Contents

List of Figures

List of Tables

1 Introduction and Problem formulation

Machine learning has become increasingly important in solving image classification tasks, which require efficient and accurate models to identify and categorize images into predefined classes. This project focuses on comparing the performance of a classical machine learning model, the Decision Tree Classifier, with a state-of-the-art deep learning model, AlexNet, for a multi-class waste classification task.

The dataset used for this study is a COCO-style dataset containing annotations for seven distinct waste categories. The objective is to preprocess the dataset, train both models on the training set, evaluate their performance on the test set, and analyze their behavior across various metrics, such as accuracy, precision, recall, and F1-score.

The project involves:

1. **Data Preparation:** Splitting the dataset into training, validation, and testing subsets, followed by preprocessing operations such as cropping, resizing, normalization, and feature extraction (e.g., HOG for the Decision Tree).

2. **Model Training:** Implementing AlexNet and Decision Tree Classifier to train on the preprocessed data using different optimization strategies.

3. **Performance Evaluation:** Using confusion matrices and classification metrics to compare the models ability to classify waste into their respective categories.

This report highlights the strengths and weaknesses of each model and provides insights into the suitability of classical and deep learning approaches for image classification tasks.

2 Methodology

2.1 Data Loading and Processing

2.1.1 Training Set Splitting

The first thing to do is to split the COCO-style dataset into three different categories for different purposes:

- **Training Set:** Used to *train* the model by optimizing the weights based on the loss function.

- **Validation Set:** Evaluates the model *during training* to tune hyperparameters and monitor overfitting.

- **Testing Set:** Provides an evaluation of the models performance after training.

If one uses the same set, then the model will be biased towards the data it has seen before and will not generalize well to new data. The detailed split procedure is described in Appendix B.1.

2.1.2 Image Preprocessing

Before feeding the images in the dataset into the network, we need to preprocess the images. The following steps are taken:

- **Cropping:** Crop each image in the dataset according to a predefined box. Each image annotation includes a bounding box in the format of `[x, y, width, height]`. For example, If bbox = `[50, 30, 100, 150]`, the box starts at pixel (50, 30) and spans 100 pixels wide and 150 pixels tall.
- **Resizing:** Resize the cropped image to a fixed size. The size of the image is set to 192×192 pixels in our case.
- **Tensorization:** Convert the image into a tensor, which the neural network can process efficiently.
- **Normalization:** Translate each pixel values in all RGB channels based on a predefined mean value and normalize them within a number in $[0, 1]$, i.e.,

$$\text{Normalized Pixel} = \frac{\text{Pixel value} - \text{Mean}}{\text{Standard Deviation}}$$

This part is accomplished in `dataset.py` as shown in Appendix D.1. This is only for experiment purposes. In actual training process, the image preprocessing is done together with the network training as shown in Appendix D.3.

2.2 Network Implementation

AlexNet is used to implement the network. The architecture of AlexNet is shown in Figure 1. The detailed architecture and Python realization is shown in Appendix A and D.2.

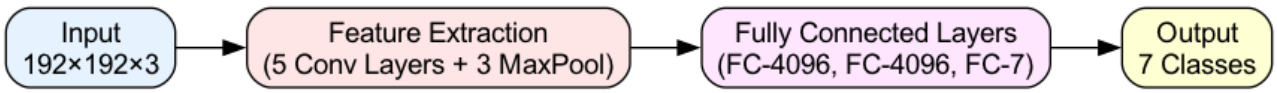

Figure 1: Overall architecture of the AlexNet

The 4 functional blocks in Figure 1 are explained as follows:

- **Input Layer:** Accept the input image of size 192×192. There are RGB three channels in total, so the input size is $192 \times 192 \times 3$.
- **Feature Extraction:** There are 5 convolutional layers and 3 max-pooling layers[1] in this block. The convolutional kernels extract particular features from the input image. The max-pooling layers reduce the spatial dimensions of the feature maps by taking the maximum value of the pixels in a certain region.
- **Fully Connected (FC) Layers:** The feature maps are flattened and fed into 3 fully connected layers, enabling the network to use the global context of the image for classification.
- **Output Layer:** The final layer has 7 neurons, corresponding to the predicted probability of a particular class, usually using the softmax function to convert logits to probabilities:

$$\text{Softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}.$$

[1]An adaptive-average-pooling layer and a flatten operation are added afterwards, ensuring a fixed-size 1-D output from the convolutional feature maps. See Appendix A for details.

