Image Classification using Hand-crafted Features, Neural Networks, and CNNs

Module: ECS8053 – Computer Vision
2025 Spring

Name: Imama Jawad
Student ID: [40462364]

Queen's University Belfast

February 3, 2025

Contents

1 Introduction

Image classification is a crucial problem in computer vision, enabling machines to recognize objects, scenes, and patterns within images. The effectiveness of different approaches to image classification varies based on the complexity of the dataset, the features used, and the learning methodology.

The three primary approaches explored in this study are:

- **Hand-crafted Features**: Traditional computer vision methods rely on manually designed features, such as SIFT and ORB, to extract relevant information from images. These methods are computationally efficient but often struggle with complex datasets.
- **Neural Networks**: Fully connected deep learning models that learn abstract representations directly from data, reducing the need for manual feature engineering but requiring large amounts of labeled training data.
- **Convolutional Neural Networks (CNNs)**: Specialized deep learning architectures designed to process image data efficiently, leveraging convolutional layers to capture spatial hierarchies and intricate patterns.

This report explores these approaches using the TinyImageNet100 dataset, assessing their strengths and limitations. We further analyze why CNNs outperform neural networks and why neural networks surpass hand-crafted features in terms of accuracy and feature learning capability.

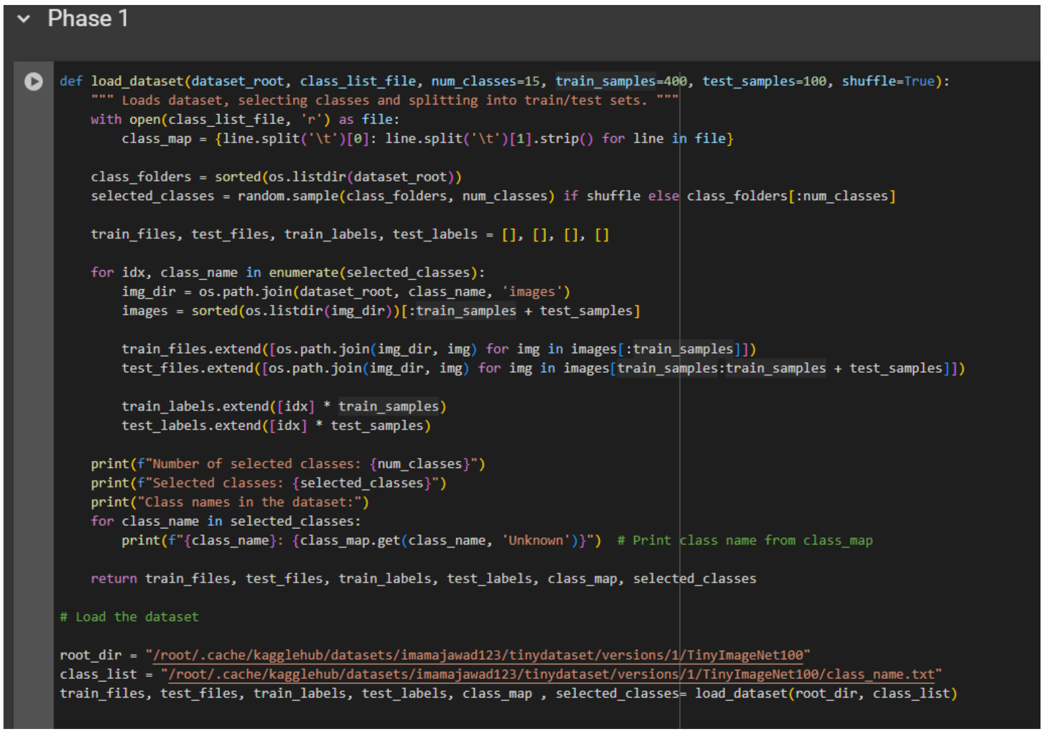
2 Methodology

2.1 Dataset Preparation

The TinyImageNet100 dataset contains 100 object classes, each with 500 images of size 64x64. A subset of 15 classes was selected randomly for this study. The dataset was split into:

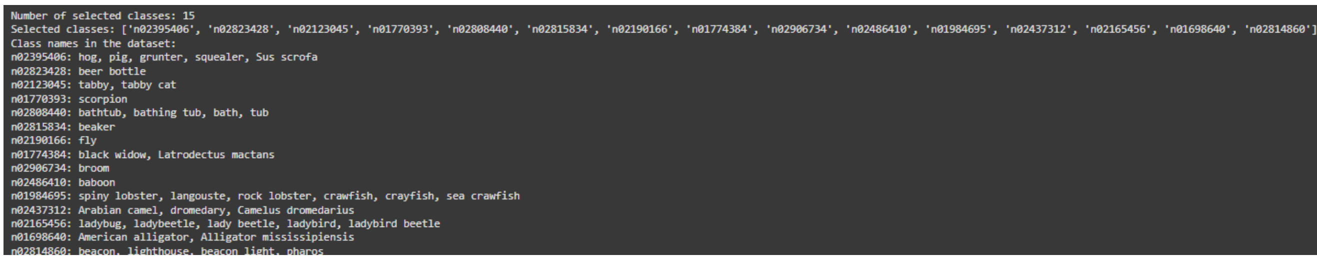
- 400 images per class for training.
- 100 images per class for testing.

By incorporating class labels from `class_names.txt`, we were able to map predictions to their respective categories, providing meaningful insights into misclassifications and model performance across different object classes.



```
def load_dataset(dataset_root, class_list_file, num_classes=15, train_samples=400, test_samples=100, shuffle=True):
    """ Loads dataset, selecting classes and splitting into train/test sets. """
    with open(class_list_file, 'r') as file:
        class_map = {line.split('\t')[0]: line.split('\t')[1].strip() for line in file}

    class_folders = sorted(os.listdir(dataset_root))
    selected_classes = random.sample(class_folders, num_classes) if shuffle else class_folders[:num_classes]

    train_files, test_files, train_labels, test_labels = [], [], [], []

    for idx, class_name in enumerate(selected_classes):
        img_dir = os.path.join(dataset_root, class_name, 'images')
        images = sorted(os.listdir(img_dir))[:train_samples + test_samples]

        train_files.extend([os.path.join(img_dir, img) for img in images[:train_samples]])
        test_files.extend([os.path.join(img_dir, img) for img in images[train_samples:train_samples + test_samples]])

        train_labels.extend([idx] * train_samples)
        test_labels.extend([idx] * test_samples)

    print(f"Number of selected classes: {num_classes}")
    print(f"Selected classes: {selected_classes}")
    print("Class names in the dataset:")
    for class_name in selected_classes:
        print(f"{class_name}: {class_map.get(class_name, 'Unknown')}")  # Print class name from class_map

    return train_files, test_files, train_labels, test_labels, class_map, selected_classes

# Load the dataset

root_dir = "/root/.cache/kagglehub/datasets/imamajawad123/tinydataset/versions/1/TinyImageNet100"
class_list = "/root/.cache/kagglehub/datasets/imamajawad123/tinydataset/versions/1/TinyImageNet100/class_name.txt"
train_files, test_files, train_labels, test_labels, class_map , selected_classes= load_dataset(root_dir, class_list)
```

Figure 1: Phase 1 Code



```
Number of selected classes: 15
Selected classes: ['n02395406', 'n02823428', 'n02123045', 'n01770393', 'n02808440', 'n02815834', 'n02190166', 'n01774384', 'n02906734', 'n02486410', 'n01984695', 'n02437312', 'n02165456', 'n01698640', 'n02814860']
Class names in the dataset:
n02395406: hog, pig, grunter, squealer, Sus scrofa
n02823428: beer bottle
n02123045: tabby, tabby cat
n01770393: scorpion
n02808440: bathtub, bathing tub, bath, tub
n02815834: beaker
n02190166: fly
n01774384: black widow, Latrodectus mactans
n02906734: broom
n02486410: baboon
n01984695: spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish
n02437312: Arabian camel, dromedary, Camelus dromedarius
n02165456: ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle
n01698640: American alligator, Alligator mississipiensis
n02814860: beacon, lighthouse, beacon light, pharos
```

Figure 2: Output- Phase 1

Class ID	**Category Name**
n02395406	Hog, pig, grunter, squealer, Sus scrofa
n02823428	Beer bottle
n02123045	Tabby, tabby cat
n01770393	Scorpion
n02808440	Bathtub, bathing tub, bath, tub
n02815834	Beaker
n02190166	Fly
n01774384	Black widow, Latrodectus mactans
n02906734	Broom
n02486410	Baboon
n01894695	Spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish
n02437312	Arabian camel, dromedary, Camelus dromedarius
n02165456	Ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle
n01698640	American alligator, Alligator mississippiensis
n02814860	Beacon, lighthouse, beacon light, pharos

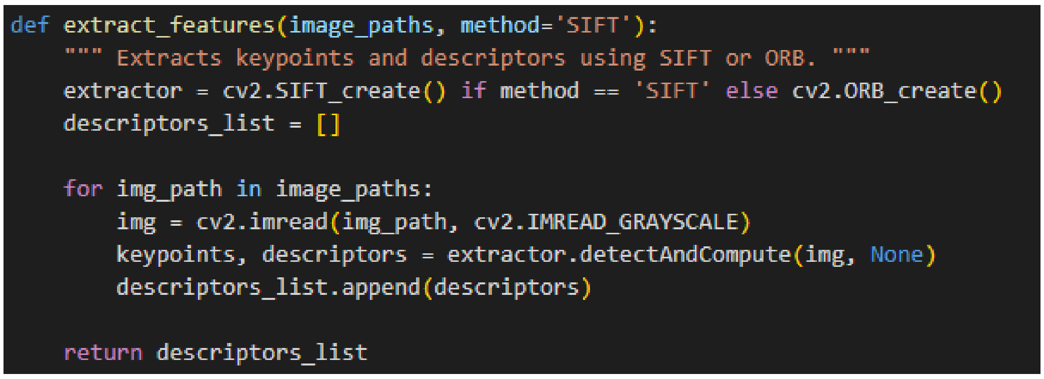
Table 1: Selected Categories and their Class IDs

2.2 Hand-crafted Feature Approach

Hand-crafted features were extracted using:

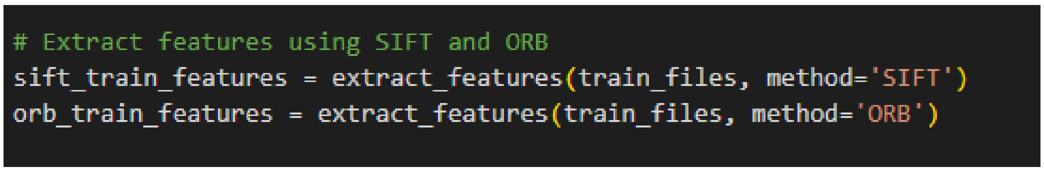
- **SIFT**: Detects key points and computes scale-invariant and rotation-invariant descriptors to encode image details.

- **ORB**: A computationally efficient alternative to SIFT that provides binary descriptors, reducing memory usage while maintaining performance.



```python
def extract_features(image_paths, method='SIFT'):
    """ Extracts keypoints and descriptors using SIFT or ORB. """
    extractor = cv2.SIFT_create() if method == 'SIFT' else cv2.ORB_create()
    descriptors_list = []

    for img_path in image_paths:
        img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
        keypoints, descriptors = extractor.detectAndCompute(img, None)
        descriptors_list.append(descriptors)

    return descriptors_list
```

Figure 3: Phase 2- Feature Extraction

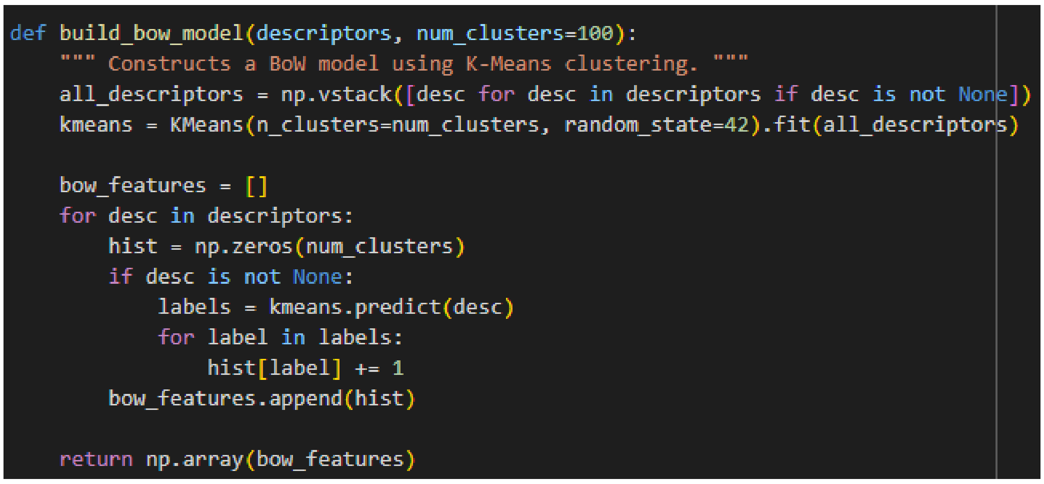


```python
# Extract features using SIFT and ORB
sift_train_features = extract_features(train_files, method='SIFT')
orb_train_features = extract_features(train_files, method='ORB')
```

Figure 4: Phase 2- Feature Extraction Call

Mid-level representations included:

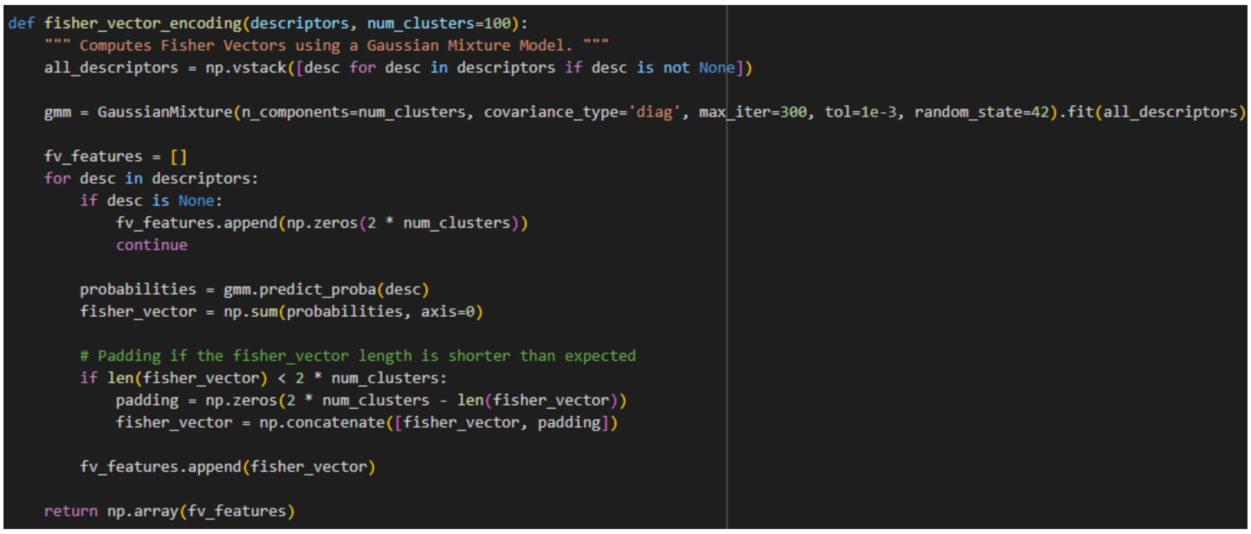
- **Bag of Words (BoW)**: Visual words were generated using K-means clustering on extracted descriptors.



```python
def build_bow_model(descriptors, num_clusters=100):
    """ Constructs a BoW model using K-Means clustering. """
    all_descriptors = np.vstack([desc for desc in descriptors if desc is not None])
    kmeans = KMeans(n_clusters=num_clusters, random_state=42).fit(all_descriptors)

    bow_features = []
    for desc in descriptors:
        hist = np.zeros(num_clusters)
        if desc is not None:
            labels = kmeans.predict(desc)
            for label in labels:
                hist[label] += 1
        bow_features.append(hist)

    return np.array(bow_features)
```

