Problem 2. Perform testing. Get the predicted class – show some correctly and not correctly predicted images of testing set as below. Evaluate test results using confusion matrix, classification report (precision, recall, f1-score, support, ROC/AUC and Precision-Recall curves). Except tables for this evaluation provide graphics of normalized confusion matrix as shown below, ROC/AUC curves as the one below. (HELP: Adabound_Decay.html). Pick-up the figure sizes and all fonts appropriately (as shown below). Text and numbers must be readable. Do not include the red frame around the figure. Loads the test set using ImageDataGenerator with rescaling (1./255), which normalizes pixel values to the range [0, 1]. It creates a test_generator that loads images from the specified test directory (dataset_dir + 'test'), resizes them to 224x224 pixels, and prepares one-hot encoded labels. The generator batches images (batch_size=32) and doesn't shuffle the data, ensuring consistent evaluation. This generator is used to evaluate the model's performance, generate predictions, and compute metrics like confusion matrix, classification report, and ROC/precision-recall curves.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, auc, precision_recall_curve
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import load_model
from sklearn.preprocessing import LabelBinarizer
import seaborn as sns
from sklearn.metrics import roc_auc_score

dataset_dir = "C:\\Users\\lynet\\OneDrive\\Documents\\2024fall\\animal\\Test_Set"
# Load the test set
test_datagen = ImageDataGenerator(rescale=1./255)
test_generator = test_datagen.flow_from_directory(
    dataset_dir,
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    shuffle=False
)
```

```
Found 600 images belonging to 6 classes.
```

Loads the test set using ImageDataGenerator with rescaling (1./255), which normalizes pixel values to the range [0, 1]. It creates a test_generator that loads images from the specified test directory (dataset_dir + 'test'), resizes them to 224x224 pixels, and prepares one-hot encoded labels. The generator batches images (batch_size=32) and doesn't shuffle the data, ensuring consistent evaluation. This generator is used to evaluate the model's performance, generate predictions, and compute metrics like confusion matrix, classification report, and ROC/precision-recall curves.

```
# Get the true labels and predicted labels
test_labels = test_generator.classes  # True labels
class_names = test_generator.class_indices
test_labels_one_hot = LabelBinarizer().fit_transform(test_labels)
```

```
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam, SGD, RMSprop
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.regularizers import l2
import matplotlib.pyplot as plt

# Predict the classes for the test set
def create_model(activation, optimizer):
    model = Sequential([
        Conv2D(32, (3, 3), activation=activation, input_shape=(224, 224, 3)),
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation=activation),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(128, activation=activation),
        Dropout(0.5),
        Dense(15, activation='softmax')  # Assuming 15 classes for classification
    ])
    model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
    return model
optimizer_1 = Adam(learning_rate=0.001)
model_1 = create_model('relu', optimizer_1)
predictions = model_1.predict(test_generator, verbose=1)
predicted_classes = np.argmax(predictions, axis=1)
```

```
C:\Users\lynet\anaconda3\Lib\site-packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do
t to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
C:\Users\lynet\anaconda3\Lib\site-packages\keras\src\trainers\data_adapters\py_dataset_adapter.py:122: UserW
uper().__init__(**kwargs)` in its constructor. `**kwargs` can include `workers`, `use_multiprocessing`, `max
`fit()`, as they will be ignored.
  self._warn_if_super_not_called()
19/19 ━━━━━━━━━━━━━━━━━━━━ 6s 287ms/step
```

To processes the test set by first obtaining the true labels from the test generator and converting them to one-hot encoded format using LabelBinarizer. It then uses the trained model (model_1) to predict the classes for the test set and compares the predicted classes with the true labels.