2.3 Model Training

In order to train the model proposed in Section 2.2, we need to define the loss function and the optimization algorithm. All the training code is shown in Appendix D.3.

2.3.1 Loss Function

A loss function generally defines a metric in a particular set, which measures the "distance" between the predicted value and the actual label. We choose the cross-entropy loss function as the metric[2]. As an example, suppose the activation value of the output layer is a vector

$$\mathbf{y} = (\mathbb{P}_i)_{i=0}^{6}.$$

The actual label vector is

$$\mathbf{t} = \mathbf{1}_k,$$

indicating that the true class is k. The cross-entropy loss is defined as

$$\text{CrossEntropy}(\mathbf{y}, \mathbf{t}) := -\sum_{i=0}^{6} \mathbf{t}_i \log(\mathbb{P}_i) = -\log(\mathbb{P}_k).$$

The closer $\mathbb{P}_k$ is to 1, the smaller the loss is.

2.3.2 Optimization Algorithm

Having defined the loss function, we need to optimize the weights of the network to minimize the loss. We choose the Adaptive Moment Estimation (Adam) optimizer, which is better version of the stochastic gradient descent (SGD) algorithm. It combines the advantages of both Momentum and RMSProp optimizers. On the one hand, it automatically adjusts the learning rate for each parameter based on its historical gradient information. On the other hand, parameters that change frequently get smaller updates, while stable parameters get larger updates.

For momentum part, it computes an exponentially decaying average of past gradients (m_t) based on:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t,$$

where g_t is the gradient at time t and β_1 is the decay rate (commonly set to 0.9).

For RMSProp part, it computes an exponentially decaying average of past squared gradients (v_t):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2,$$

where β_2 controls the decay rate for the RMSProp term (commonly set to 0.999).

Adam then adjusts each parameter θ_t using both m_t (momentum) and v_t (adaptive learning rate):

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t},$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t,$$

where η is the learning rate and ϵ is a small constant to prevent division by zero (commonly 10^{-8}).

[2]Mathematically, the cross-entropy loss is not metric because of its non-symmetric property. However, it can be viewed as the length of a geodesic on a Riemannian manifold (plus an extra entropy constant).

3 Results and Performance Evaluation

We will mainly compare the performance of the AlexNet model with the decision tree classifier.

3.1 Confusion Matrix

A confusion matrix is a performance evaluation metric for classification problems. We provide the evaluation code[3] in Appendix D.4 for AlexNet. If a classifier performs well, the confusion matrix will have high values in the diagonal and low values elsewhere. As shown in Figure 2 and Figure 3, the AlexNet model has a better performance than the decision tree classifier.