Figure 5: Phase 2- BOW Implementation

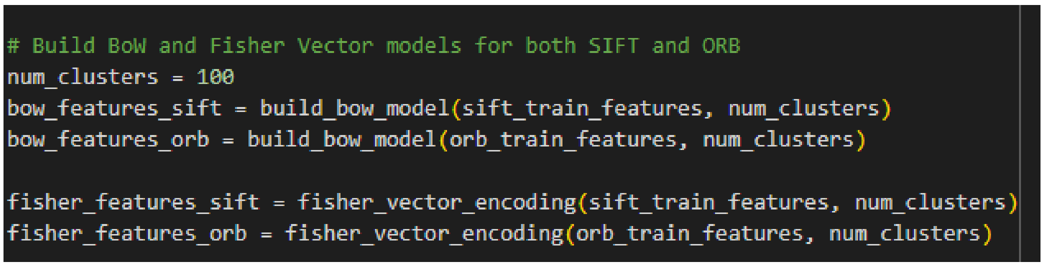
- **Fisher Vectors**: Utilized Gaussian Mixture Models (GMM) to encode image features in a high-dimensional space for improved classification accuracy.



```python
def fisher_vector_encoding(descriptors, num_clusters=100):
    """ Computes Fisher Vectors using a Gaussian Mixture Model. """
    all_descriptors = np.vstack([desc for desc in descriptors if desc is not None])

    gmm = GaussianMixture(n_components=num_clusters, covariance_type='diag', max_iter=300, tol=1e-3, random_state=42).fit(all_descriptors)

    fv_features = []
    for desc in descriptors:
        if desc is None:
            fv_features.append(np.zeros(2 * num_clusters))
            continue

        probabilities = gmm.predict_proba(desc)
        fisher_vector = np.sum(probabilities, axis=0)

        # Padding if the fisher_vector length is shorter than expected
        if len(fisher_vector) < 2 * num_clusters:
            padding = np.zeros(2 * num_clusters - len(fisher_vector))
            fisher_vector = np.concatenate([fisher_vector, padding])

        fv_features.append(fisher_vector)

    return np.array(fv_features)
```

Figure 6: Phase 2- Fisher Vector

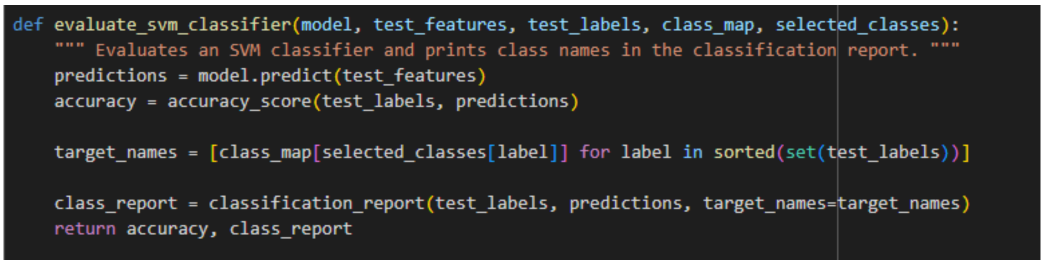
In both cases, the number of clusters was set to 100. For the BoW model, K-Means clustering was used to cluster the descriptors into visual words, while for Fisher Vectors, a Gaussian Mixture Model (GMM) with 100 components was employed to capture the underlying distributions of the features.



```
# Build BoW and Fisher Vector models for both SIFT and ORB
num_clusters = 100
bow_features_sift = build_bow_model(sift_train_features, num_clusters)
bow_features_orb = build_bow_model(orb_train_features, num_clusters)

fisher_features_sift = fisher_vector_encoding(sift_train_features, num_clusters)
fisher_features_orb = fisher_vector_encoding(orb_train_features, num_clusters)
```

Figure 7: Phase 2- Fisher/Bow Call

Classification was performed using a linear Support Vector Machine (SVM) trained on the extracted feature representations.

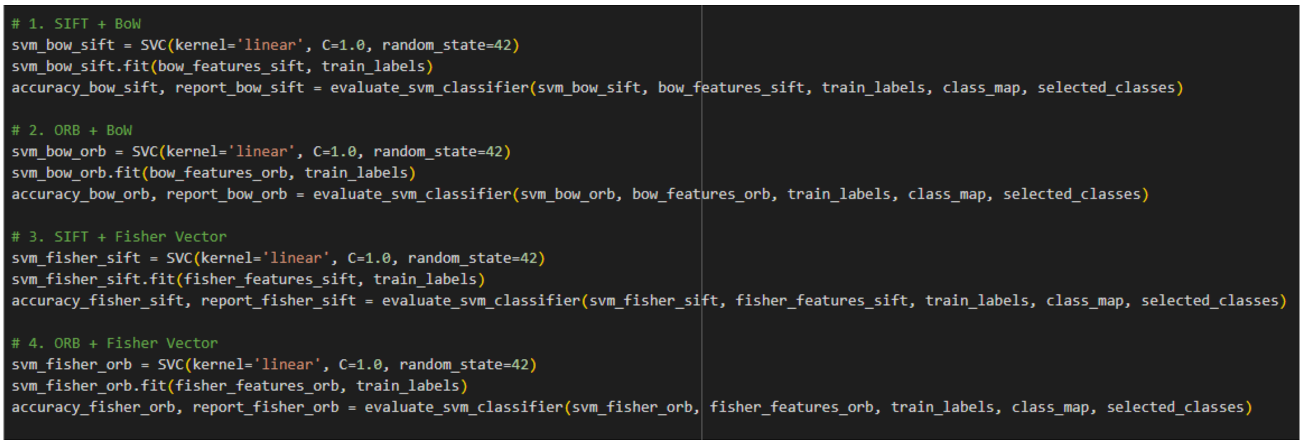


```
def evaluate_svm_classifier(model, test_features, test_labels, class_map, selected_classes):
    """ Evaluates an SVM classifier and prints class names in the classification report. """
    predictions = model.predict(test_features)
    accuracy = accuracy_score(test_labels, predictions)

    target_names = [class_map[selected_classes[label]] for label in sorted(set(test_labels))]

    class_report = classification_report(test_labels, predictions, target_names=target_names)
    return accuracy, class_report
```

Figure 8: Phase 2- SVM Evaluation

In this experiment, I aimed to evaluate and compare the four combinations of hand-crafted feature extraction methods for object recognition.
The following Models were explored:

- **SIFT + BoW**
- **ORB + BoW**
- **SIFT + Fisher Vector**
- **ORB + Fisher Vector**



```
# 1. SIFT + BoW
svm_bow_sift = SVC(kernel='linear', C=1.0, random_state=42)
svm_bow_sift.fit(bow_features_sift, train_labels)
accuracy_bow_sift, report_bow_sift = evaluate_svm_classifier(svm_bow_sift, bow_features_sift, train_labels, class_map, selected_classes)

# 2. ORB + BoW
svm_bow_orb = SVC(kernel='linear', C=1.0, random_state=42)
svm_bow_orb.fit(bow_features_orb, train_labels)
accuracy_bow_orb, report_bow_orb = evaluate_svm_classifier(svm_bow_orb, bow_features_orb, train_labels, class_map, selected_classes)

# 3. SIFT + Fisher Vector
svm_fisher_sift = SVC(kernel='linear', C=1.0, random_state=42)
svm_fisher_sift.fit(fisher_features_sift, train_labels)
accuracy_fisher_sift, report_fisher_sift = evaluate_svm_classifier(svm_fisher_sift, fisher_features_sift, train_labels, class_map, selected_classes)

# 4. ORB + Fisher Vector
svm_fisher_orb = SVC(kernel='linear', C=1.0, random_state=42)
svm_fisher_orb.fit(fisher_features_orb, train_labels)
accuracy_fisher_orb, report_fisher_orb = evaluate_svm_classifier(svm_fisher_orb, fisher_features_orb, train_labels, class_map, selected_classes)
```

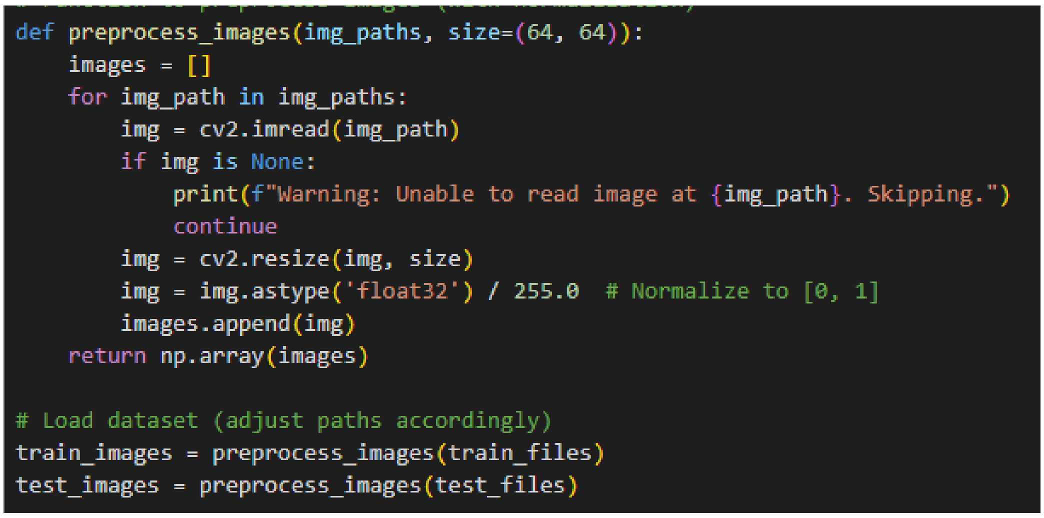
Figure 9: Phase 2 - Model Combinations

2.3 Neural Network Approach

A Simple fully connected neural network (MLP) without Convolution was implemented with the following configuration:

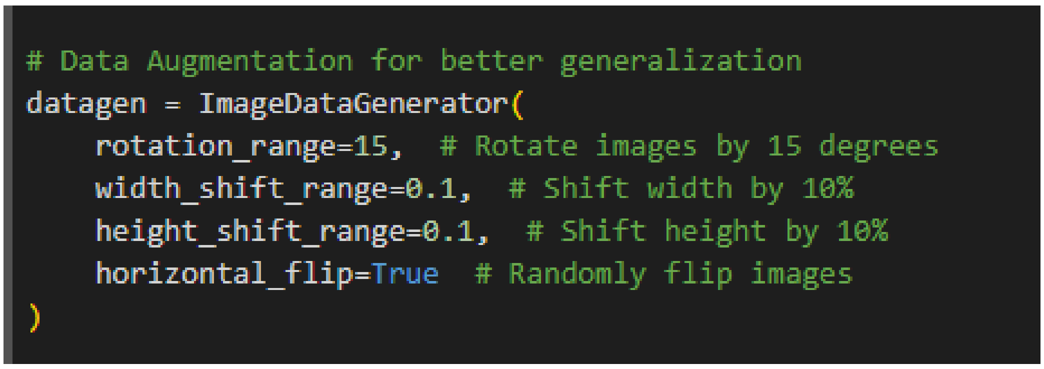
- Input layer: Flattened image pixels of size 64x64, normalized to the range [0, 1].
- Three hidden layers with 1024, 512, and 256 neurons, each using ReLU activation and Batch Normalization for improved convergence.
- Dropout regularization with rates of 0.4, 0.3, and 0.3 in the respective hidden layers to prevent overfitting.
- Softmax activation function in the output layer for multi-class classification.
- The AdamW optimizer with a learning rate of 0.001 and weight decay for regularization, along with sparse categorical cross-entropy loss function.

Training was conducted for 50 epochs with a batch size of 64. Data augmentation techniques such as rotation, width/height shifting, and horizontal flipping were applied to improve generalization. Early stopping was used with a patience of 10 epochs, and learning rate reduction was employed during training for better convergence. Hyperparameter tuning was performed to optimize the learning rate and dropout rates.



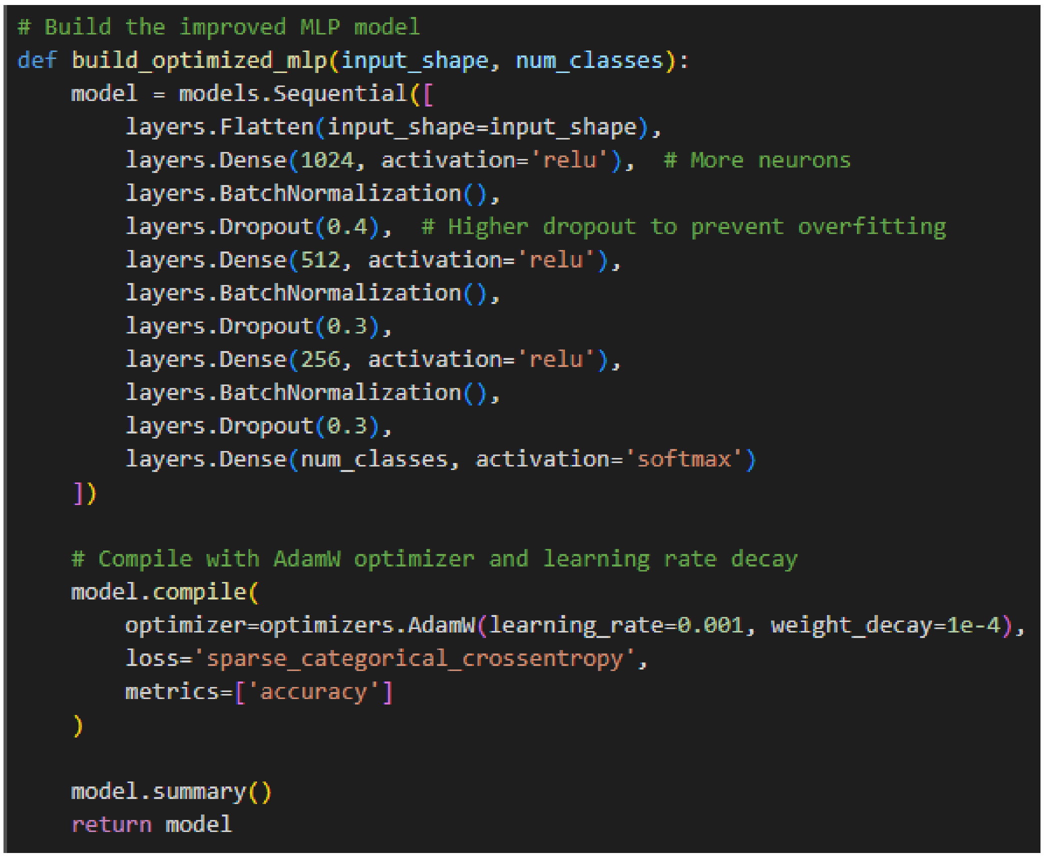
```python
def preprocess_images(img_paths, size=(64, 64)):
    images = []
    for img_path in img_paths:
        img = cv2.imread(img_path)
        if img is None:
            print(f"Warning: Unable to read image at {img_path}. Skipping.")
            continue
        img = cv2.resize(img, size)
        img = img.astype('float32') / 255.0  # Normalize to [0, 1]
        images.append(img)
    return np.array(images)

# Load dataset (adjust paths accordingly)
train_images = preprocess_images(train_files)
test_images = preprocess_images(test_files)
```

