Comprehensive Analysis of Image Processing and Model Training Code for Waste Bin Classification

Juan Arevalo[a,1]

[a]SETAS; [b]Artificial Intelligence; [c]Centennial college; [d]Alumni

Jbaruz

***Abstract*—*This paper presents a comprehensive analysis of a code implementation for classifying images of waste bins as 'full' or 'empty'. The process involves key steps in image preprocessing, data augmentation, model training, evaluation, and prediction, leveraging advanced techniques in computer vision and artificial intelligence (AI). The code's functionality is detailed, its relationship to computer vision and AI is explored, and potential real-world applications are discussed. This paper aims to provide a clear understanding of the methodologies employed and their effectiveness in developing a robust image classification model.*

Keywords*—*Image Preprocessing, Data Augmentation, Convolutional Neural Network (CNN), Computer Vision, Artificial Intelligence (AI), Waste Management, Image Classification, Model Training, Model Evaluation, Real-World Applications

1. Introduction

Classification waste bins as 'full' or 'empty' is a practical problem that significantly benefits from advancements in computer vision and AI. This paper examines a code implementation designed to address this issue by utilizing a Convolutional Neural Network (CNN) for image classification. The goal is to develop a model that accurately identifies the state of waste bins, thereby optimizing waste collection processes and improving operational efficiency. This analysis covers the code's structure, functionality, and relevance to computer vision and AI.

2. Image Preprocessing and Data Augmentation

The *resize_images* function standardizes image sizes using the LANCZOS filter. This standardization is crucial in computer vision to ensure uniformity in input dimensions, simplifying the model training process [8]. Images are resized to a uniform size of 224x224 pixels.

```
def resize_images(directory_path, desired_size=(224, 224)):
    for filename in os.listdir(directory_path):
        if filename.endswith(".jpg") or filename.endswith(".png"):
            img_path = os.path.join(directory_path, filename)
            img = Image.open(img_path)
            img = img.resize(desired_size, Image.Resampling.LANCZOS)
            img.save(img_path)
```

Code 1. *Python code to resize images.*

The *rename_files* function renames image files in a specified directory with a base filename and a counter. This standardization is crucial for maintaining a consistent naming convention, simplifying file management and retrieval [10]. Files with .jpg and .png extensions are renamed sequentially, ensuring uniqueness by incrementing the counter even if a file with the new name already exists. This prevents naming conflicts and potential overwriting.

```
def rename_files(directory_path, base_filename):
    counter = 1
    for filename in os.listdir(directory_path):
        if filename.endswith(".jpg") or filename.endswith(".png"):
            new_filename = f"{base_filename}_{counter}.jpg"
            source = os.path.join(directory_path, filename)
            destination = os.path.join(directory_path, new_filename)

            # Check if the destination file already exists
            if not os.path.exists(destination):
                try:
                    os.rename(source, destination)
                    counter += 1
                except PermissionError:
                    print(f"Could not rename file: {source}. It might be in use by another process.")
            else:
                print(f"Cannot rename {source} to {destination}: destination file already exists.")
                counter += 1
```

Code 2. *Python code to rename image files.*

The *check_image_sizes* function checks and prints the dimensions of image files in a specified directory. This step is important for validating the consistency of image sizes within the dataset, which is essential for further processing or analysis [4]. The function iterates through the files, opens each image, and records its size, providing a clear overview of the image dimensions in the dataset.

```
def check_image_sizes(directory_path):
    # List to store the sizes of each image
    sizes = []
    for filename in os.listdir(directory_path):
        if filename.endswith((".jpg", ".png")):  # Check for jpg and png files
            img_path = os.path.join(directory_path, filename)
            with Image.open(img_path) as img:
                print(f"{filename}: {img.size}")  # Print filename and its size
                sizes.append(img.size)  # Add the size to the list
    return sizes
```

Code 3. *Python code to check image files.*

The *train_datagen* object, an instance of *ImageDataGenerator* is configured for data augmentation, enhancing the model's robustness by introducing variability in training data.