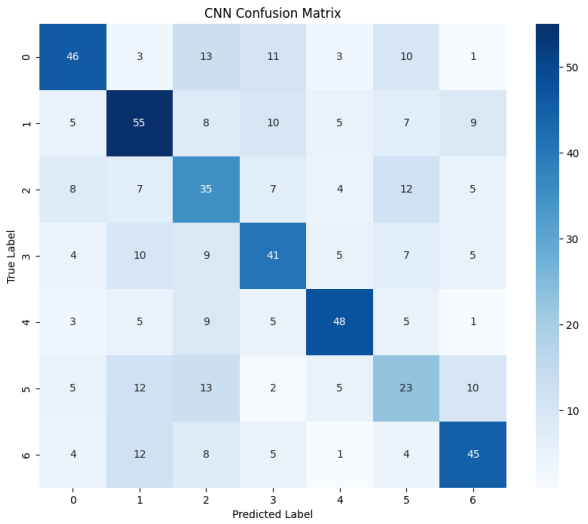

Figure 2: AlexNet Confusion Matrix

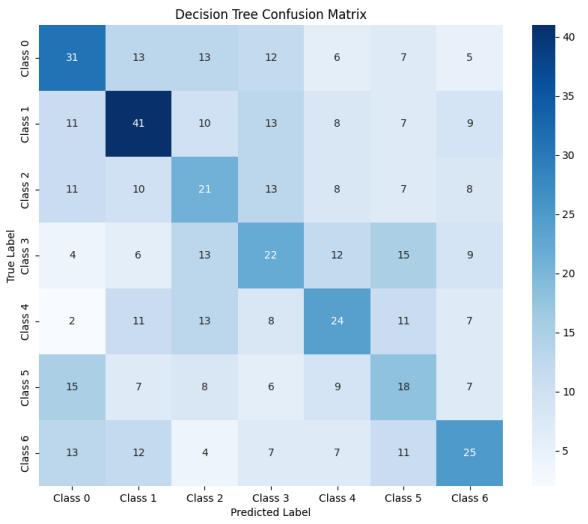

Figure 3: Decision Tree Confusion Matrix

3.2 Other Performance Metrics

We use accuracy, precision, recall, and F1 score as the performance metrics (which is embedded in a python library called `sklearn.metrics`). The comparison between the decision tree classifier and AlexNet is shown in Table 1. The mathematical definitions of these metrics are shown in Appendix C.

As shown in Table 1, the AlexNet model outperforms the decision tree classifier in all metrics. The reason is still under research, but it is believed that decision tree classifier relies on manually extracted features (HOG), which are limited in capturing diverse and abstract patterns. On the other hand, AlexNet can automatically learn features from the data (using those convolutional kernels) and thus can capture more complex patterns. Also, the decision tree classifier is prone to overfitting, while for AlexNet, the dropout layer and batch normalization layer can prevent overfitting to some extent.

For a closer look at the performance of the two methods in each class, we find that for Class 2 (plastic) and Class 5 (hazardous), both methods have a low F1 score. This is probably because these two categories are even ambiguous to human, or share similar features with other classes. For example, hazardous waste can be also plastic, metal or electronic (which is often the case). Therefore, in the process of labeling, the human labeler may also confused about the class of the waste.

[3]All codes for decision tree can be seen in the attachment or at https://github.com/Marcobisky/ml-project2.git

Table 1: Performance Comparison Between AlexNet and Decision Tree Classifier

Class	**Support**	**AlexNet**			**Decision Tree**		
		Precision	Recall	F1-Score	Precision	Recall	F1-Score
Class 0 (paper)	87	0.61	0.53	0.57	0.3563	0.3563	0.3563
Class 1 (metal)	99	0.53	0.56	0.54	0.4100	0.4141	0.4121
Class 2 (plastic)	78	0.37	0.45	0.40	0.2561	0.2692	0.2625
Class 3 (glass)	81	0.51	0.51	0.51	0.2716	0.2716	0.2716
Class 4 (food)	76	0.68	0.63	0.65	0.3243	0.3158	0.3200
Class 5 (hazardous)	70	0.34	0.33	0.33	0.2368	0.2571	0.2466
Class 6 (electronic)	79	0.59	0.57	0.58	0.3571	0.3165	0.3356
Accuracy	570		0.5140			0.3193	
Macro Avg	570	0.52	0.51	0.51	0.3160	0.3144	0.3149
Weighted Avg	570	0.52	0.51	0.52	0.3211	0.3193	0.3199

4 Conclusion

This project compares the Decision Tree Classifier and AlexNet in a multi-class image classification task. The evaluation results reveal that AlexNet significantly outperforms the Decision Tree Classifier across all performance metrics, achieving a test accuracy of 51.4% compared to the Decision Trees 31.9%.