Figure 10: Phase 3 - Preprocessing



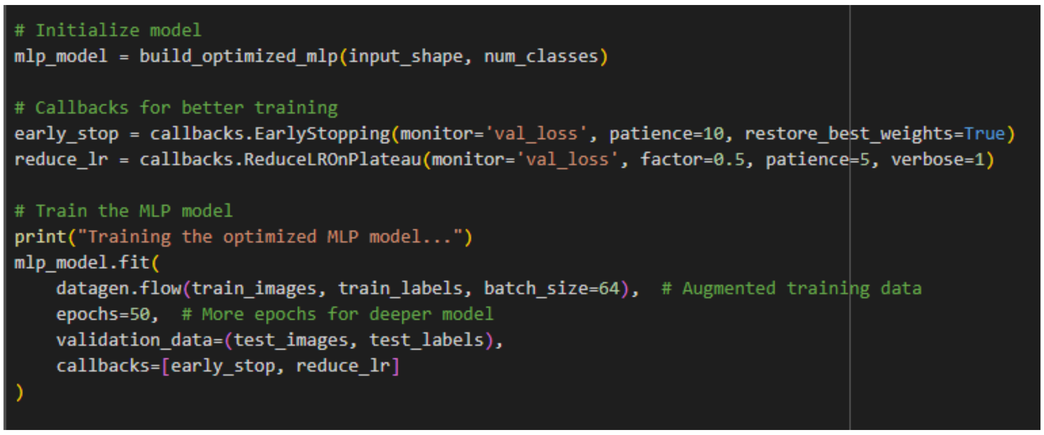
```python
# Data Augmentation for better generalization
datagen = ImageDataGenerator(
    rotation_range=15,  # Rotate images by 15 degrees
    width_shift_range=0.1,  # Shift width by 10%
    height_shift_range=0.1,  # Shift height by 10%
    horizontal_flip=True  # Randomly flip images
)
```

Figure 11: Phase 3 - Data Augmentation



```python
# Build the improved MLP model
def build_optimized_mlp(input_shape, num_classes):
    model = models.Sequential([
        layers.Flatten(input_shape=input_shape),
        layers.Dense(1024, activation='relu'),  # More neurons
        layers.BatchNormalization(),
        layers.Dropout(0.4),  # Higher dropout to prevent overfitting
        layers.Dense(512, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.3),
        layers.Dense(256, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.3),
        layers.Dense(num_classes, activation='softmax')
    ])

    # Compile with AdamW optimizer and learning rate decay
    model.compile(
        optimizer=optimizers.AdamW(learning_rate=0.001, weight_decay=1e-4),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    model.summary()
    return model
```

Figure 12: Phase 3 - Neural Network Model Structure



```python
# Initialize model
mlp_model = build_optimized_mlp(input_shape, num_classes)

# Callbacks for better training
early_stop = callbacks.EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
reduce_lr = callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, verbose=1)

# Train the MLP model
print("Training the optimized MLP model...")
mlp_model.fit(
    datagen.flow(train_images, train_labels, batch_size=64),  # Augmented training data
    epochs=50,  # More epochs for deeper model
    validation_data=(test_images, test_labels),
    callbacks=[early_stop, reduce_lr]
)
```

Figure 13: Phase 3- Initialization, Early Stopping Plus Model Fitting

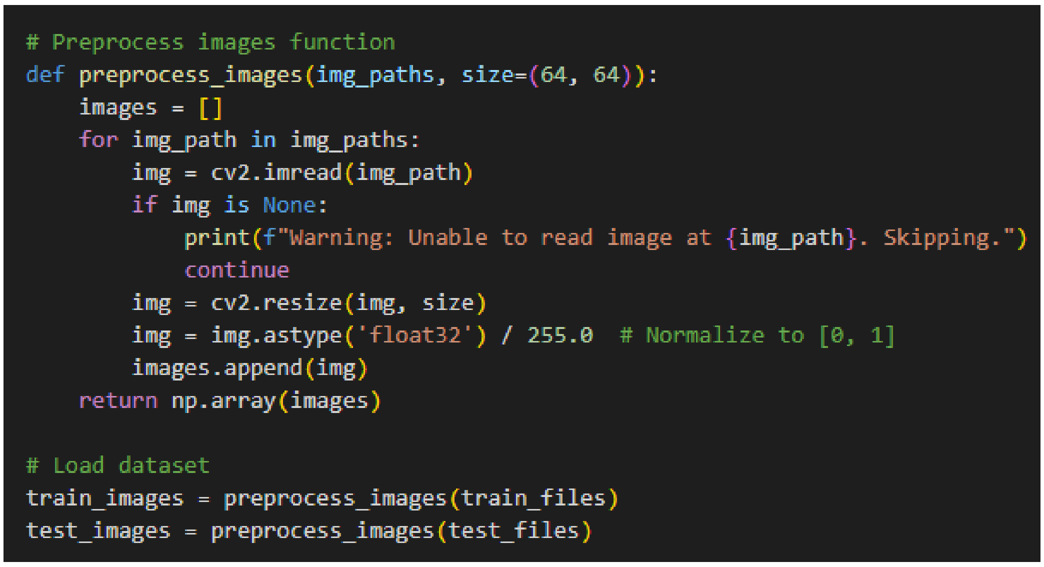
2.4 Convolutional Neural Network Approach

The CNN architecture includes the following key components:

- Three convolutional layers with increasing filter sizes (32, 64, 128), each followed by max-pooling.
- Fully connected layers with 256 neurons and dropout (0.5) to prevent overfitting.
- Batch normalization applied after each convolutional layer to stabilize learning and improve convergence speed.
- Data augmentation techniques (rotation, width/height shift, and horizontal flip) applied during training to improve model robustness.

The model architecture and the training process are outlined in detail as follows:

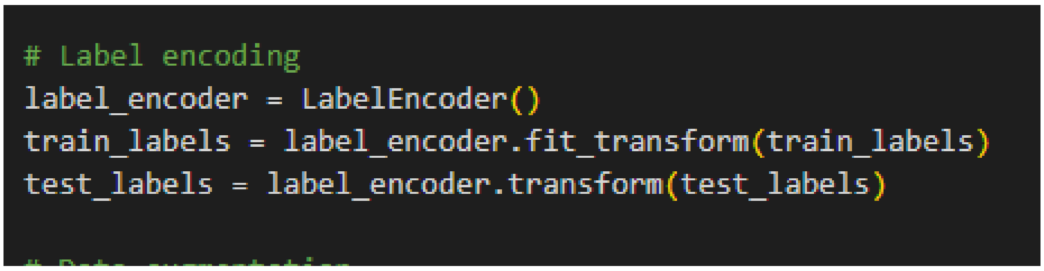
Preprocess images function:



```
# Preprocess images function
def preprocess_images(img_paths, size=(64, 64)):
    images = []
    for img_path in img_paths:
        img = cv2.imread(img_path)
        if img is None:
            print(f"Warning: Unable to read image at {img_path}. Skipping.")
            continue
        img = cv2.resize(img, size)
        img = img.astype('float32') / 255.0  # Normalize to [0, 1]
        images.append(img)
    return np.array(images)

# Load dataset
train_images = preprocess_images(train_files)
test_images = preprocess_images(test_files)
```

Figure 14: Phase 4- Preprocessing

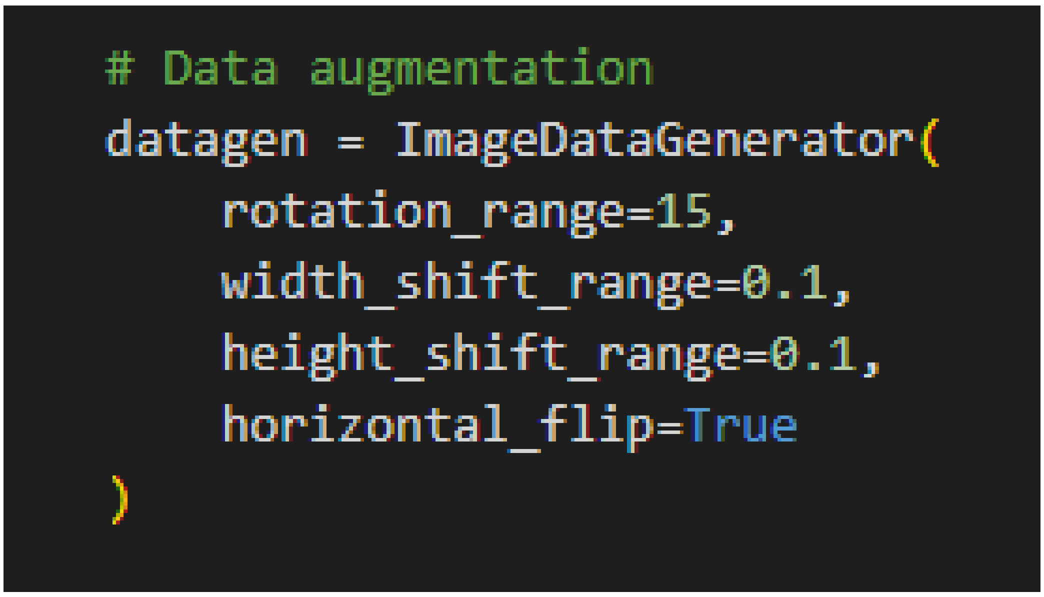
Label encoding:



```
# Label encoding
label_encoder = LabelEncoder()
train_labels = label_encoder.fit_transform(train_labels)
test_labels = label_encoder.transform(test_labels)
```

Figure 15: Phase 4- Label Encoding

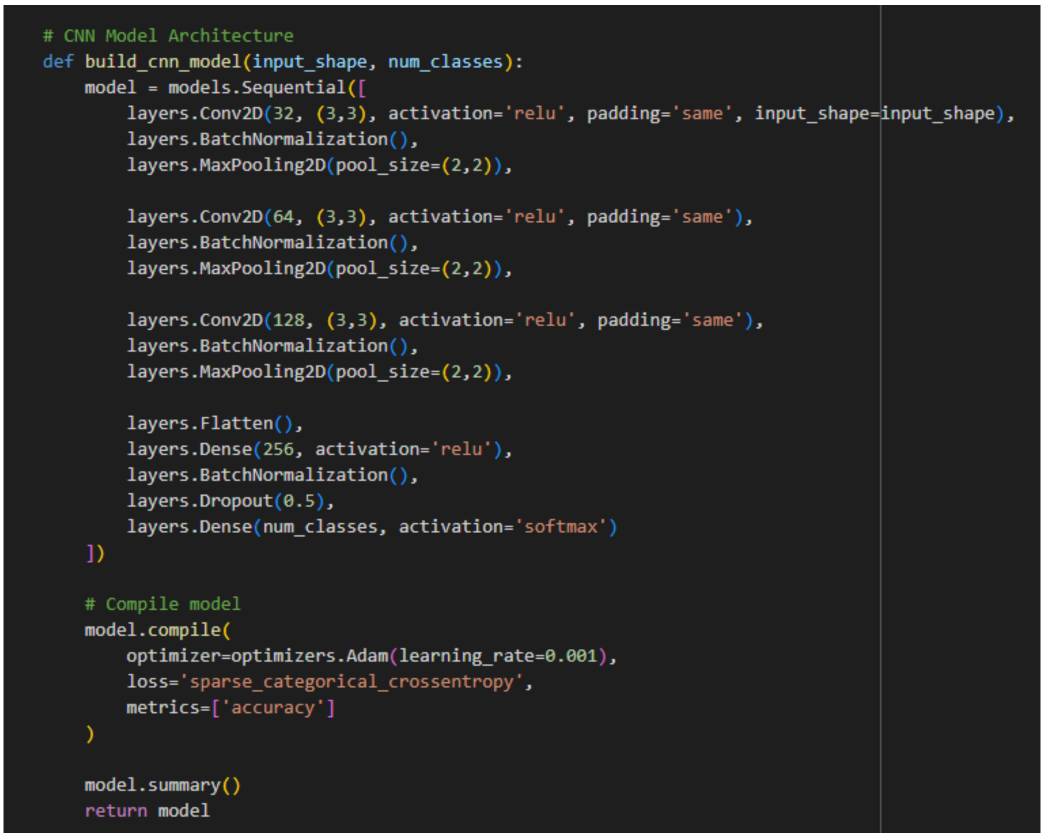
Data augmentation:



```
# Data augmentation
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)
```

Figure 16: Phase 4- Data Augmentation

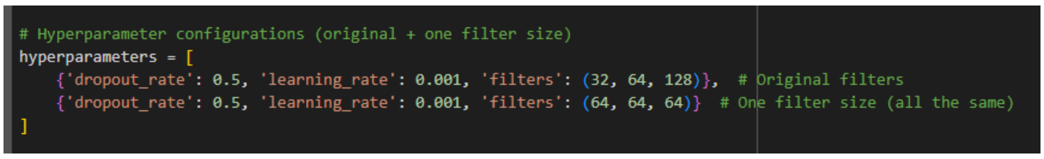
CNN Model Architecture (Keras):



```python
# CNN Model Architecture
def build_cnn_model(input_shape, num_classes):
    model = models.Sequential([
        layers.Conv2D(32, (3,3), activation='relu', padding='same', input_shape=input_shape),
        layers.BatchNormalization(),
        layers.MaxPooling2D(pool_size=(2,2)),

        layers.Conv2D(64, (3,3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D(pool_size=(2,2)),

        layers.Conv2D(128, (3,3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D(pool_size=(2,2)),

        layers.Flatten(),
        layers.Dense(256, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation='softmax')
    ])

    # Compile model
    model.compile(
        optimizer=optimizers.Adam(learning_rate=0.001),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    model.summary()
    return model
```