This configuration is crucial for improving the robustness and generalization of the model by introducing variability in the training data [6]. The parameters are set as follows: *rescale=1./255*: Normalizes the pixel values to the range [0, 1]. *rotation_range=45*: Increases rotation range to 45 degrees, enhancing the model's ability to recognize images from different angles. *width_shift_range=0.3* and *height_shift_range=0.3*: Increased shift ranges to 30%, providing more variability in image translations. *shear_range=0.3*: Increased to 30%, allowing for more shearing transformations. *zoom_range=0.3*: Increased to 30%, adding variability in zoom levels. *horizontal_flip=True*: Enables random horizontal flipping of images. *fill_mode='reflect'*: Changed from 'nearest' to 'reflect' for better edge quality in augmented images. *brightness_range=[0.5, 1.5]*: Varies the brightness of images to enhance robustness against lighting conditions. *channel_shift_range=150.0*: Randomly shifts the color channels, adding color variability. *validation_split=0.1*: Reserves 10% of the data for validation purposes.

```
    train_datagen = ImageDataGenerator(
        rescale=1. / 255,
        rotation_range=45,
        width_shift_range=0.3,
        height_shift_range=0.3,
        shear_range=0.3,
        zoom_range=0.3,
        horizontal_flip=True,
        fill_mode='reflect',
        brightness_range=[0.5, 1.5],
        channel_shift_range=150.0,
        validation_split=0.1
    )
```

Code 4. *Python code train_datagen.*

Information

This comprehensive data augmentation strategy ensures that the model is exposed to a wide range of variations, leading to better performance and generalization on unseen data [9].

3. Weight and Bias Application

The *wandb* library is integrated for tracking and managing machine learning experiments. This integration is crucial for optimizing hyperparameters, visualizing training progress, and maintaining reproducibility [5]. The script performs the following tasks:

3.1. Setup and Login

Verifies WandB installation and logs in using the API key from the environment variable. *api_key = os.getenv("WANDB_API_KEY")*: Retrieves the API key from the environment variable. If not set, raises an error.

3.2. Sweep Configuration

Defines a sweep configuration for hyperparameter tuning. This configuration includes various hyperparameters such as learning rate, batch size, epochs, and dropout rate, which are varied to find the optimal settings. *sweep_config*: Contains the method for the sweep, the metric to optimize, and the parameter values to explore.

3.3. Image Preprocessing

Resizes and renames images to ensure consistency and ease of processing. *resize_images(directory_path, desired_size=(224, 224))*: Resizes images to 224x224 pixels using the LANCZOS filter. *rename_files(directory_path, base_filename)*: Renames image files with a base filename and a counter to maintain a consistent naming convention.

3.4. Training Configuration

Defines the Convolutional Neural Network (CNN) model and sets up data augmentation parameters. The training process includes data augmentation to enhance the model's robustness and uses early stopping and model checkpointing to prevent overfitting. *train_datagen = ImageDataGenerator(...)*: Configures data augmentation parameters such as rescaling, rotation, shifting, shearing, zooming, flipping, and brightness adjustment.

3.5. Model Training

Uses the WandB callback to log training metrics and configurations. This ensures that all experiment details are captured and can be analyzed later. *wandb.init(config=sweep_config, project="waste_bin_prediction")*: Initializes the WandB run with the sweep configuration. *model.fit(..., callbacks=[..., WandbCallback(...)])*: Trains the model with the specified callbacks, including WandB for logging.

3.6. Prediction

Implements a function to make predictions on new images using the trained model. This function loads the image, preprocesses it, makes a prediction, and displays the image with the prediction label and confidence. *make_prediction(image_path)*: Loads and preprocesses the image, makes a prediction, and displays the result.

The script concludes by verifying the setup of the temporary directory and providing an example usage for making predictions on selected images.

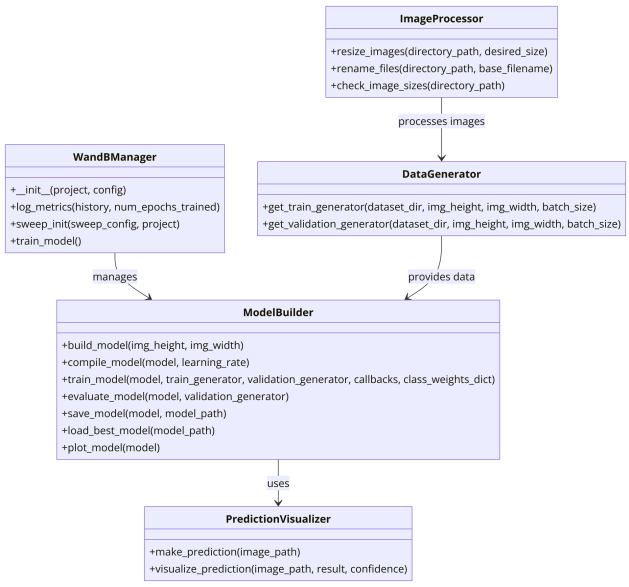

Figure 1. *UML Diagram*

4. Model Architecture: Sequential_15

The CNN model *"sequential_15"* performs image classification by progressively extracting features through convolutional layers, batch normalization, and max pooling. It flattens the output and passes it through dense layers for final classification. This model is important for accurately identifying patterns in images, making it suitable for complex image recognition tasks. Its structure ensures efficient learning and generalization [1].