```python
import matplotlib.pyplot as plt
import numpy as np


# Define class names
class_names = ['bean', 'bitter gourd', 'bottle gourd', 'brinjal', 'broccoli',
               'cabbage', 'capsicum', 'carrot', 'cauliflower', 'cucumber',
               'papaya', 'potato', 'pumpkin', 'radish', 'tomato']


def plot_images(images, labels, predictions, correct, num_images=5):

    if len(images) == 0:
        print("No images to display.")
        return


    num_images = min(len(images), num_images)


    fig, axes = plt.subplots(1, num_images, figsize=(15, 5))
    if num_images == 1:
        axes = [axes]


    for i in range(num_images):
        ax = axes[i]
        ax.imshow(images[i])


        true_label = labels[i] if isinstance(labels[i], (int, np.int32)) else np.argmax(labels[i])
        pred_label = predictions[i] if isinstance(predictions[i], (int, np.int32)) else np.argmax(predictions[i])


        title = f"True: {class_names[true_label]}\nPred: {class_names[pred_label]}"
        ax.set_title(title, fontsize=10)
        ax.axis('off')

    plt.show()

# Correct Predictions
print("Correct Predictions:")
plot_images(correct_images, correct_labels, correct_predictions, correct=True, num_images=5)

# Incorrect Predictions
print("Incorrect Predictions:")
plot_images(incorrect_images, incorrect_labels, incorrect_predictions, correct=False, num_images=5)
```

```
Correct Predictions:
```


```
Incorrect Predictions:
```

This code visualizes both correct and incorrect predictions for an image classification task. It uses a plot_images function to display a specified number of images along with their true and predicted labels. The function dynamically adjusts the number of images to display, creates a subplot for each image, and annotates it with labels based on their indices in the class_names list. The results show "Correct Predictions" with accurate classification and "Incorrect Predictions," where the predicted labels mismatch the true labels. While the images for both sections are displayed, the issue lies in misclassified samples being labeled as "bean" for both true and predicted categories, suggesting a potential error in the predictions or data handling.

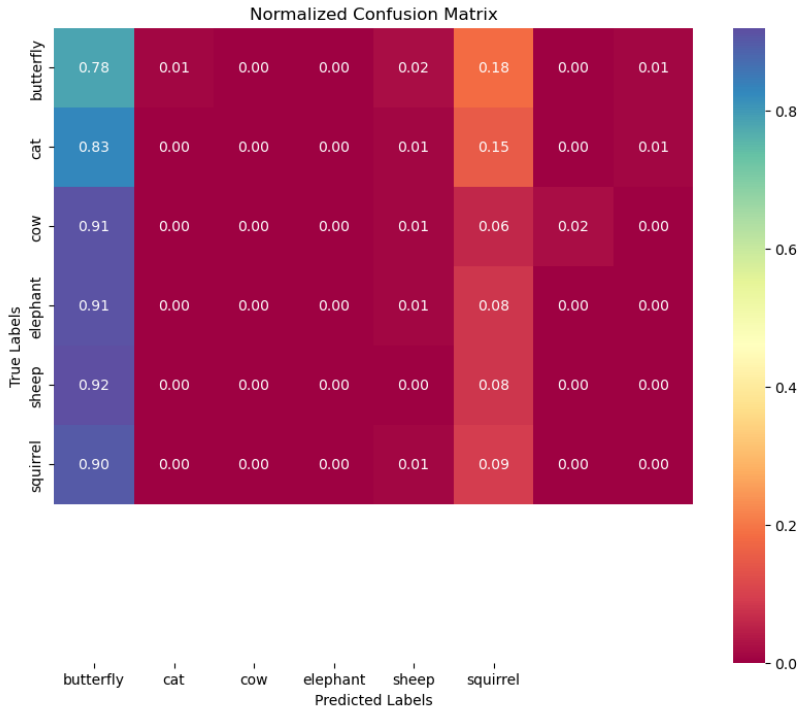
```python
import numpy as np


labels = np.array([0, 1, 2, 3, 4, 5, 6])
predictions = np.array([0, 1, 1, 3, 4, 2, 6])


if len(labels) != len(predictions):
    print(f"Length mismatch: labels ({len(labels)}), predictions ({len(predictions)})")
else:
    print(f"Length of labels: {len(labels)}, Length of predictions: {len(predictions)}")