AlexNets superior performance can be attributed to its ability to automatically learn hierarchical and abstract features through convolutional layers, making it robust to complex image patterns. On the other hand, the Decision Trees reliance on manually extracted HOG features limits its capacity to generalize, leading to poorer performance.

Also from Section 3.2, we learned that for a classifier, the classes should be well-defined and distinct, for a human classifier at least. Bad classification will result in badly-labelled dataset and thus a bad classifier.

In conclusion, while Decision Trees may be suitable for simple classification tasks with well-defined features, deep learning models like AlexNet are more effective for complex, high-dimensional image data. Future work could explore hybrid approaches, combining handcrafted features and deep learning to improve classification performance further.

A Detailed AlexNet Architecture

Table 2: AlexNet Architecture

Layer Type	**Parameters**	**Input Size**	**Output Size**
Input	-	$192 \times 192 \times 3$	$192 \times 192 \times 3$
Conv2D + ReLU	64 filters, 11×11, stride 4	$192 \times 192 \times 3$	$48 \times 48 \times 64$
MaxPool2D	3×3, stride 2	$48 \times 48 \times 64$	$24 \times 24 \times 64$
Conv2D + ReLU	192 filters, 5×5	$24 \times 24 \times 64$	$24 \times 24 \times 192$
MaxPool2D	3×3, stride 2	$24 \times 24 \times 192$	$12 \times 12 \times 192$
Conv2D + ReLU	384 filters, 3×3	$12 \times 12 \times 192$	$12 \times 12 \times 384$
Conv2D + ReLU	256 filters, 3×3	$12 \times 12 \times 384$	$12 \times 12 \times 256$
Conv2D + ReLU	256 filters, 3×3	$12 \times 12 \times 256$	$12 \times 12 \times 256$
MaxPool2D	3×3, stride 2	$12 \times 12 \times 256$	$6 \times 6 \times 256$
AdaptiveAvgPool2D	Output: 6×6	$6 \times 6 \times 256$	$6 \times 6 \times 256$
Flatten	-	$6 \times 6 \times 256$	9216
Fully Connected (FC)	4096 neurons	9216	4096
Fully Connected (FC)	4096 neurons	4096	4096
Fully Connected (FC)	7 neurons (output)	4096	7

B Detailed Manual for All Codes

B.1 Training Set Splitting

First, ensure you have the following directory structure:

```
GUID_FullName_ML-Code/
├── mlenv/ # Python environment
├── Project2
│   ├── 2024_uestc_autlab/
│   │   ├── data/
│   │   │   ├── coco/ # Original COCO dataset
│   │   │   │   ├── JPEGImages/
│   │   │   │   └── annotations.json
│   │   │   ├── data_coco_test/annotations.json # This will be generated
│   │   │   ├── data_coco_train/annotations.json # This will be generated
│   │   │   └── data_coco_valid/annotations.json # This will be generated
│   │   ├── split3.py
│   │   ├── train.py
│   │   ├── model.py
│   │   ├── dataset.py
│   │   └── evaluate.py
│   └── image/
│       └── confusion_matrix_final.png
└── Lab4/
    ├── coco/
    │   ├── annotations.json
    │   └── JPEGImages/
    ├── image/
    │   └── confusion_matrix_decision_tree.png
    └── 2024_uestc_autlab/
        ├── outputs/
        ├── utils/
        ├── dataset.py
        ├── ml_evaluate.py
        └── ml_model.py
```

In unix-like systems, you can run the following command to split the dataset:

```
cd 2024_uestc_autlab
source mlenv/bin/activate
python split3.py
```

This will generate `data_coco_train`, `data_coco_valid`, and `data_coco_test` directories with the corresponding annotations.