Figure 17: Phase 4- Model Architecture

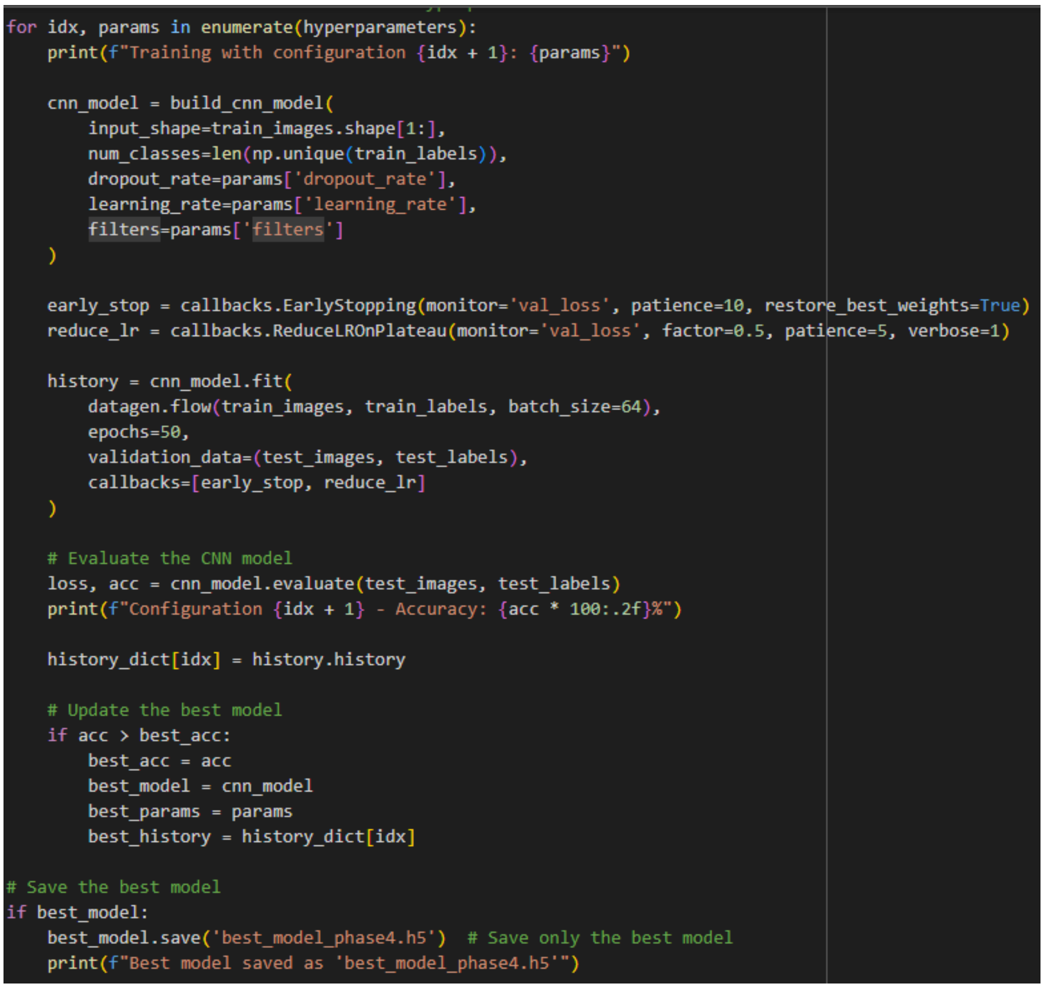
Hyperparameter configurations: Tried Different filter sizes in Hyperparameters.



```python
# Hyperparameter configurations (original + one filter size)
hyperparameters = [
    {'dropout_rate': 0.5, 'learning_rate': 0.001, 'filters': (32, 64, 128)},  # Original filters
    {'dropout_rate': 0.5, 'learning_rate': 0.001, 'filters': (64, 64, 64)}  # One filter size (all the same)
]
```

Figure 18: Phase 4- HyperParameter Tuning

Training and evaluation with hyperparameter tuning:



```python
for idx, params in enumerate(hyperparameters):
    print(f"Training with configuration {idx + 1}: {params}")

    cnn_model = build_cnn_model(
        input_shape=train_images.shape[1:],
        num_classes=len(np.unique(train_labels)),
        dropout_rate=params['dropout_rate'],
        learning_rate=params['learning_rate'],
        filters=params['filters']
    )

    early_stop = callbacks.EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
    reduce_lr = callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, verbose=1)

    history = cnn_model.fit(
        datagen.flow(train_images, train_labels, batch_size=64),
        epochs=50,
        validation_data=(test_images, test_labels),
        callbacks=[early_stop, reduce_lr]
    )

    # Evaluate the CNN model
    loss, acc = cnn_model.evaluate(test_images, test_labels)
    print(f"Configuration {idx + 1} - Accuracy: {acc * 100:.2f}%")

    history_dict[idx] = history.history

    # Update the best model
    if acc > best_acc:
        best_acc = acc
        best_model = cnn_model
        best_params = params
        best_history = history_dict[idx]

# Save the best model
if best_model:
    best_model.save('best_model_phase4.h5')  # Save only the best model
    print(f"Best model saved as 'best_model_phase4.h5'")
```

Figure 19: Phase 4 - Training/ Evaluation

3 Results

To evaluate the performance, the model's training and validation loss curves were plotted, showing the loss reduction over epochs. The test accuracy was recorded which indicated how each model performed.

Additionally, the confusion matrix was made to show how well a model performed across the classes, and the classification report showed precision, recall, and F1-scores. Furthermore results from different Hyper parameters were recorded if applicable.

3.1 Phase 2: Handcrafted Features

Model 1: SIFT + BoW (Test Data)

Class	**Precision**	**Recall**	**F1-Score**	**Support**
hog, pig, grunter, squealer, Sus scrofa	0.07	0.09	0.08	100
beer bottle	0.07	0.08	0.07	100
tabby, tabby cat	0.08	0.06	0.07	100
scorpion	0.05	0.05	0.05	100
bathtub, bathing tub, bath, tub	0.12	0.13	0.12	100
beaker	0.07	0.08	0.07	100
fly	0.09	0.12	0.10	100
black widow, Latrodectus mactans	0.12	0.17	0.14	100
broom	0.11	0.12	0.12	100
baboon	0.11	0.06	0.08	100
spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish	0.05	0.04	0.05	100
Arabian camel, dromedary, Camelus dromedarius	0.09	0.07	0.08	100
ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle	0.03	0.03	0.03	100
American alligator, Alligator mississipiensis	0.06	0.04	0.05	100
beacon, lighthouse, beacon light, pharos	0.10	0.09	0.09	100
Accuracy			0.08	1500
Macro avg	0.08	0.08	0.08	1500
Weighted avg	0.08	0.08	0.08	1500

Model 1: SIFT + BoW (Train Data)

Class	**Precision**	**Recall**	**F1-Score**	**Support**
hog, pig, grunter, squealer, Sus scrofa	0.27	0.24	0.25	400
beer bottle	0.28	0.33	0.31	400
tabby, tabby cat	0.27	0.28	0.28	400
scorpion	0.31	0.30	0.31	400
bathtub, bathing tub, bath, tub	0.27	0.36	0.31	400
beaker	0.30	0.32	0.31	400
fly	0.24	0.21	0.22	400
black widow, Latrodectus mactans	0.40	0.47	0.43	400
broom	0.29	0.28	0.29	400
baboon	0.28	0.29	0.28	400
spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish	0.36	0.40	0.38	400
Arabian camel, dromedary, Camelus dromedarius	0.28	0.19	0.23	400
ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle	0.37	0.26	0.30	400
American alligator, Alligator mississipiensis	0.32	0.31	0.32	400
beacon, lighthouse, beacon light, pharos	0.30	0.30	0.30	400
Accuracy			0.30	6000
Macro avg	0.30	0.30	0.30	6000
Weighted avg	0.30	0.30	0.30	6000

Model 2: ORB + BoW (Test Data)

Class	**Precision**	**Recall**	**F1-Score**	**Support**
hog, pig, grunter, squealer, Sus scrofa	0.05	0.01	0.02	100
beer bottle	0.07	0.01	0.02	100
tabby, tabby cat	0.19	0.10	0.13	100
scorpion	0.10	0.03	0.05	100
bathtub, bathing tub, bath, tub	0.07	0.82	0.13	100
beaker	0.00	0.00	0.00	100
fly	0.00	0.00	0.00	100
black widow, Latrodectus mactans	0.12	0.02	0.03	100
broom	0.00	0.00	0.00	100
baboon	0.10	0.03	0.05	100
spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish	0.16	0.05	0.08	100
Arabian camel, dromedary, Camelus dromedarius	0.03	0.01	0.02	100
ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle	0.04	0.01	0.02	100
American alligator, Alligator mississipiensis	0.14	0.03	0.05	100
beacon, lighthouse, beacon light, pharos	0.00	0.00	0.00	100
Accuracy			0.07	1500
Macro avg	0.07	0.07	0.04	1500
Weighted avg	0.07	0.07	0.04	1500

Model 2: ORB + BoW (Train Data)

Class	**Precision**	**Recall**	**F1-Score**	**Support**
hog, pig, grunter, squealer, Sus scrofa	0.23	0.04	0.08	400
beer bottle	0.18	0.02	0.04	400
tabby, tabby cat	0.20	0.06	0.09	400
scorpion	0.23	0.07	0.11	400
bathtub, bathing tub, bath, tub	0.08	0.90	0.15	400
beaker	0.00	0.00	0.00	400
fly	0.00	0.00	0.00	400
black widow, Latrodectus mactans	0.12	0.02	0.03	400
broom	0.00	0.00	0.00	400
baboon	0.10	0.02	0.03	400
spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish	0.16	0.05	0.08	400
Arabian camel, dromedary, Camelus dromedarius	0.03	0.01	0.02	400
ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle	0.04	0.01	0.02	400
American alligator, Alligator mississipiensis	0.14	0.03	0.05	400
beacon, lighthouse, beacon light, pharos	0.00	0.00	0.00	400
Accuracy			0.07	6000
Macro avg	0.07	0.07	0.04	6000
Weighted avg	0.07	0.07	0.04	6000

3.2 Model 3: SIFT + Fisher Vector (Test Set)

2*Class	Classification Report			2*Support
	Precision	Recall	F1-Score	
hog, pig, grunter, squealer, Sus scrofa	0.09	0.10	0.09	100
beer bottle	0.04	0.07	0.05	100
tabby, tabby cat	0.18	0.04	0.07	100
scorpion	0.11	0.09	0.10	100
bathtub, bathing tub, bath, tub	0.10	0.15	0.12	100
beaker	0.03	0.01	0.01	100
fly	0.11	0.07	0.09	100
black widow, Latrodectus mactans	0.10	0.15	0.12	100
broom	0.07	0.09	0.08	100
baboon	0.09	0.14	0.11	100
spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish	0.09	0.08	0.08	100
Arabian camel, dromedary, Camelus dromedarius	0.11	0.09	0.10	100
ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle	0.06	0.07	0.07	100
American alligator, Alligator mississipiensis	0.04	0.02	0.03	100
beacon, lighthouse, beacon light, pharos	0.08	0.06	0.07	100
accuracy			0.08	1500
macro avg	0.09	0.08	0.08	1500
weighted avg	0.09	0.08	0.08	1500

3.3 Model 3: SIFT + Fisher Vector (Training Set)

2*Class	Classification Report			2*Support
	Precision	Recall	F1-Score	
hog, pig, grunter, squealer, Sus scrofa	0.24	0.26	0.25	400
beer bottle	0.26	0.24	0.25	400
tabby, tabby cat	0.28	0.30	0.29	400
scorpion	0.27	0.30	0.29	400
bathtub, bathing tub, bath, tub	0.27	0.36	0.31	400
beaker	0.24	0.28	0.26	400
fly	0.26	0.20	0.23	400
black widow, Latrodectus mactans	0.41	0.47	0.44	400
broom	0.27	0.28	0.28	400
baboon	0.30	0.28	0.29	400
spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish	0.35	0.36	0.36	400
Arabian camel, dromedary, Camelus dromedarius	0.30	0.22	0.26	400
ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle	0.29	0.14	0.19	400
American alligator, Alligator mississipiensis	0.33	0.31	0.32	400
beacon, lighthouse, beacon light, pharos	0.27	0.30	0.28	400
accuracy			0.29	6000
macro avg	0.29	0.29	0.29	6000
weighted avg	0.29	0.29	0.29	6000

3.4 Model 4: ORB + Fisher Vector (Test Set)

2*Class	Classification Report			2*Support
	Precision	Recall	F1-Score	
hog, pig, grunter, squealer, Sus scrofa	0.00	0.00	0.00	100
beer bottle	0.00	0.00	0.00	100
tabby, tabby cat	0.00	0.00	0.00	100
scorpion	0.00	0.00	0.00	100
bathtub, bathing tub, bath, tub	0.07	0.98	0.12	100
beaker	0.00	0.00	0.00	100
fly	0.00	0.00	0.00	100
black widow, Latrodectus mactans	0.00	0.00	0.00	100
broom	0.00	0.00	0.00	100
baboon	0.00	0.00	0.00	100
spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish	0.00	0.00	0.00	100
Arabian camel, dromedary, Camelus dromedarius	0.00	0.00	0.00	100
ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle	0.00	0.00	0.00	100
American alligator, Alligator mississipiensis	0.00	0.00	0.00	100
beacon, lighthouse, beacon light, pharos	0.00	0.00	0.00	100
accuracy			0.07	1500
macro avg	0.00	0.07	0.01	1500
weighted avg	0.00	0.07	0.01	1500

3.5 Model 4: ORB + Fisher Vector (Training Set)

2*Class	Classification Report			2*Support
	Precision	Recall	F1-Score	
hog, pig, grunter, squealer, Sus scrofa	0.00	0.00	0.00	400
beer bottle	0.00	0.00	0.00	400
tabby, tabby cat	0.22	0.01	0.01	400
scorpion	0.14	0.05	0.07	400
bathtub, bathing tub, bath, tub	0.08	0.88	0.14	400
beaker	0.18	0.01	0.03	400
fly	0.10	0.12	0.11	400
black widow, Latrodectus mactans	0.00	0.00	0.00	400
broom	0.00	0.00	0.00	400
baboon	0.04	0.01	0.01	400
spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish	0.00	0.00	0.00	400
Arabian camel, dromedary, Camelus dromedarius	0.00	0.00	0.00	400
ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle	0.00	0.00	0.00	400
American alligator, Alligator mississipiensis	0.00	0.00	0.00	400
beacon, lighthouse, beacon light, pharos	0.00	0.00	0.00	400
accuracy			0.07	6000
macro avg	0.05	0.07	0.04	6000
weighted avg	0.04	0.07	0.04	6000

3.6 Features Stats

The performance of the SIFT and ORB feature extraction methods was analyzed in terms of both the extracted keypoints and their contribution to the overall model accuracy.

3.6.1 SIFT Feature Extraction

SIFT extracts a larger number of keypoints, as shown by the total of 200,021 keypoints across 6,000 images. Despite this large feature set, the accuracy observed from the test data remained relatively low. This outcome suggests that while the method provides rich feature representations, it does not necessarily lead to better performance in the classification task. Possible factors contributing to this could include overfitting due to the large number of keypoints, a lack of meaningful patterns in the data, or insufficient regularization.

3.6.2 ORB Feature Extraction

ORB, on the other hand, extracts significantly fewer keypoints, with only 1,360 keypoints detected across 6,000 images. This small number of keypoints results in poorer feature distributions when combined with BoW and Fisher Vector techniques, which directly impacts the performance of the model. Despite these limitations, ORB's accuracy was not far off from that of SIFT, indicating that a reduced keypoint count may still capture useful information, but to a lesser extent. This highlights the fact that feature extraction quality is not the sole determinant of model success. Other aspects, such as feature representation, model choice, and hyperparameter tuning, can play crucial roles in improving the accuracy.