Table 1. *Model: "sequential_15"*

Layer (type)	Output Shape	Param #
conv2d_60 (Conv2D)	(None, 148, 148, 32)	896
batch_normalization_60 (BatchNormalization)	(None, 148, 148, 32)	128
max_pooling2d_60 (MaxPooling2D)	(None, 74, 74, 32)	0
conv2d_61 (Conv2D)	(None, 72, 72, 64)	18496
batch_normalization_61 (BatchNormalization)	(None, 72, 72, 64)	256
max_pooling2d_61 (MaxPooling2D)	(None, 36, 36, 64)	0
conv2d_62 (Conv2D)	(None, 34, 34, 128)	73856
batch_normalization_62 (BatchNormalization)	(None, 34, 34, 128)	512
max_pooling2d_62 (MaxPooling2D)	(None, 17, 17, 128)	0
conv2d_63 (Conv2D)	(None, 15, 15, 256)	295168
batch_normalization_63 (BatchNormalization)	(None, 15, 15, 256)	1024
max_pooling2d_63 (MaxPooling2D)	(None, 7, 7, 256)	0
flatten_15 (Flatten)	(None, 12544)	0
dropout_15 (Dropout)	(None, 12544)	0
dense_30 (Dense)	(None, 512)	6423040
dense_31 (Dense)	(None, 1)	513
Total params: 6,813,889		
Trainable params: 6,812,929		
Non-trainable params: 960		

Convolutional Layer Formula

A convolutional layer operates by applying a kernel to the input image, described by the following formula:

$$(I * K)(x, y) = \sum_i \sum_j I(x + i, y + j) \cdot K(i, j) \tag{1}$$

where I is the input image, K is the kernel, and (x, y) are the coordinates of the output feature map.

Activation Function

Each convolutional layer is typically followed by a nonlinear activation function, such as ReLU:

$$\text{ReLU}(x) = \max(0, x) \tag{2}$$

Batch Normalization

Batch normalization normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation:

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{3}$$

where μ_B is the batch mean and σ_B^2 is the batch variance.

Pooling Layer

The max pooling operation can be defined as:

$$P(x, y) = \max_{0 \le i < f} \max_{0 \le j < f} I(x + i, y + j) \tag{4}$$

where f is the size of the pooling filter.

Dense Layer

The dense layer applies the following transformation:

$$y = Wx + b \tag{5}$$

where W is the weight matrix, x is the input vector, and b is the bias vector.

> **Note**
>
> *The combination of these layers allows the CNN to learn hierarchical features from the input images, making it a powerful tool for image classification tasks.*

5. Make Prediction

The script uses TensorFlow and PIL to load and preprocess an image, make a prediction using a pre-trained model, and visualize the result. The `make_prediction` function loads the image, scales it, and predicts if the image is "empty" or "full." The `visualize_prediction` function displays the image with a text label indicating the prediction and confidence level. The prediction process is initiated through a file dialog to select the image.

```python
from PIL import Image, ImageDraw, ImageFont
import os
import numpy as np
from tensorflow.keras.preprocessing import image
from tkinter import filedialog
from tkinter import Tk

def make_prediction(image_path):
    img = image.load_img(image_path, target_size=(img_height, img_width))
    img_array = image.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0)
    img_array /= 255.0
    prediction = model.predict(img_array)
    if prediction[0][0] > 0.5:
        return "empty", prediction[0][0]
    else:
        return "full", 1 - prediction[0][0]

def visualize_prediction(image_path, result, confidence):
    img = Image.open(image_path)
    draw = ImageDraw.Draw(img)
    font = ImageFont.load_default()
    text = f"{result} ({confidence*100:.0f}%)"
    color = "red" if result == "full" else "green"
    draw.rectangle([10, 10, 110, 110], outline=color, width=0)
    draw.text((20, 20), text, fill=color, font=font, font_size=20)
    img.show()

root = Tk()
root.withdraw()
image_path = filedialog.askopenfilename()

result, confidence = make_prediction(image_path)
print(f"Prediction: {result} (Confidence: {confidence*100:.2f}%)")
visualize_prediction(image_path, result, confidence)
```

Code 5. *Python code make_prediction and visualize_prediction.*

5.1. Predictions results

The results show image predictions of waste bins being either "empty" or "full" with confidence percentages. The model effectively identifies and labels the status of each bin.