B.2 Image Preprocessing

After correctly changing the directories, run:

```
python dataset.py
```

The output will be:

```
loading annotations into memory...
Done (t=0.01s)
creating index...
index created!
ID: 0, Label: paper
ID: 1, Label: metal
ID: 2, Label: plastic
ID: 3, Label: glass
ID: 4, Label: food
ID: 5, Label: hazardous
ID: 6, Label: electronic
2642
```

B.3 Model Training

Directly run:

```
python train.py
```

About 30 epoches, the loss would be stable at around 0.2, the accuracy would be roughly 0.5.

B.4 Performance Evaluation

To evaluate the AlexNet model, run:

```
python evaluate.py
```

This will print all the performance metrics and the figure of the confusion matrix under `./Project2/image/confusion_matrix_final.png`.

C Performace Metric Definitions

C.1 Accuracy

Accuracy measures the proportion of correctly classified instances out of the total instances. It is defined as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}.$$

For a confusion matrix:

$$\text{Accuracy} = \frac{\sum_{i=1}^{C} \text{TP}_i}{\sum_{i=1}^{C} (\text{TP}_i + \text{FP}_i + \text{FN}_i)},$$

where

- C: Number of classes.
- TP_i: True Positives for class i.
- FP_i: False Positives for class i.
- FN_i: False Negatives for class i.

C.2 Precision

Precision measures the proportion of true positive predictions out of all predictions made for a particular class. It is defined as:

$$\text{Precision}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i},$$

where

- TP_i: True Positives for class i.
- FP_i: False Positives for class i.

For multi-class classification, weighted average precision is:

$$\text{Precision weighted avg} = \frac{\sum_{i=1}^{C} (\text{Precision}_i \cdot \text{Support}_i)}{\sum_{i=1}^{C} \text{Support}_i},$$

where Support_i is the number of true instances for class i.

C.3 Recall

Recall (or Sensitivity) measures the proportion of true positive predictions out of all actual instances for a particular class. It is defined as:

$$\text{Recall}_i = \frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i},$$

where

- TP_i: True Positives for class i.
- FN_i: False Negatives for class i.

For multi-class classification, weighted average recall is:

$$\text{Recall weighted avg} = \frac{\sum_{i=1}^{C} (\text{Recall}_i \cdot \text{Support}_i)}{\sum_{i=1}^{C} \text{Support}_i}.$$

C.4 F1 Score

The F1 Score is the harmonic mean of precision and recall, providing a balance between the two:

$$\text{F1}_i = 2 \cdot \frac{\text{Precision}_i \cdot \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}.$$

For multi-class classification, weighted average F1 score is:

$$\text{F1 weighted avg} = \frac{\sum_{i=1}^{C} (\text{F1}_i \cdot \text{Support}_i)}{\sum_{i=1}^{C} \text{Support}_i}$$

C.5 Support

For a class c , the support is given by:

$$\text{Support}_c = \sum_{i=1}^{N} \mathbf{1}(y_{\text{true},i} = c),$$

where

- N: Total number of samples in the dataset.
- $y_{\text{true},i}$: The true label for the i-th sample.
- $\mathbf{1}$(condition): Indicator function that equals 1 if the condition is true, and 0 otherwise.