3.6.3 Insights and Further Investigation

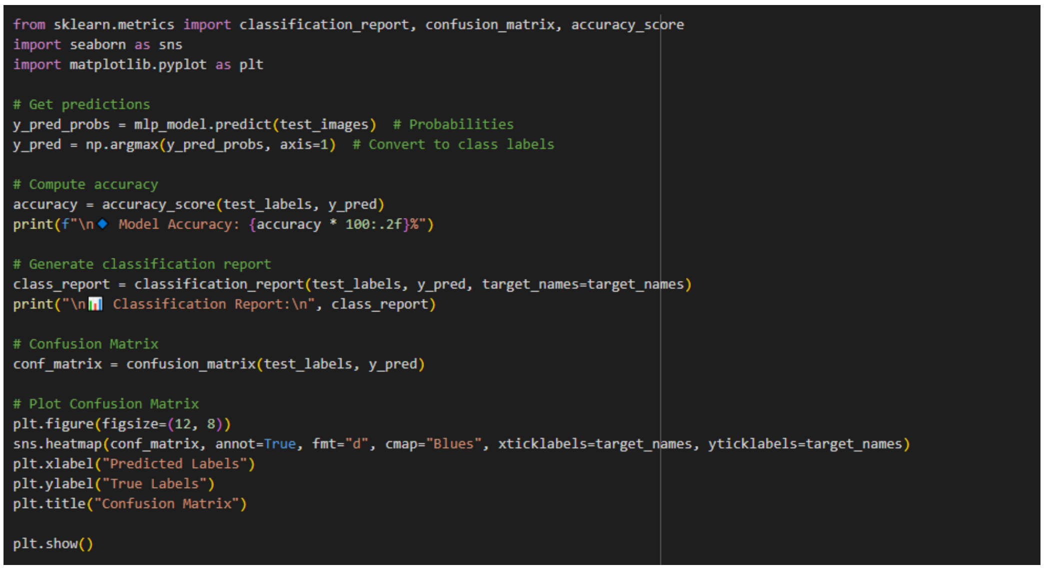
While SIFT yielded slightly better accuracy, the difference in performance between SIFT and ORB was not as significant as expected. This suggests that feature extraction alone might not be the key factor in enhancing model performance. A potential avenue for further investigation would be to explore more advanced techniques for feature selection, regularization, or even combining SIFT and ORB to leverage the strengths of both methods.

Feature Extraction Method	Metric	Value
SIFT Feature Extraction Stats	Total Images Processed	6000
	Total Keypoints Extracted	200021
	Average Keypoints per Image	33.34
	Min Keypoints	0
	Max Keypoints	193
ORB Feature Extraction Stats	Total Images Processed	6000
	Total Keypoints Extracted	1360
	Average Keypoints per Image	0.23
	Min Keypoints	0
	Max Keypoints	1
SIFT + BoW BoW Cluster Distribution Stats	Mean Cluster Distribution	0.33
	Max Cluster Frequency	0.5218
	Min Cluster Frequency	0.2483
ORB + BoW BoW Cluster Distribution Stats	Mean Cluster Distribution	0.00
	Max Cluster Frequency	0.0055
	Min Cluster Frequency	0.001
SIFT + Fisher Vector Fisher Vector Stats	Feature Vector Shape	(6000, 200)
	Mean Values	[0.3491, 0.2023, 0.4775, 0.2777, 0.3799]
	Variance Values	[0.4357, 0.2212, 0.5570, 0.3170, 0.4470]
	Min Value	0.0
	Max Value	101.1781
ORB + Fisher Vector Fisher Vector Stats	Feature Vector Shape	(6000, 200)
	Mean Values	[6.3188e-164, 1.6658e-004, 3.5306e-043, 8.7318e-117, 1.6459e-160]
	Variance Values	[2.4703e-323, 1.6647e-004, 7.4779e-082, 4.5739e-229, 1.6252e-316]
	Min Value	0.0
	Max Value	1.0

Table 2: Stats for the number of keypoints extracted in each step

3.7 Phase 3: Simple Neural Network Approach

The **model accuracy** stands at **35.33%**, which indicates that approximately one-third of the predictions made by the neural network were correct. While this accuracy is relatively low, it serves as a baseline performance for this task.



```python
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt

# Get predictions
y_pred_probs = mlp_model.predict(test_images)  # Probabilities
y_pred = np.argmax(y_pred_probs, axis=1)  # Convert to class labels

# Compute accuracy
accuracy = accuracy_score(test_labels, y_pred)
print(f"\n🔹 Model Accuracy: {accuracy * 100:.2f}%")

# Generate classification report
class_report = classification_report(test_labels, y_pred, target_names=target_names)
print("\n📊 Classification Report:\n", class_report)

# Confusion Matrix
conf_matrix = confusion_matrix(test_labels, y_pred)

# Plot Confusion Matrix
plt.figure(figsize=(12, 8))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=target_names, yticklabels=target_names)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix")

plt.show()
```

Figure 20: Phase 3- Evaluation Function Call

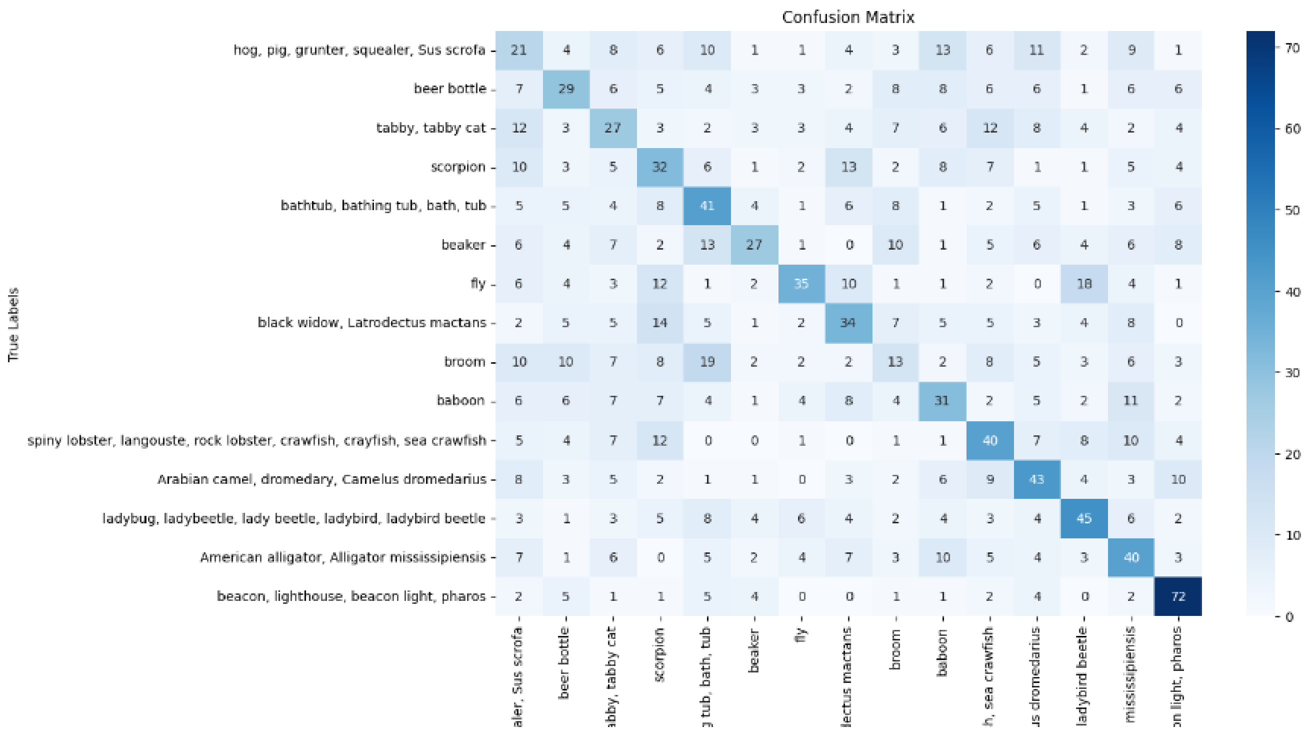
Model Test Accuracy: 35.33%

Class	Precision	Recall	F1-Score	Support
hog, pig, grunter, squealer, Sus scrofa	0.19	0.21	0.20	100
beer bottle	0.33	0.29	0.31	100
tabby, tabby cat	0.27	0.27	0.27	100
scorpion	0.27	0.32	0.29	100
bathtub, bathing tub, bath, tub	0.33	0.41	0.37	100
beaker	0.48	0.27	0.35	100
fly	0.54	0.35	0.42	100
black widow, Latrodectus mactans	0.35	0.34	0.35	100
broom	0.18	0.13	0.15	100
baboon	0.32	0.31	0.31	100
spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish	0.35	0.40	0.37	100
Arabian camel, dromedary, Camelus dromedarius	0.38	0.43	0.41	100
ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle	0.45	0.45	0.45	100
American alligator, Alligator mississipiensis	0.33	0.40	0.36	100
beacon, lighthouse, beacon light, pharos	0.57	0.72	0.64	100
Accuracy			**0.35**	1500
Macro Avg	0.36	0.35	0.35	1500
Weighted Avg	0.36	0.35	0.35	1500

Table 3: Classification Report for Simple Neural Network

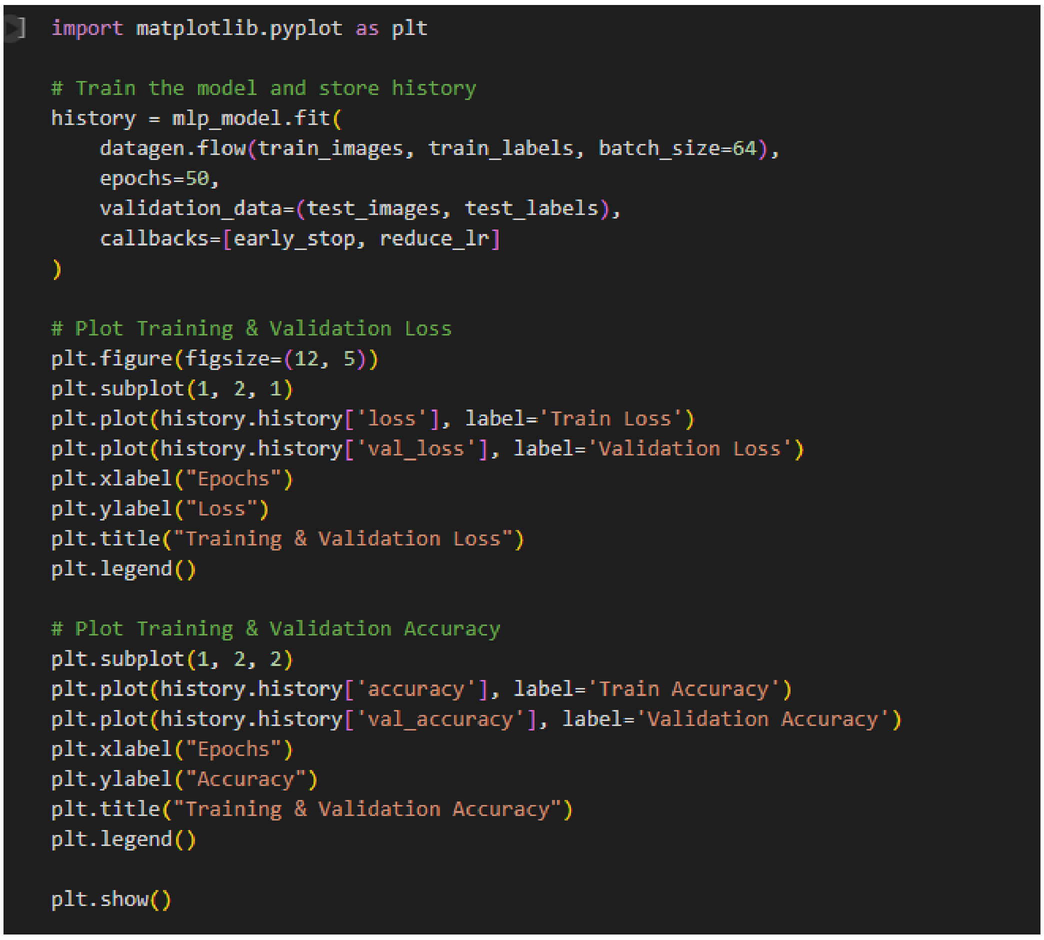
- **Precision, Recall, and F1-Score:** The model's performance varies significantly across different classes, as seen in the table. Some classes, such as *beacon, lighthouse, beacon light, pharos* (Precision: 0.57, Recall: 0.72, F1-Score: 0.64), demonstrate relatively strong results, with a decent balance between precision and recall. This suggests that the model was able to identify instances of this class with reasonable accuracy while minimizing false positives.
- On the other hand, classes like *broom* (Precision: 0.18, Recall: 0.13, F1-Score: 0.15) show poor performance. The very low precision and recall values indicate that the model struggled to correctly identify or differentiate objects in this class, leading to a significant number of false positives and false negatives. This could be due to similarities between classes or insufficient representation of this class in the training data.
- **Macro Average:** The **Macro Avg** values for precision (0.36), recall (0.35), and F1-score (0.35) provide an overall summary of the model's performance across all classes. These metrics suggest that, while the model achieves moderate performance on some classes, there is room for improvement in handling the variety of classes within the dataset.
- **Weighted Average:** The **Weighted Avg** is also 0.36 for precision, recall, and F1-score, reflecting the average performance across all classes, weighted by their support (i.e., the number of instances per

class). This is consistent with the Macro Average, showing that the model's overall performance is relatively balanced but not optimal.



	aler, Sus scrofa	beer bottle	abby, tabby cat	scorpion	g tub, bath, tub	beaker	fly	lectus mactans	broom	baboon	h, sea crawfish	us dromedarius	ladybird beetle	mississipiensis	on light, pharos
hog, pig, grunter, squealer, Sus scrofa	21	4	8	6	10	1	1	4	3	13	6	11	2	9	1
beer bottle	7	29	6	5	4	3	3	2	8	8	6	6	1	6	6
tabby, tabby cat	12	3	27	3	2	3	3	4	7	6	12	8	4	2	4
scorpion	10	3	5	32	6	1	2	13	2	8	7	1	1	5	4
bathtub, bathing tub, bath, tub	5	5	4	8	41	4	1	6	8	1	2	5	1	3	6
beaker	6	4	7	2	13	27	1	0	10	1	5	6	4	6	8
fly	6	4	3	12	1	2	35	10	1	1	2	0	18	4	1
black widow, Latrodectus mactans	2	5	5	14	5	1	2	34	7	5	5	3	4	8	0
broom	10	10	7	8	19	2	2	2	13	2	8	5	3	6	3
baboon	6	6	7	7	4	1	4	8	4	31	2	5	2	11	2
spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish	5	4	7	12	0	0	1	0	1	1	40	7	8	10	4
Arabian camel, dromedary, Camelus dromedarius	8	3	5	2	1	1	0	3	2	6	9	43	4	3	10
ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle	3	1	3	5	8	4	6	4	2	4	3	4	45	6	2
American alligator, Alligator mississipiensis	7	1	6	0	5	2	4	7	3	10	5	4	3	40	3
beacon, lighthouse, beacon light, pharos	2	5	1	1	5	4	0	0	1	1	2	4	0	2	72

Figure 21: Confusion Matrix - Neural Network



```python
import matplotlib.pyplot as plt

# Train the model and store history
history = mlp_model.fit(
    datagen.flow(train_images, train_labels, batch_size=64),
    epochs=50,
    validation_data=(test_images, test_labels),
    callbacks=[early_stop, reduce_lr]
)

# Plot Training & Validation Loss
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training & Validation Loss")
plt.legend()

# Plot Training & Validation Accuracy
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Training & Validation Accuracy")
plt.legend()

plt.show()
```

Figure 22: Phase 3- Training Validation Accuracy Curve Code

The model's performance was evaluated over 50 epochs. During training, the loss for both the training set and the validation set was tracked. Initially, the model exhibited a high loss, which gradually decreased over time, indicating the model was learning and improving. However, the validation loss showed signs of fluctuations despite a general downward trend.