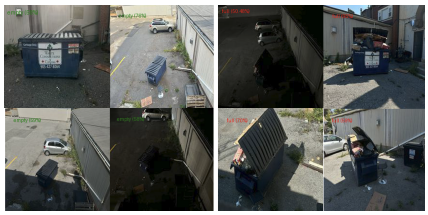

Figure 2. *Waste bins results*

6. Results from Weight and Bias

Weights and biases are crucial parameters in neural networks, learned during training. They determine the strength and direction of the input features' influence on predictions. Proper tuning of these parameters is essential for model accuracy and generalization. Monitoring weight and bias results helps in understanding model behavior, ensuring it captures relevant patterns without overfitting or underfitting.

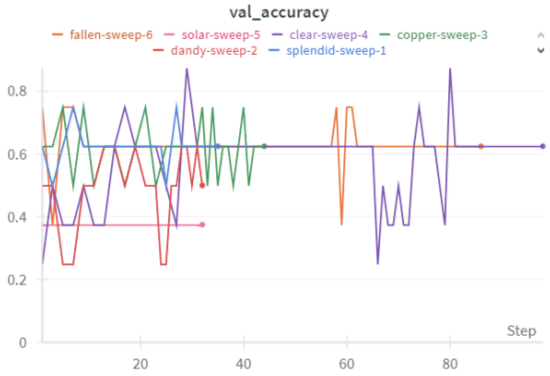

Figure 3. *Validation Accuracy*

The plot shows the validation accuracy for various hyperparameter configurations over multiple steps. Each line represents a different sweep configuration. The fluctuations indicate the sensitivity of validation accuracy to different hyperparameters. Observing these trends helps identify stable and high-performing configurations. Consistent accuracy across steps suggests robust parameter settings, while significant variability indicates areas for further tuning.

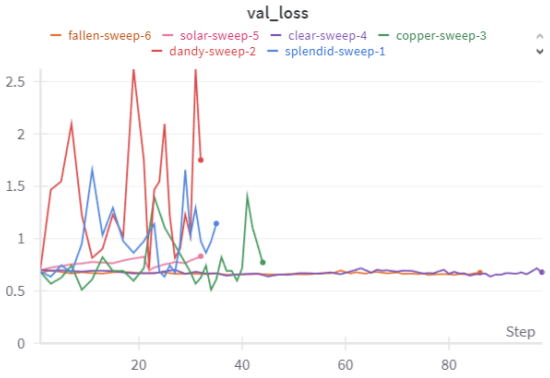

Figure 4. *Validation Loss*

The plot shows the validation loss for various hyperparameter configurations over multiple steps. Each line represents a different sweep configuration. The fluctuations in validation loss highlight the model's sensitivity to different hyperparameters. Lower and more stable validation loss indicates better generalization and model performance. High and fluctuating loss suggests overfitting or poor parameter settings. This analysis aids in identifying optimal hyperparameters for minimizing validation loss and enhancing model robustness.

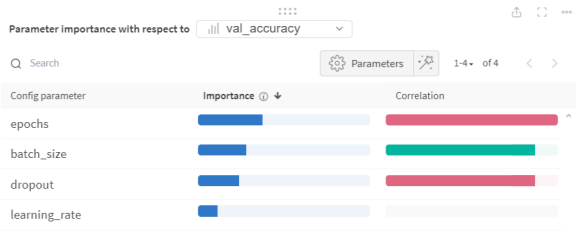

Figure 5. *Parameter importance - Value Accuracy*

The diagram illustrates the importance and correlation of various hyperparameters with respect to validation accuracy. The parameters analyzed are epochs, batch size, dropout, and learning rate.

- **Epochs**: Highly correlated with validation accuracy, indicating that increasing the number of epochs generally improves accuracy.
- **Batch Size**: Positively correlated, suggesting that larger batch sizes might enhance accuracy.
- **Dropout**: Shows a negative correlation, implying that higher dropout rates might reduce accuracy.
- **Learning Rate**: Minimal correlation, indicating it has less impact on validation accuracy within the tested range.

This analysis is crucial for optimizing model performance by fine-tuning these hyperparameters to achieve better accuracy.