D All Related Python Code

D.1 dataset.py

Listing 1: Experiment Code for Image Preprocessing

```python
import numpy as np
import torch
import torch.utils.data as data
from pycocotools.coco import COCO
import os

from PIL import Image
from torchvision.transforms import functional as F
from matplotlib import pyplot as plt


class MyCOCODataset(data.Dataset):
    def __init__(self, data_root, annofile, output_size=(192, 192)):
        self.data_root = data_root
        self.annofile = annofile
        self.coco = COCO(annofile)
        self.instance_ids = list(self.coco.anns.keys())
        self.instances = self.coco.anns
        self.output_size = output_size

    def __getitem__(self, index):
        id = self.instance_ids[index]
        ann = self.instances[id]
        imgid = ann["image_id"]
        # [x, y , w, h]
        bbox = ann["bbox"]
        img_file = self.coco.loadImgs([imgid])[0]["file_name"]
        img_file = img_file.replace('\\', '/')
        img = Image.open(os.path.join(self.data_root, img_file))

        # cutout the instance from bbox
        _bbox_int = list(map(int, bbox))
        img = img.crop(
            (
                _bbox_int[0],  # x1
                _bbox_int[1],  # y1
                _bbox_int[0] + _bbox_int[2],  # x2 = x1 + w
                _bbox_int[1] + _bbox_int[3],  # y2 = y1 + h
            )
        )
        # resize the iamge to corresponding size
        img = img.resize(self.output_size, Image.Resampling.BILINEAR)
        if img.mode == "L":
            img = img.convert("RGB")
```

```
45
46          # convert image to tensor
47          img = np.array(img)
48
49          # handle the annotation
50          category = ann["category_id"]
51          category = np.array(category, dtype=np.int64)
52
53          assert img.shape[0] == self.output_size[0]
54          assert img.shape[1] == self.output_size[1]
55          assert img.shape[2] == 3
56
57          return img, category
58
59      def __len__(self):
60          return len(self.instance_ids)
61
62      # utility function to print all categories
63      def print_all_categories(self):
64          for id, category in self.coco.cats.items():
65              print(f'ID: {id}, Label: {category["name"]}')
66
67
68  if __name__ == "__main__":
69      dataset = MyCOCODataset(
70          "./Lab4/coco",
71          "./Lab4/coco/annotations.json",
72          # "./Project2/2024_uestc_autlab/data/data_coco_test",
73          # "./Project2/2024_uestc_autlab/data/data_coco_test/annotations.json
    ",
74      )
75      dataset.print_all_categories()
76      print(len(dataset))
77      img, label = dataset[0]
78      plt.imshow(img)
```

D.2 model.py

Listing 2: AlexNet Structure

```
1  from functools import partial
2  from typing import Any, Optional
3
4  import torch
5  import torch.nn as nn
6
7  class AlexNet(nn.Module):
8      def __init__(self, num_classes: int = 1000, dropout: float = 0.5) ->
    None:
9          super().__init__()
```

```
        # NOTE: visual backbone
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )

        # [batch, 256, 6,6]
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(p=dropout),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x

if __name__ == "__main__":
    # test code
    model = AlexNet(num_classes=1000)
    x = torch.randn(1, 3, 192, 192)
    print(model(x).shape)
```

D.3 `train.py`

Listing 3: Training Code

```
# train.py
import torch
from torch import nn
```

```
import torch.optim as optim
from torch.utils.data import DataLoader
from dataset import MyCOCODataset
from model import AlexNet

# Hyperparameters
LEARNING_RATE = 3e-4
BATCH_SIZE = 32
NUM_CLASSES = 7
NUM_EPOCHS = 50
PATIENCE = 8
WEIGHT_DECAY = 1e-4

# Data normalization values
MEAN = torch.tensor([0.485, 0.456, 0.406]).view(1, 3, 1, 1)
STD = torch.tensor([0.229, 0.224, 0.225]).view(1, 3, 1, 1)

def normalize_batch(images, device):
    images = images.float() / 255.0
    images = images.permute(0, 3, 1, 2)
    images = (images - MEAN.to(device)) / STD.to(device)
    return images

def train_model(model, train_loader, val_loader, device):
    model = model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(
        model.parameters(),
        lr=LEARNING_RATE,
        weight_decay=WEIGHT_DECAY
    )
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, mode='max', factor=0.1, patience=3, verbose=True
    )

    best_accuracy = 0
    no_improvement = 0

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

        for images, labels in train_loader:
            images = normalize_batch(images, device)
            images, labels = images.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
```

```
            loss.backward()

            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
            optimizer.step()
            running_loss += loss.item()

        avg_train_loss = running_loss / len(train_loader)

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

        with torch.no_grad():
            for images, labels in val_loader:
                images = normalize_batch(images, device)
                images, labels = images.to(device), labels.to(device)

                outputs = model(images)
                _, predicted = torch.max(outputs, 1)

                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        accuracy = correct / total
        print(f'Epoch [{epoch+1}/{NUM_EPOCHS}] - '
              f'Loss: {avg_train_loss:.4f} - '
              f'Validation Accuracy: {accuracy:.4f}')

        scheduler.step(accuracy)

        if accuracy > best_accuracy:
            best_accuracy = accuracy
            no_improvement = 0
            torch.save({
                'epoch': epoch,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'accuracy': accuracy,
            }, './Project2/2024_uestc_autlab/outputs/best_model.pth')
        else:
            no_improvement += 1

        if no_improvement >= PATIENCE:
            print(f'Early stopping triggered after epoch {epoch+1}')
            break

if __name__ == "__main__":
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