In the first few epochs, both training and validation loss were relatively high, with values of around 2.01 for training loss and 2.05 for validation loss. As the epochs progressed, the training loss improved, reaching a minimum of approximately 1.96 by epoch 27. On the other hand, the validation loss initially decreased but then fluctuated slightly, settling around 2.03 by epoch 32.

The model's accuracy also showed improvement during training, starting from 35.06% in epoch 1 and steadily increasing throughout. However, the validation accuracy showed less consistency, hovering around 35% to 36% with small fluctuations across the epochs.

Moreover, the learning rate was adjusted using the `ReduceLROnPlateau` callback. The learning rate started at 2.5×10^{-4} and was progressively reduced after epochs 9, 27, and 32. This dynamic adjustment of the learning rate aimed to stabilize training and improve convergence, as seen in the gradual decrease in loss values.

The validation loss curve demonstrates the challenges of achieving generalization, with periodic increases in loss values. This suggests that while the model is improving on the training data, it may still face issues related to overfitting or difficulty generalizing to the validation set.

Overall, despite the fluctuations in validation loss, the model shows promising trends with improvements in both training accuracy and loss over time.

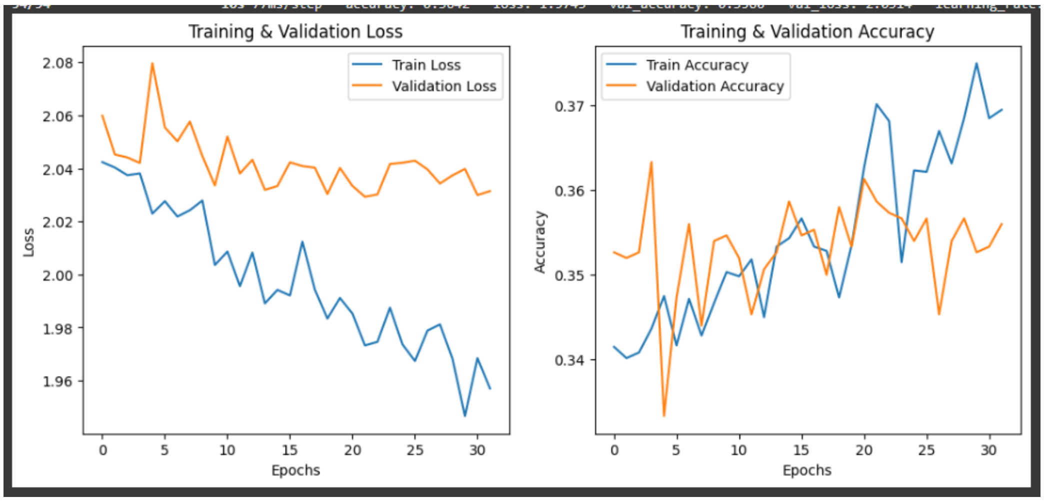
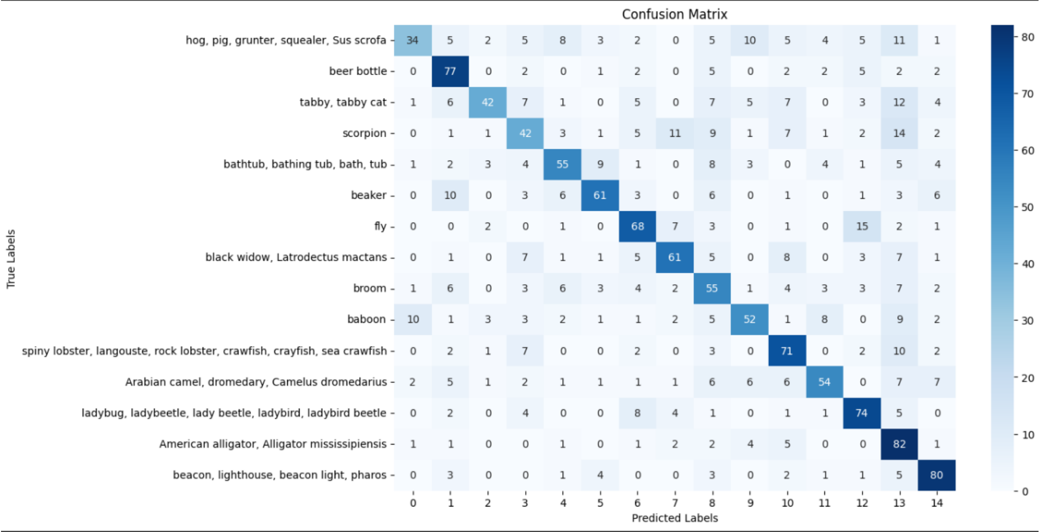


Figure 23: Training/Validation Accuracy Curve

3.8 Phase 4: Convolutional Neural Network Approach

The CNN had by far the most promising results out of the three approaches.

Model Test Accuracy: 60.53%



True Labels	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
hog, pig, grunter, squealer, Sus scrofa	34	5	2	5	8	3	2	0	5	10	5	4	5	11	1
beer bottle	0	77	0	2	0	1	2	0	5	0	2	2	5	2	2
tabby, tabby cat	1	6	42	7	1	0	5	0	7	5	7	0	3	12	4
scorpion	0	1	1	42	3	1	5	11	9	1	7	1	2	14	2
bathtub, bathing tub, bath, tub	1	2	3	4	55	9	1	0	8	3	0	4	1	5	4
beaker	0	10	0	3	6	61	3	0	6	0	1	0	1	3	6
fly	0	0	2	0	1	0	68	7	3	0	1	0	15	2	1
black widow, Latrodectus mactans	0	1	0	7	1	1	5	61	5	0	8	0	3	7	1
broom	1	6	0	3	6	3	4	2	55	1	4	3	3	7	2
baboon	10	1	3	3	2	1	1	2	5	52	1	8	0	9	2
spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish	0	2	1	7	0	0	2	0	3	0	71	0	2	10	2
Arabian camel, dromedary, Camelus dromedarius	2	5	1	2	1	1	1	1	6	6	6	54	0	7	7
ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle	0	2	0	4	0	0	8	4	1	0	1	1	74	5	0
American alligator, Alligator mississipiensis	1	1	0	0	1	0	1	2	2	4	5	0	0	82	1
beacon, lighthouse, beacon light, pharos	0	3	0	0	1	4	0	0	3	0	2	1	1	5	80

Figure 24: Confusion Matrix - CNN (Test)

- **Classes with Strong Performance**:
  - 'Beer bottle' and 'ladybug' demonstrate solid performance with good precision (0.63 and 0.64, respectively) and recall (0.77 and 0.74, respectively). These classes have relatively balanced F1-scores, indicating effective identification by the model.
  - The 'American alligator' class stands out for its high recall (0.82), demonstrating the model's strong ability to detect this class, though its precision (0.45) is lower, suggesting false positives.
- **Classes with Mixed Performance**:
  - 'Bathtub' and 'beaker' show moderate F1-scores (0.59 and 0.66), indicating that while the model can identify these classes reasonably well, there is still room for improvement.
  - 'Spiny lobster' has decent recall (0.71) but lower precision (0.59), pointing to a relatively high number of false positives, despite successfully capturing most instances.
- **Classes with Poor Performance**:
  - 'Scorpion' and 'broom' show particularly low precision (0.47 and 0.45, respectively), indicating a high rate of false positives. These classes also suffer from lower F1-scores (0.44 and 0.49), showing

that the model struggles to balance both false positives and false negatives for these classes.

  – 'Hog, pig, grunter' suffers from low recall (0.34), suggesting that the model misses many instances of this class.

- **General Observations**:
  – Overall accuracy stands at 60.53%, a moderate performance suggesting that the model has potential but needs further optimization.
  – The macro average (Precision: 0.62, Recall: 0.61, F1-Score: 0.60) suggests that, on average, the model performs decently across all classes but could benefit from fine-tuning.
  – The weighted average (Precision: 0.62, Recall: 0.61, F1-Score: 0.60) mirrors the macro average, indicating that class distribution impacts overall metrics.

- **Conclusion**: The CNN model shows decent results for certain classes but struggles with others. Classes with higher support, like 'beer bottle' and 'hog, pig', have a more significant influence on overall performance. Further tuning of the architecture, such as adjusting filter sizes, dropout rates, or even increasing the dataset's diversity, could lead to better performance, especially for the poorly predicted classes.

Class	**Precision**	**Recall**	**F1-score**	**Support**
hog, pig, grunter, squealer, Sus scrofa	0.68	0.34	0.45	100
beer bottle	0.63	0.77	0.69	100
tabby, tabby cat	0.76	0.42	0.54	100
scorpion	0.47	0.42	0.44	100
bathtub, bathing tub, bath, tub	0.64	0.55	0.59	100
beaker	0.72	0.61	0.66	100
fly	0.63	0.68	0.65	100
black widow, Latrodectus mactans	0.68	0.61	0.64	100
broom	0.45	0.55	0.49	100
baboon	0.63	0.52	0.57	100
spiny lobster, langouste, rock lobster, crawfish, crayfish, sea crawfish	0.59	0.71	0.64	100
Arabian camel, dromedary, Camelus dromedarius	0.69	0.54	0.61	100
ladybug, ladybeetle, lady beetle, ladybird, ladybird beetle	0.64	0.74	0.69	100
American alligator, Alligator mississipiensis	0.45	0.82	0.58	100
beacon, lighthouse, beacon light, pharos	0.70	0.80	0.74	100
Accuracy			**0.61**	1500
Macro avg	0.62	0.61	0.60	1500
Weighted avg	0.62	0.61	0.60	1500

Table 4: Classification Report for CNN Model - Test Data

Configuration	Dropout Rate	Learning Rate	Filters
1	0.5	0.001	(32, 64, 128)
2	0.5	0.001	(64, 64, 64)

Table 5: Hyperparameter Configurations for CNN Model

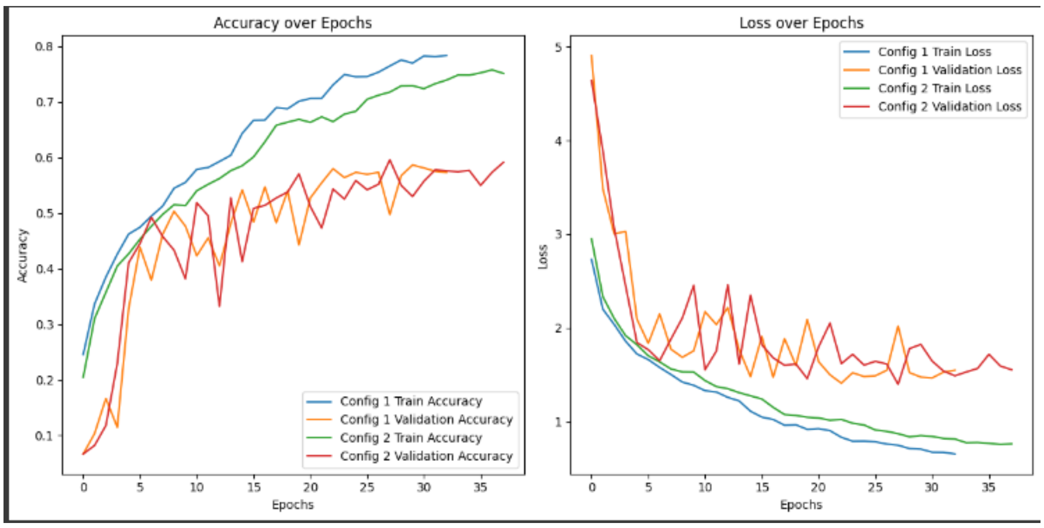


Figure 25: Training/Validation accuracy curves

4 Discussions

4.1 Comparison of Approaches

The performance of SIFT, ORB, and neural networks (both simple and convolutional) highlights their strengths and limitations in image classification.

SIFT vs. ORB: SIFT extracts significantly more keypoints than ORB, yet this did not translate to better accuracy, possibly due to overfitting or insufficiently meaningful features [1]. ORB, with fewer keypoints, achieved comparable results, suggesting that feature quantity alone does not determine model success [2]. Instead, feature representation, model architecture, and hyperparameter tuning play crucial roles. However, test accuracy from both of these approaches was very low.

Neural Networks: The simple neural network achieved 35.33% accuracy after 50 epochs, with training loss decreasing but validation loss fluctuating, indicating overfitting. The ReduceLROnPlateau callback stabilized training by dynamically adjusting the learning rate, yet the model's limited architecture hindered higher accuracy [5].

Convolutional Neural Networks (CNNs): CNNs outperformed SIFT and ORB, achieving 60.53% accuracy. Their ability to learn hierarchical features made them robust to variations in illumination, scale, and rotation [3]. This is consistent with prior findings demonstrating CNNs' superior adaptability to complex image classification tasks [4]. However, CNNs require substantial training data and proper regularization to prevent overfitting. Performance was also influenced by class distribution, suggesting that dataset diversity and architectural refinements could further improve accuracy.

4.2 Training Procedure and Parameter Tuning

Neural Network Training: Trained over 50 epochs, the simple neural network's validation loss fluctuated despite a declining training loss. The learning rate, initially set at 2.5×10^{-4}, was dynamically reduced at epochs 9, 27, and 32 to stabilize training [5]. However, accuracy remained low, indicating potential architectural limitations or insufficient data.

CNN Training: The CNN achieved 60.53% accuracy, with hyperparameters such as dropout rates, learning rates, and filter configurations playing key roles [3]. Two filter settings—(32, 64, 128) and (64, 64, 64)—were tested. A dropout rate of 0.5 mitigated overfitting, and a learning rate of 0.001 balanced convergence. While training showed steady improvement, further tuning could enhance performance, particularly for underrepresented classes [4].

5 Conclusion

Convolutional Neural Networks (CNNs) outperform traditional neural networks and hand-crafted feature methods in this study, achieving the highest accuracy of 60.53% [3]. CNNs excel due to their ability to automatically learn hierarchical feature representations, making them highly robust to variations in illumination, scale, and rotation [4]. This adaptability allows CNNs to effectively handle complex image data, confirming previous research in deep learning for vision tasks [5].

In contrast, neural networks, despite performing better than hand-crafted features, still lag behind CNNs. With only 35.33% accuracy, they struggle to generalize complex image patterns and require extensive training data and regularization to avoid overfitting [5].

