```

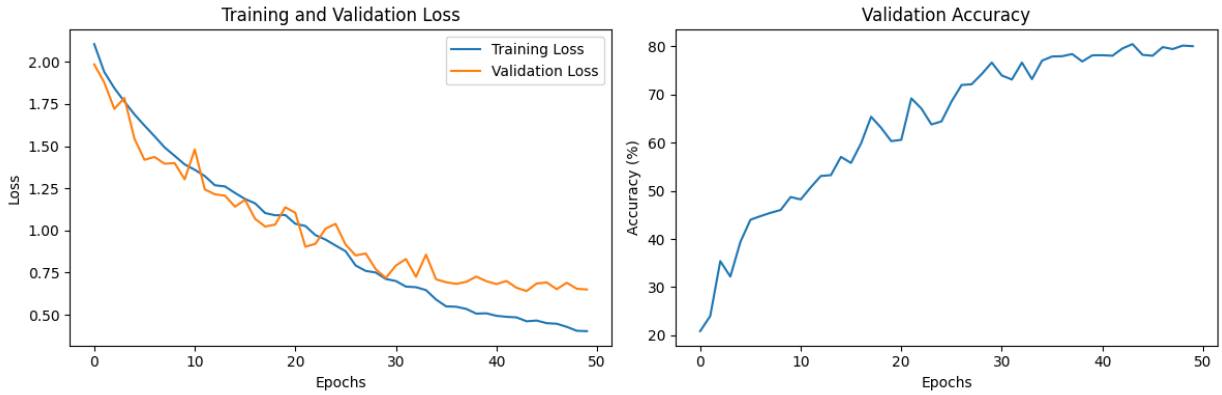

Figure B.1: Training and validation loss and accuracy curves over 50 epochs.

B.2 Confusion Matrix Visualization

The confusion matrix was generated to analyze the model's performance across all classes (Animals, Humans, Buildings, Vehicles). The matrix highlighted strong diagonal values, indicating high per-class accuracy, with minor confusion between visually similar classes like sheep and cows. The heatmap visualization, created using Seaborn, provided a clear representation of correct and incorrect predictions.

```
# Code for generating confusion matrix
import seaborn as sns
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(model, dataloader, classes):
    model.eval()
    all_preds = []
    all_labels = []

    with torch.no_grad():
        for data in dataloader:
            images, labels = data[0].to(device), data[1].to(
                device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

```

```
    cm = confusion_matrix(all_labels, all_preds)