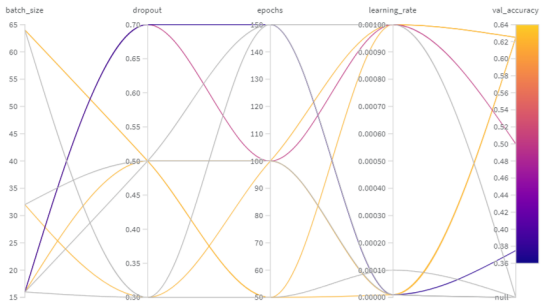

Figure 6. *Parameter importance - Value Accuracy*

The parallel coordinates plot visualizes the relationships between hyperparameters (batch size, dropout, epochs, learning rate) and validation accuracy. It highlights how different combinations of these parameters affect model performance. For example, higher epochs and moderate learning rates tend to improve accuracy, while extreme values of dropout negatively impact performance. This plot is essential for understanding and fine-tuning hyperparameters to optimize the model's accuracy.

7. Results

The dataset contains 81 images, with 41 labeled as "full" and 40 as "empty," all resized to 224x224 pixels. During training, the model's accuracy is 55.56%, with a validation accuracy of 62.5%. The training stops early due to no improvement in validation loss. The best validation loss achieved is 0.61973. The model's final prediction for a test image is "full" with an 89.35% confidence.

These results demonstrate predictive capability but also highlight areas for potential improvement. To enhance performance, consider increasing the dataset size, experimenting with more complex models, and using cross-validation. Additionally, further diversifying data augmentation techniques or employing synthetic data generation could improve model robustness.

Conclusion and Industry Applications

The provided code demonstrates the use of a Convolutional Neural Network (CNN) for classifying images of waste bins as "full" or "empty." The model leverages data augmentation and hyperparameter tuning to improve its accuracy and generalization. The use of libraries such as TensorFlow, PIL, and WandB enhances the efficiency and effectiveness of the training and evaluation processes.

Conclusion

The model achieves a validation accuracy of 62.5% with an early stopping mechanism to prevent overfitting. The resizing and consistent labeling of images ensure uniform input dimensions, which is crucial for effective model training. While the model shows some predictive capability, further hyperparameter tuning and additional data may enhance its performance.

Future Enhancements and Broader Impact

Ongoing developments in AI, such as more sophisticated neural network architectures and advanced data augmentation techniques, have the potential to further optimize waste management systems. For instance, the use of Generative Adversarial Networks (GANs) for data augmentation can improve model robustness by creating synthetic training data [3]. Additionally, advancements in real-time data processing and the integration of Internet of Things (IoT) devices can enhance the practical application of the model, leading to more efficient and responsive waste collection strategies [2]. Expanding the dataset with more diverse images and applying transfer learning from pre-trained models on larger datasets, such as ImageNet [7], can significantly boost model accuracy and robustness.

Industry Applications

Computer vision techniques, like the one implemented in this code, have various industry applications:

- **Waste Management**: Automated monitoring of waste bin status to optimize collection routes and schedules.
- **Retail**: Inventory management through real-time monitoring of stock levels on shelves.
- **Healthcare**: Monitoring patient activity and bed occupancy in hospitals.
- **Security**: Surveillance systems to detect unauthorized access or monitor crowd density.
- **Mining**: Detecting the fallout of Ground Engaging Tools (GET) in shovels to prevent plant shutdowns and damage to crushers.
- **Mining**: Monitoring ore grade by analyzing visual characteristics of the material on conveyor belts.
- **Mining**: Ensuring safety by detecting the presence of personnel or vehicles in hazardous areas.
- **Mining**: Inspecting haul roads for maintenance needs by identifying potholes and debris.

Implementing Video Live Processing

To implement this model for live video processing instead of static images, consider the following steps:

- **Video Capture**: Use a video capture library such as OpenCV to access the camera feed in real-time.
- **Frame Extraction**: Extract individual frames from the video feed at regular intervals.
- **Preprocessing**: Apply the same preprocessing steps (resizing, normalization) to each extracted frame.
- **Prediction**: Use the trained model to predict the status for each frame.
- **Visualization**: Overlay predictions on the video frames and display the annotated video in real-time.

These modifications enable the deployment of the model in dynamic environments, providing continuous monitoring and real-time decision-making capabilities.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[2] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions", *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[3] I. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, "Generative adversarial nets", *Advances in neural information processing systems*, vol. 27, pp. 2672–2680, 2014.

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", *arXiv preprint arXiv:1409.1556*, 2014.

[5] F. Chollet *et al.*, "Keras: The python deep learning library", *ascl*, ascl–1806, 2015.

[6] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", *International conference on machine learning*, pp. 448–456, 2015.

[7] O. Russakovsky, J. Deng, H. Su, *et al.*, "Imagenet large scale visual recognition challenge", *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[8] C. Szegedy, W. Liu, Y. Jia, *et al.*, "Going deeper with convolutions", *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[9] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition", *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.