Hand-crafted methods like SIFT and ORB, while not as powerful as CNNs, still have notable advantages for smaller datasets or lower computational demands [1, 2]. SIFT, despite extracting a large number of keypoints, provides detailed feature representations that work well for less varied or smaller datasets [1]. ORB, though extracting fewer keypoints, offers efficient performance and is valuable in applications where speed and efficiency are prioritized [2]. These methods remain valuable for smaller, less complex datasets but struggle to compete with the adaptability and feature-learning capabilities of CNNs, especially in large-scale image classification tasks [3, 4].

Word Count

2618 words

References

References

[1] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[2] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, “ORB: An efficient alternative to SIFT or SURF,” in *Proc. 2011 IEEE Int. Conf. Comput. Vis.*, Barcelona, Spain, 2011, pp. 2564–2571.

[3] A. Krizhevsky, I. Sutskever, and G. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[4] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[5] F. Chollet, “Deep learning with Python,” 2nd ed. Manning Publications, 2021.

A Appendix A: Google Colab Code

```
# -*- coding: utf-8 -*-
"""Imama_Jawad_Code_40462364.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1Jgmn6oQDrys3QJbTMX7iPvdNC6ro7QVj

Library Imports
"""

import os
import zipfile
import random
import joblib
import cv2
import kagglehub
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans
from sklearn.svm import SVC
from sklearn.mixture import GaussianMixture
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import accuracy_score, classification_report, confusion_mat
from tensorflow.keras import layers, models, optimizers, callbacks
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam

"""Importing the TinyDataset Folder I uploaded from my kaggle (public)"""

path = kagglehub.dataset_download("imamajawad123/tinydataset")

print("Path to dataset files:", path)

path2 = os.path.join(path, "TinyImageNet100")
```

```
files = os.listdir(path2)

print("Sample folders/files in", path2, ":", files)

path2

"""##Phase 1"""

def load_dataset(dataset_root, class_list_file, num_classes=15, train_samples=40
    """ Loads dataset, selecting classes and splitting into train/test sets. """
    with open(class_list_file, 'r') as file:
        class_map = {line.split('\t')[0]: line.split('\t')[1].strip() for line i

    class_folders = sorted(os.listdir(dataset_root))
    selected_classes = random.sample(class_folders, num_classes) if shuffle else

    train_files, test_files, train_labels, test_labels = [], [], [], []

    for idx, class_name in enumerate(selected_classes):
        img_dir = os.path.join(dataset_root, class_name, 'images')
        images = sorted(os.listdir(img_dir))[:train_samples + test_samples]

        train_files.extend([os.path.join(img_dir, img) for img in images[:train_
        test_files.extend([os.path.join(img_dir, img) for img in images[train_sa

        train_labels.extend([idx] * train_samples)
        test_labels.extend([idx] * test_samples)

    print(f"Number of selected classes: {num_classes}")
    print(f"Selected classes: {selected_classes}")
    print("Class names in the dataset:")
    for class_name in selected_classes:
        print(f"{class_name}: {class_map.get(class_name, 'Unknown')}")  # Print

    return train_files, test_files, train_labels, test_labels, class_map, select

# Load the dataset

root_dir = "/root/.cache/kagglehub/datasets/imamajawad123/tinydataset/versions/1
class_list = "/root/.cache/kagglehub/datasets/imamajawad123/tinydataset/versions
train_files, test_files, train_labels, test_labels, class_map , selected_classes
```

```
"""##Phase 2

Test Set Accuracy + Classification Report
"""

def extract_features(image_paths, method='SIFT'):
    """ Extracts keypoints and descriptors using SIFT or ORB. """
    extractor = cv2.SIFT_create() if method == 'SIFT' else cv2.ORB_create()
    descriptors_list = []

    for img_path in image_paths:
        img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
        keypoints, descriptors = extractor.detectAndCompute(img, None)
        descriptors_list.append(descriptors)

    return descriptors_list

def build_bow_model(descriptors, num_clusters=100):
    """ Constructs a BoW model using K-Means clustering. """
    all_descriptors = np.vstack([desc for desc in descriptors if desc is not Non
    kmeans = KMeans(n_clusters=num_clusters, random_state=42).fit(all_descriptor

    bow_features = []
    for desc in descriptors:
        hist = np.zeros(num_clusters)
        if desc is not None:
            labels = kmeans.predict(desc)
            for label in labels:
                hist[label] += 1
        bow_features.append(hist)

    return np.array(bow_features)

def fisher_vector_encoding(descriptors, num_clusters=100):
    """ Computes Fisher Vectors using a Gaussian Mixture Model. """
    all_descriptors = np.vstack([desc for desc in descriptors if desc is not Non

    gmm = GaussianMixture(n_components=num_clusters, covariance_type='diag', max

    fv_features = []
    for desc in descriptors:
        if desc is None:
            fv_features.append(np.zeros(2 * num_clusters))
            continue

        probabilities = gmm.predict_proba(desc)
        fisher_vector = np.sum(probabilities, axis=0)

        # Padding if the fisher_vector length is shorter than expected
        if len(fisher_vector) < 2 * num_clusters:
            padding = np.zeros(2 * num_clusters - len(fisher_vector))
            fisher_vector = np.concatenate([fisher_vector, padding])

        fv_features.append(fisher_vector)

    return np.array(fv_features)

def evaluate_svm_classifier(model, test_features, test_labels, class_map, select
    """ Evaluates an SVM classifier and prints class names in the classification
    predictions = model.predict(test_features)
    accuracy = accuracy_score(test_labels, predictions)

    target_names = [class_map[selected_classes[label]] for label in sorted(set(t

    class_report = classification_report(test_labels, predictions, target_names=
    return accuracy, class_report

# Extract features using SIFT and ORB for train and test
sift_train_features = extract_features(train_files, method='SIFT')
orb_train_features = extract_features(train_files, method='ORB')

sift_test_features = extract_features(test_files, method='SIFT')
orb_test_features = extract_features(test_files, method='ORB')

# Build BoW and Fisher Vector models for both SIFT and ORB (train data)
num_clusters = 100
bow_features_sift = build_bow_model(sift_train_features, num_clusters)
bow_features_orb = build_bow_model(orb_train_features, num_clusters)

fisher_features_sift = fisher_vector_encoding(sift_train_features, num_clusters)
fisher_features_orb = fisher_vector_encoding(orb_train_features, num_clusters)
```

```
# Train SVM models for all 4 combinations
svm_bow_sift = SVC(kernel='linear', C=1.0, random_state=42)
svm_bow_sift.fit(bow_features_sift, train_labels)

svm_bow_orb = SVC(kernel='linear', C=1.0, random_state=42)
svm_bow_orb.fit(bow_features_orb, train_labels)

svm_fisher_sift = SVC(kernel='linear', C=1.0, random_state=42)
svm_fisher_sift.fit(fisher_features_sift, train_labels)

svm_fisher_orb = SVC(kernel='linear', C=1.0, random_state=42)
svm_fisher_orb.fit(fisher_features_orb, train_labels)

# Build BoW and Fisher Vector models for the test set
bow_features_sift_test = build_bow_model(sift_test_features, num_clusters)
bow_features_orb_test = build_bow_model(orb_test_features, num_clusters)

fisher_features_sift_test = fisher_vector_encoding(sift_test_features, num_clust
fisher_features_orb_test = fisher_vector_encoding(orb_test_features, num_cluster

# Evaluate and print classification results for the test set

# 1. SIFT + BoW
accuracy_bow_sift, report_bow_sift = evaluate_svm_classifier(svm_bow_sift, bow_f

# 2. ORB + BoW
accuracy_bow_orb, report_bow_orb = evaluate_svm_classifier(svm_bow_orb, bow_feat

# 3. SIFT + Fisher Vector
accuracy_fisher_sift, report_fisher_sift = evaluate_svm_classifier(svm_fisher_si

# 4. ORB + Fisher Vector
accuracy_fisher_orb, report_fisher_orb = evaluate_svm_classifier(svm_fisher_orb,

# Print results for test set
print("Model 1: SIFT + BoW")
print(f"Accuracy: {accuracy_bow_sift:.2f}")
print(report_bow_sift)

print("\nModel 2: ORB + BoW")
print(f"Accuracy: {accuracy_bow_orb:.2f}")
print(report_bow_orb)

print("\nModel 3: SIFT + Fisher Vector")
print(f"Accuracy: {accuracy_fisher_sift:.2f}")
print(report_fisher_sift)

print("\nModel 4: ORB + Fisher Vector")
print(f"Accuracy: {accuracy_fisher_orb:.2f}")
print(report_fisher_orb)

# Evaluate and print best model on the test set
best_accuracy = max(accuracy_bow_sift, accuracy_bow_orb, accuracy_fisher_sift, a
if best_accuracy == accuracy_bow_sift:
    print("\nBest Model on Test Set: SIFT + BoW")
elif best_accuracy == accuracy_bow_orb:
    print("\nBest Model on Test Set: ORB + BoW")
elif best_accuracy == accuracy_fisher_sift:
    print("\nBest Model on Test Set: SIFT + Fisher Vector")
else:
    print("\nBest Model on Test Set: ORB + Fisher Vector")

"""Train set Classification Reports"""

def extract_features(image_paths, method='SIFT'):
    """ Extracts keypoints and descriptors using SIFT or ORB. """
    extractor = cv2.SIFT_create() if method == 'SIFT' else cv2.ORB_create()
    descriptors_list = []

    for img_path in image_paths:
        img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
        keypoints, descriptors = extractor.detectAndCompute(img, None)
        descriptors_list.append(descriptors)

    return descriptors_list

def build_bow_model(descriptors, num_clusters=100):
    """ Constructs a BoW model using K-Means clustering. """
    all_descriptors = np.vstack([desc for desc in descriptors if desc is not Non
    kmeans = KMeans(n_clusters=num_clusters, random_state=42).fit(all_descriptor
    bow_features = []
    for desc in descriptors:
        hist = np.zeros(num_clusters)
        if desc is not None:
            labels = kmeans.predict(desc)
            for label in labels:
                hist[label] += 1
        bow_features.append(hist)

    return np.array(bow_features)

def fisher_vector_encoding(descriptors, num_clusters=100):
    """ Computes Fisher Vectors using a Gaussian Mixture Model. """
    all_descriptors = np.vstack([desc for desc in descriptors if desc is not Non

    gmm = GaussianMixture(n_components=num_clusters, covariance_type='diag', max

    fv_features = []
    for desc in descriptors:
        if desc is None:
            fv_features.append(np.zeros(2 * num_clusters))
            continue

        probabilities = gmm.predict_proba(desc)
        fisher_vector = np.sum(probabilities, axis=0)

        # Padding if the fisher_vector length is shorter than expected
        if len(fisher_vector) < 2 * num_clusters:
            padding = np.zeros(2 * num_clusters - len(fisher_vector))
            fisher_vector = np.concatenate([fisher_vector, padding])

        fv_features.append(fisher_vector)

    return np.array(fv_features)

def evaluate_svm_classifier(model, test_features, test_labels, class_map, select
    """ Evaluates an SVM classifier and prints class names in the classification
    predictions = model.predict(test_features)
    accuracy = accuracy_score(test_labels, predictions)

    target_names = [class_map[selected_classes[label]] for label in sorted(set(t
    class_report = classification_report(test_labels, predictions, target_names=
    return accuracy, class_report

# Extract features using SIFT and ORB
sift_train_features = extract_features(train_files, method='SIFT')
orb_train_features = extract_features(train_files, method='ORB')

# Build BoW and Fisher Vector models for both SIFT and ORB
num_clusters = 100
bow_features_sift = build_bow_model(sift_train_features, num_clusters)
bow_features_orb = build_bow_model(orb_train_features, num_clusters)

fisher_features_sift = fisher_vector_encoding(sift_train_features, num_clusters)
fisher_features_orb = fisher_vector_encoding(orb_train_features, num_clusters)

# Train and evaluate SVM models for all 4 combinations

# 1. SIFT + BoW
svm_bow_sift = SVC(kernel='linear', C=1.0, random_state=42)
svm_bow_sift.fit(bow_features_sift, train_labels)
accuracy_bow_sift, report_bow_sift = evaluate_svm_classifier(svm_bow_sift, bow_f

# 2. ORB + BoW
svm_bow_orb = SVC(kernel='linear', C=1.0, random_state=42)
svm_bow_orb.fit(bow_features_orb, train_labels)
accuracy_bow_orb, report_bow_orb = evaluate_svm_classifier(svm_bow_orb, bow_feat

# 3. SIFT + Fisher Vector
svm_fisher_sift = SVC(kernel='linear', C=1.0, random_state=42)
svm_fisher_sift.fit(fisher_features_sift, train_labels)
accuracy_fisher_sift, report_fisher_sift = evaluate_svm_classifier(svm_fisher_si

# 4. ORB + Fisher Vector
svm_fisher_orb = SVC(kernel='linear', C=1.0, random_state=42)
svm_fisher_orb.fit(fisher_features_orb, train_labels)
accuracy_fisher_orb, report_fisher_orb = evaluate_svm_classifier(svm_fisher_orb,

# Print results
print("Model 1: SIFT + BoW")
print(f"Accuracy: {accuracy_bow_sift:.2f}")
print(report_bow_sift)

print("\nModel 2: ORB + BoW")
print(f"Accuracy: {accuracy_bow_orb:.2f}")
print(report_bow_orb)

print("\nModel 3: SIFT + Fisher Vector")
print(f"Accuracy: {accuracy_fisher_sift:.2f}")
print(report_fisher_sift)

print("\nModel 4: ORB + Fisher Vector")
print(f"Accuracy: {accuracy_fisher_orb:.2f}")
print(report_fisher_orb)

# Evaluate and print best model
best_accuracy = max(accuracy_bow_sift, accuracy_bow_orb, accuracy_fisher_sift, a
if best_accuracy == accuracy_bow_sift:
    print("\nBest Model: SIFT + BoW")
elif best_accuracy == accuracy_bow_orb:
    print("\nBest Model: ORB + BoW")
elif best_accuracy == accuracy_fisher_sift:
    print("\nBest Model: SIFT + Fisher Vector")
else:
    print("\nBest Model: ORB + Fisher Vector")

# Save the best model to a file
joblib.dump(svm_fisher_sift, 'best_model_phase2.pkl')
print("Best model saved as 'best_model_phase2.pkl'.")

def analyze_descriptors(descriptors_list, method_name):
    """ Compute stats on the extracted features (SIFT or ORB). """
    num_keypoints = [desc.shape[0] if desc is not None else 0 for desc in descri

    print(f"\n=== {method_name} Feature Extraction Stats ===")
    print(f"Total Images Processed: {len(num_keypoints)}")
    print(f"Total Keypoints Extracted: {sum(num_keypoints)}")
    print(f"Average Keypoints per Image: {np.mean(num_keypoints):.2f}")
    print(f"Min Keypoints: {np.min(num_keypoints)}")
    print(f"Max Keypoints: {np.max(num_keypoints)}")

def analyze_bow(bow_features, method_name):
    """ Compute stats on the BoW model (cluster distribution). """
    avg_histogram = np.mean(bow_features, axis=0)

    print(f"\n=== {method_name} BoW Cluster Distribution Stats ===")
    print(f"Mean Cluster Distribution: {np.mean(avg_histogram):.2f}")
    print(f"Max Cluster Frequency: {np.max(avg_histogram)}")
    print(f"Min Cluster Frequency: {np.min(avg_histogram)}")

def analyze_fisher_vectors(fisher_features, method_name):
    """ Compute stats on Fisher Vectors (mean, variance, etc.). """