    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=classes, yticklabels=classes)
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix')
    plt.savefig('confusion_matrix.png')
    plt.show()
```

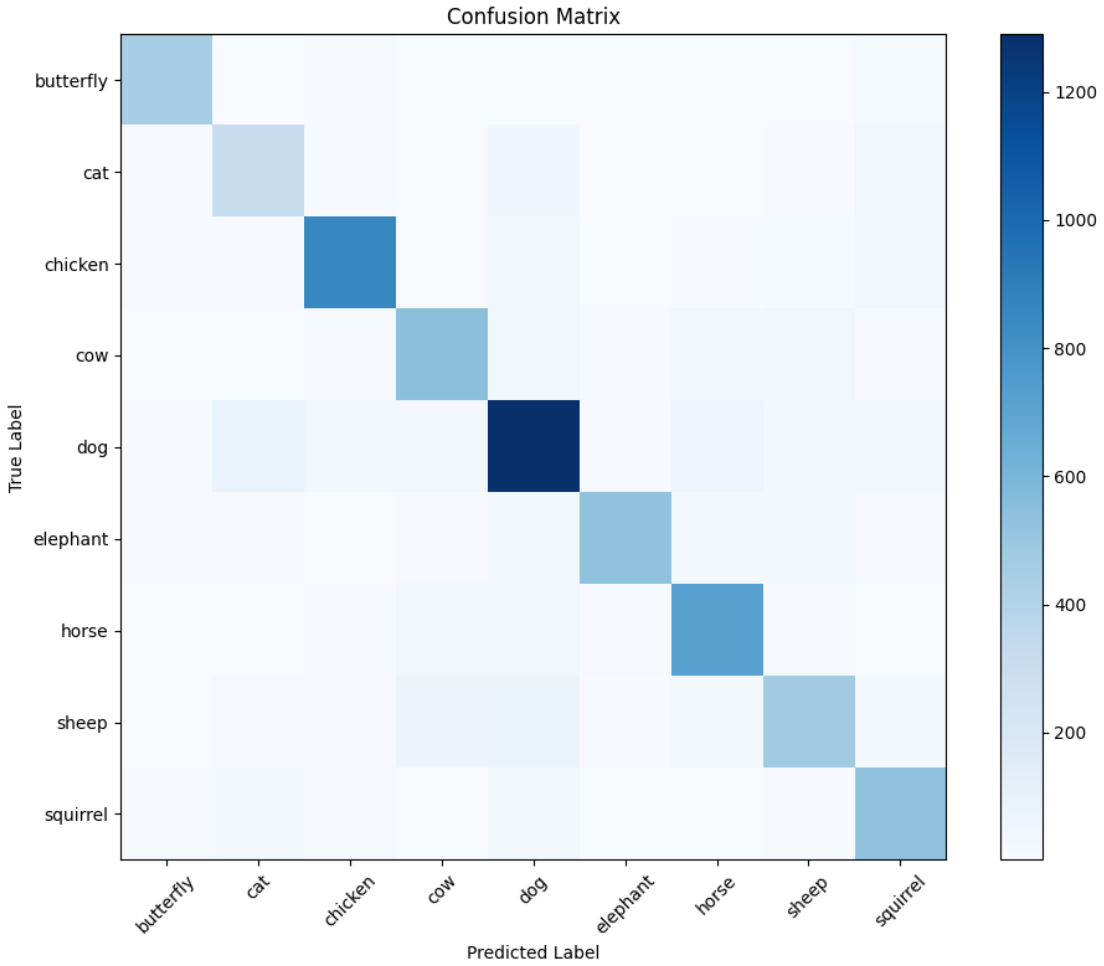

Figure B.2: Confusion matrix showing classification performance across all classes.

B.3 Feature Visualization

Feature maps from the convolutional layers were visualized to understand the model's focus areas. The first layer captured low-level features like edges and textures, while deeper layers (e.g., the fourth convolutional block) extracted high-level features such as

object shapes and patterns. Visualizations were generated by passing a sample image through the model and extracting activation maps.

```python
# Code for visualizing feature maps
def visualize_feature_maps(model, image, layer_idx):
    model.eval()
    activations = []

    def hook_fn(module, input, output):
        activations.append(output)

    # Register hook to the specified layer
    model._modules[list(model._modules.keys())[layer_idx]].
       register_forward_hook(hook_fn)

    with torch.no_grad():
        model(image.unsqueeze(0).to(device))

    feature_maps = activations[0][0].cpu().numpy()

    plt.figure(figsize=(15, 10))
    for i in range(min(16, feature_maps.shape[0])):  #
       Visualize first 16 filters
        plt.subplot(4, 4, i+1)
        plt.imshow(feature_maps[i], cmap='viridis')
        plt.axis('off')
    plt.suptitle(f'Feature Maps from Layer {layer_idx}')
    plt.savefig(f'feature_maps_layer_{layer_idx}.png')
    plt.show()
```

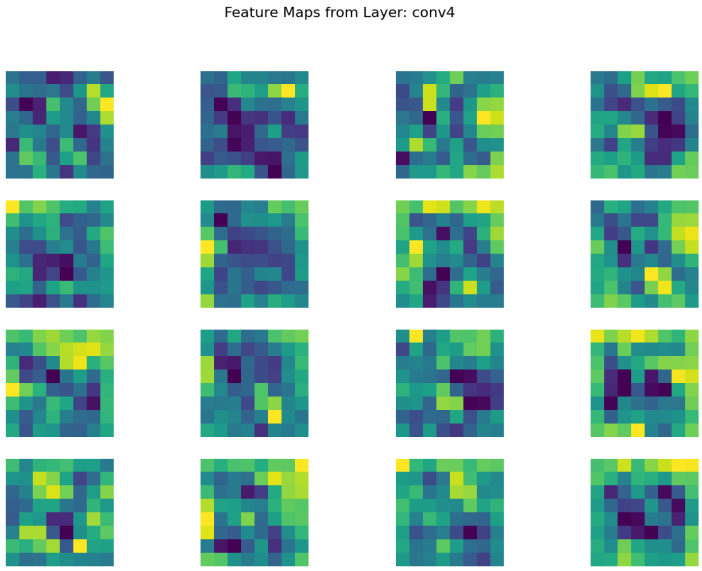

Figure B.3: Feature maps from the fourth convolutional layer, highlighting high-level feature extraction.