```

```
    train_set = MyCOCODataset(
        "./Lab4/coco",
        "./Project2/2024_uestc_autlab/data/data_coco_train/annotations.json"
    ,
    )
    val_set = MyCOCODataset(
        "./Lab4/coco",
        "./Project2/2024_uestc_autlab/data/data_coco_valid/annotations.json"
    ,
    )

    train_loader = DataLoader(train_set, batch_size=BATCH_SIZE, shuffle=True
    )
    val_loader = DataLoader(val_set, batch_size=BATCH_SIZE)

    model = AlexNet(num_classes=NUM_CLASSES)
    train_model(model, train_loader, val_loader, device)
```

D.4 evaluate.py

Listing 4: Evaluation Code

```
# evaluate.py
import torch
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix, classification_report
from torch.utils.data import DataLoader
from dataset import MyCOCODataset
from model import AlexNet

def normalize_batch(images, device):
    MEAN = torch.tensor([0.485, 0.456, 0.406]).view(1, 3, 1, 1)
    STD = torch.tensor([0.229, 0.224, 0.225]).view(1, 3, 1, 1)
    images = images.float() / 255.0
    images = images.permute(0, 3, 1, 2)
    images = (images - MEAN.to(device)) / STD.to(device)
    return images

def evaluate_model(model_path, test_loader, device, num_classes=7):
    # Load model
    model = AlexNet(num_classes=num_classes)
    checkpoint = torch.load(model_path, map_location=device)
    model.load_state_dict(checkpoint['model_state_dict'])
    model = model.to(device)
    model.eval()

    # Collect predictions
    predictions = []
```

```
    ground_truth = []

    with torch.no_grad():
        for images, labels in test_loader:
            images = normalize_batch(images, device)
            images = images.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs, 1)
            predictions.extend(predicted.cpu().numpy())
            ground_truth.extend(labels.numpy())

    predictions = np.array(predictions)
    ground_truth = np.array(ground_truth)

    # Calculate metrics
    accuracy = (predictions == ground_truth).mean()
    cm = confusion_matrix(ground_truth, predictions)
    cr = classification_report(ground_truth, predictions)

    # Plot confusion matrix
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title('CNN Confusion Matrix')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.savefig('./Project2/image/confusion_matrix_final.png')
    plt.close()

    return accuracy, cm, cr

if __name__ == "__main__":
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # Load test dataset
    test_set = MyCOCODataset(
        "./Lab4/coco",
        "./Project2/2024_uestc_autlab/data/data_coco_test/annotations.json",
    )
    test_loader = DataLoader(test_set, batch_size=32)

    # Evaluate
    accuracy, conf_matrix, class_report = evaluate_model('./Project2/2024
   _uestc_autlab/outputs/best_model3.pth', test_loader, device)
    print(f"Final Test Accuracy: {accuracy:.4f}")
    print("\nClassification Report:")
    print(class_report)
    print("\nConfusion Matrix:")
    print(conf_matrix)
```