    print(f"\n=== {method_name} Fisher Vector Stats ===")
    print(f"Feature Vector Shape: {fisher_features.shape}")
    print(f"Mean Values: {np.mean(fisher_features, axis=0)[:5]} ... (first 5 val
    print(f"Variance Values: {np.var(fisher_features, axis=0)[:5]} ... (first 5 
    print(f"Min Value: {np.min(fisher_features)}")
    print(f"Max Value: {np.max(fisher_features)}")

# Analyze extracted features
analyze_descriptors(sift_train_features, "SIFT")
analyze_descriptors(orb_train_features, "ORB")

# Analyze BoW models
analyze_bow(bow_features_sift, "SIFT + BoW")
analyze_bow(bow_features_orb, "ORB + BoW")

# Analyze Fisher Vectors
analyze_fisher_vectors(fisher_features_sift, "SIFT + Fisher Vector")
analyze_fisher_vectors(fisher_features_orb, "ORB + Fisher Vector")

"""Test + Train Set Confusion Matrices"""

# Function to plot confusion matrices
def plot_confusion_matrix(true_labels, predictions, model_name, class_labels):
    conf_matrix = confusion_matrix(true_labels, predictions)

    plt.figure(figsize=(10, 7))
    sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=clas
    plt.xlabel("Predicted Labels")
    plt.ylabel("True Labels")
    plt.title(f"Confusion Matrix - {model_name}")
    plt.show()
```

```
# Predictions for each model on the training set
y_pred_bow_sift_train = svm_bow_sift.predict(bow_features_sift)
y_pred_bow_orb_train = svm_bow_orb.predict(bow_features_orb)
y_pred_fisher_sift_train = svm_fisher_sift.predict(fisher_features_sift)
y_pred_fisher_orb_train = svm_fisher_orb.predict(fisher_features_orb)
# Build BoW and Fisher Vector models for the test set
bow_features_sift_test = build_bow_model(sift_test_features, num_clusters)
bow_features_orb_test = build_bow_model(orb_test_features, num_clusters)

fisher_features_sift_test = fisher_vector_encoding(sift_test_features, num_clust
fisher_features_orb_test = fisher_vector_encoding(orb_test_features, num_cluster

# Predictions for each model on the test set
y_pred_bow_sift_test = svm_bow_sift.predict(bow_features_sift_test)
y_pred_bow_orb_test = svm_bow_orb.predict(bow_features_orb_test)
y_pred_fisher_sift_test = svm_fisher_sift.predict(fisher_features_sift_test)
y_pred_fisher_orb_test = svm_fisher_orb.predict(fisher_features_orb_test)

# Now you can proceed with the rest of the code to plot confusion matrices
# Class labels
target_names = [class_map[selected_classes[label]] for label in sorted(set(train

# Plot Confusion Matrices for training set
plot_confusion_matrix(train_labels, y_pred_bow_sift_train, "SIFT + BoW (Train)",
plot_confusion_matrix(train_labels, y_pred_bow_orb_train, "ORB + BoW (Train)", t
plot_confusion_matrix(train_labels, y_pred_fisher_sift_train, "SIFT + Fisher Vec
plot_confusion_matrix(train_labels, y_pred_fisher_orb_train, "ORB + Fisher Vecto

# Plot Confusion Matrices for test set
plot_confusion_matrix(test_labels, y_pred_bow_sift_test, "SIFT + BoW (Test)", ta
plot_confusion_matrix(test_labels, y_pred_bow_orb_test, "ORB + BoW (Test)", targ
plot_confusion_matrix(test_labels, y_pred_fisher_sift_test, "SIFT + Fisher Vecto
plot_confusion_matrix(test_labels, y_pred_fisher_orb_test, "ORB + Fisher Vector

mapped_labels = [class_map[selected_classes[label]] for label in test_labels]

target_names = [class_map[selected_classes[label]] for label in sorted(set(test_
target_names

"""##Phase 3"""
```

```
# Function to preprocess images (with normalization)
def preprocess_images(img_paths, size=(64, 64)):
    images = []
    for img_path in img_paths:
        img = cv2.imread(img_path)
        if img is None:
            print(f"Warning: Unable to read image at {img_path}. Skipping.")
            continue
        img = cv2.resize(img, size)
        img = img.astype('float32') / 255.0  # Normalize to [0, 1]
        images.append(img)
    return np.array(images)

# Load dataset (adjust paths accordingly)
train_images = preprocess_images(train_files)
test_images = preprocess_images(test_files)

# Check if images were loaded correctly
if len(train_images) == 0 or len(test_images) == 0:
    raise ValueError("No images were loaded. Check the file paths and ensure ima

# Ensure labels are numpy arrays
train_labels = np.array(train_labels)
test_labels = np.array(test_labels)

# Convert labels to categorical encoding
label_encoder = LabelEncoder()
train_labels = label_encoder.fit_transform(train_labels)
test_labels = label_encoder.transform(test_labels)

# Data Augmentation for better generalization
datagen = ImageDataGenerator(
    rotation_range=15,  # Rotate images by 15 degrees
    width_shift_range=0.1,  # Shift width by 10%
    height_shift_range=0.1,  # Shift height by 10%
    horizontal_flip=True  # Randomly flip images
)

# Build the improved MLP model
def build_optimized_mlp(input_shape, num_classes):
    model = models.Sequential([
        layers.Flatten(input_shape=input_shape),
        layers.Dense(1024, activation='relu'),  # More neurons
        layers.BatchNormalization(),
        layers.Dropout(0.4),  # Higher dropout to prevent overfitting
        layers.Dense(512, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.3),
        layers.Dense(256, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.3),
        layers.Dense(num_classes, activation='softmax')
    ])

    # Compile with AdamW optimizer and learning rate decay
    model.compile(
        optimizer=optimizers.AdamW(learning_rate=0.001, weight_decay=1e-4),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    model.summary()
    return model

# Model Training Settings
input_shape = train_images.shape[1:]  # Input shape
num_classes = len(np.unique(train_labels))  # Number of classes

# Initialize model
mlp_model = build_optimized_mlp(input_shape, num_classes)

# Callbacks for better training
early_stop = callbacks.EarlyStopping(monitor='val_loss', patience=10, restore_be
reduce_lr = callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience

# Train the MLP model
print("Training the optimized MLP model...")
mlp_model.fit(
    datagen.flow(train_images, train_labels, batch_size=64),  # Augmented traini
    epochs=50,  # More epochs for deeper model
    validation_data=(test_images, test_labels),
    callbacks=[early_stop, reduce_lr]
)

# Evaluate the model
print("Evaluating the optimized MLP model...")
loss, acc = mlp_model.evaluate(test_images, test_labels)
print(f"Optimized MLP Accuracy: {acc * 100:.2f}%")

# Get predictions
y_pred_probs = mlp_model.predict(test_images)  # Probabilities
y_pred = np.argmax(y_pred_probs, axis=1)  # Convert to class labels

# Compute accuracy
accuracy = accuracy_score(test_labels, y_pred)
print(f"\n Model Accuracy: {accuracy * 100:.2f}%")

# Generate classification report
class_report = classification_report(test_labels, y_pred, target_names=target_na
print("\n Classification Report:\n", class_report)

# Confusion Matrix
conf_matrix = confusion_matrix(test_labels, y_pred)

# Plot Confusion Matrix
plt.figure(figsize=(12, 8))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=target_n
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix")

plt.show()

# Train the model and store history
history = mlp_model.fit(
    datagen.flow(train_images, train_labels, batch_size=64),
    epochs=50,
    validation_data=(test_images, test_labels),
    callbacks=[early_stop, reduce_lr]
)

# Plot Training & Validation Loss
```

```
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training & Validation Loss")
plt.legend()

# Plot Training & Validation Accuracy
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.title("Training & Validation Accuracy")
plt.legend()

plt.show()

mlp_model.save('best_model_phase_3.h5')
print("MLP Model saved as best_model_phase_3.h5")

"""##Phase 4"""

# Preprocess images function
def preprocess_images(img_paths, size=(64, 64)):
    images = []
    for img_path in img_paths:
        img = cv2.imread(img_path)
        if img is None:
            print(f"Warning: Unable to read image at {img_path}. Skipping.")
            continue
        img = cv2.resize(img, size)
        img = img.astype('float32') / 255.0  # Normalize to [0, 1]
        images.append(img)
    return np.array(images)

# Load dataset
train_images = preprocess_images(train_files)
test_images = preprocess_images(test_files)
```

```
train_labels = np.array(train_labels)
test_labels = np.array(test_labels)

# Label encoding
label_encoder = LabelEncoder()
train_labels = label_encoder.fit_transform(train_labels)
test_labels = label_encoder.transform(test_labels)

# Data augmentation
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)

# CNN Model Architecture
def build_cnn_model(input_shape, num_classes):
    model = models.Sequential([
        layers.Conv2D(32, (3,3), activation='relu', padding='same', input_shape=
        layers.BatchNormalization(),
        layers.MaxPooling2D(pool_size=(2,2)),

        layers.Conv2D(64, (3,3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D(pool_size=(2,2)),

        layers.Conv2D(128, (3,3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D(pool_size=(2,2)),

        layers.Flatten(),
        layers.Dense(256, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation='softmax')
    ])

    # Compile model
    model.compile(
        optimizer=optimizers.Adam(learning_rate=0.001),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    model.summary()
    return model

# Train and evaluate CNN
cnn_model = build_cnn_model(input_shape=train_images.shape[1:], num_classes=len(

early_stop = callbacks.EarlyStopping(monitor='val_loss', patience=10, restore_be
reduce_lr = callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience

history = cnn_model.fit(
    datagen.flow(train_images, train_labels, batch_size=64),
    epochs=50,
    validation_data=(test_images, test_labels),
    callbacks=[early_stop, reduce_lr]
)

# Evaluate the CNN model
loss, acc = cnn_model.evaluate(test_images, test_labels)
print(f"Optimized CNN Accuracy: {acc * 100:.2f}%")

# Plot training and validation accuracy/loss
plt.figure(figsize=(12, 6))

# Accuracy plot
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# Loss plot
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()

# Final Accuracy
accuracy = np.sum(y_pred == test_labels) / len(test_labels)
print(f"Final Accuracy: {accuracy * 100:.2f}%")

# Classification Report
y_pred_probs = cnn_model.predict(test_images)
y_pred = np.argmax(y_pred_probs, axis=1)

# Print classification report with class names
class_report = classification_report(test_labels, y_pred, target_names=target_na
print("\nClassification Report:\n", class_report)

# Confusion Matrix
conf_matrix = confusion_matrix(test_labels, y_pred)

# Plot Confusion Matrix
plt.figure(figsize=(12, 8))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=label_en
            target_names)
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.title("Confusion Matrix")
plt.show()
cnn_model.save('cnn_model_phase4.h5')  # Save only the best model
print(f"Best model saved as 'cnn_model_phase4.h5'")

"""Additional hyperparameter tuning for phase 4"""

# Preprocess images function
def preprocess_images(img_paths, size=(64, 64)):
    images = []
    for img_path in img_paths:
        img = cv2.imread(img_path)
        if img is None:
            print(f"Warning: Unable to read image at {img_path}. Skipping.")
            continue
        img = cv2.resize(img, size)
        img = img.astype('float32') / 255.0  # Normalize to [0, 1]
        images.append(img)
    return np.array(images)

# Load dataset
train_images = preprocess_images(train_files)
test_images = preprocess_images(test_files)

train_labels = np.array(train_labels)
test_labels = np.array(test_labels)

# Label encoding
label_encoder = LabelEncoder()
train_labels = label_encoder.fit_transform(train_labels)
test_labels = label_encoder.transform(test_labels)

# Data augmentation
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)

# CNN Model Architecture (Keras)
def build_cnn_model(input_shape, num_classes, dropout_rate=0.5, learning_rate=0.
    model = models.Sequential([
        layers.Conv2D(filters[0], (3,3), activation='relu', padding='same', inpu
        layers.BatchNormalization(),
        layers.MaxPooling2D(pool_size=(2,2)),

        layers.Conv2D(filters[1], (3,3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D(pool_size=(2,2)),

        layers.Conv2D(filters[2], (3,3), activation='relu', padding='same'),
        layers.BatchNormalization(),
        layers.MaxPooling2D(pool_size=(2,2)),

        layers.Flatten(),
        layers.Dense(256, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(dropout_rate),
        layers.Dense(num_classes, activation='softmax')
    ])

    # Compile model
    model.compile(
        optimizer=optimizers.Adam(learning_rate=learning_rate),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    model.summary()
    return model

# Hyperparameter configurations (original + one filter size)
hyperparameters = [
    {'dropout_rate': 0.5, 'learning_rate': 0.001, 'filters': (32, 64, 128)},  # 
    {'dropout_rate': 0.5, 'learning_rate': 0.001, 'filters': (64, 64, 64)}  # On
]

best_model = None
best_acc = 0
history_dict = {}

# Train and evaluate CNN with different hyperparameters
for idx, params in enumerate(hyperparameters):
    print(f"Training with configuration {idx + 1}: {params}")

    cnn_model = build_cnn_model(
        input_shape=train_images.shape[1:],
        num_classes=len(np.unique(train_labels)),
        dropout_rate=params['dropout_rate'],
        learning_rate=params['learning_rate'],
        filters=params['filters']
    )
    early_stop = callbacks.EarlyStopping(monitor='val_loss', patience=10, restor
    reduce_lr = callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.5, pati

    history = cnn_model.fit(
        datagen.flow(train_images, train_labels, batch_size=64),
        epochs=50,
        validation_data=(test_images, test_labels),
        callbacks=[early_stop, reduce_lr]
    )

    # Evaluate the CNN model
    loss, acc = cnn_model.evaluate(test_images, test_labels)
    print(f"Configuration {idx + 1} - Accuracy: {acc * 100:.2f}%")

    history_dict[idx] = history.history

    # Update the best model
    if acc > best_acc:
        best_acc = acc
        best_model = cnn_model
        best_params = params
        best_history = history_dict[idx]

# Save the best model
if best_model:
    best_model.save('best_model_phase4.h5')  # Save only the best model
    print(f"Best model saved as 'best_model_phase4.h5'")

# Plot training and validation accuracy/loss for all configurations
plt.figure(figsize=(12, 6))

# Accuracy plot
plt.subplot(1, 2, 1)
for idx in range(len(hyperparameters)):
    plt.plot(history_dict[idx]['accuracy'], label=f'Config {idx + 1} Train Accur
    plt.plot(history_dict[idx]['val_accuracy'], label=f'Config {idx + 1} Validat
plt.title('Accuracy over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
```

```
# Loss plot
plt.subplot(1, 2, 2)
for idx in range(len(hyperparameters)):
    plt.plot(history_dict[idx]['loss'], label=f'Config {idx + 1} Train Loss')
    plt.plot(history_dict[idx]['val_loss'], label=f'Config {idx + 1} Validation
plt.title('Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```