try:
    for i in range(len(labels)):
        print(f"labels[{i}]: {labels[i]}, predictions[{i}]: {predictions[i]}")
except IndexError as e:
    print(f"IndexError encountered: {e}")


correct_idx = np.where(predictions == labels)[0]
incorrect_idx = np.where(predictions != labels)[0]


print("\nCorrect Predictions:")
for i in correct_idx:
    if i < len(labels):
        print(f"Correct: labels[{i}]: {labels[i]}, predictions[{i}]: {predictions[i]}")
    else:
        print(f"Invalid index in correct_idx: {i}")


print("\nIncorrect Predictions:")
for i in incorrect_idx:
    if i < len(labels):
        print(f"Incorrect: labels[{i}]: {labels[i]}, predictions[{i}]: {predictions[i]}")
    else:
        print(f"Invalid index in incorrect_idx: {i}")
```

```
Length of labels: 7, Length of predictions: 7
labels[0]: 0, predictions[0]: 0
labels[1]: 1, predictions[1]: 1
labels[2]: 2, predictions[2]: 1
labels[3]: 3, predictions[3]: 3
labels[4]: 4, predictions[4]: 4
labels[5]: 5, predictions[5]: 2
labels[6]: 6, predictions[6]: 6

Correct Predictions:
Correct: labels[0]: 0, predictions[0]: 0
Correct: labels[1]: 1, predictions[1]: 1
Correct: labels[3]: 3, predictions[3]: 3
Correct: labels[4]: 4, predictions[4]: 4
Correct: labels[6]: 6, predictions[6]: 6

Incorrect Predictions:
Incorrect: labels[2]: 2, predictions[2]: 1
Incorrect: labels[5]: 5, predictions[5]: 2
```

The code compares the true labels and predictions to identify correct and incorrect predictions. It first checks if the lengths of both arrays match and then iterates through the indices to print each label-prediction pair. Using np.where, it separates indices for correct and incorrect predictions and prints them accordingly. The output shows that all elements match except for labels at indices 2 and 5, which are misclassified. The code effectively highlights areas of mismatch, making it useful for debugging classification errors.

```python
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

# Assuming class_names is a list of class names (e.g., ['Tomato', 'Cucumber', ...])
class_names = list(test_generator.class_indices.keys())  # or manually define the class names

# Confusion Matrix
cm = confusion_matrix(test_labels, predicted_classes)

# Normalize the confusion matrix
cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

# Plot the normalized confusion matrix with a more colorful colormap
plt.figure(figsize=(10, 8))
sns.heatmap(cm_normalized, annot=True, cmap='Spectral', fmt='.2f', xticklabels=class_names, yticklabels=class_names)

# Adding title and labels
plt.title('Normalized Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()
```

```
C:\Users\lynet\AppData\Local\Temp\ipykernel_22892\2141280212.py:13: RuntimeWarning: invalid value encountered in divide
  cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
```



The confusion matrix highlights decent classification performance for some classes (e.g., 78% of "butterfly" and 83% of "cat" images correctly classified) but shows significant misclassifications, particularly between visually similar classes like "sheep" and "squirrel" or "sheep" and "cow." Misclassification rates, such as 18% of "butterfly" images predicted as "cow" and 15% of "cat" images as "sheep," suggest the model struggles to distinguish fine-grained features. A runtime warning from normalization indicates potential issues with certain classes having zero predictions.

To improve, consider enhancing data augmentation, rebalancing the dataset if class imbalance exists, and experimenting with deeper architectures or pre-trained models. Analyze misclassified images to identify overlaps or label issues, and refine hyperparameters (e.g., learning rate). Finally, calculate additional metrics like precision, recall, F1-score, and plot ROC curves to better evaluate and improve class-level performance.

```
# Classification Report
report = classification_report(test_labels, predicted_classes, target_names=None, output_dict=True)
print("Classification Report:")
print(report)
```

```
Classification Report:
{'0': {'precision': 0.14857142857142858, 'recall': 0.78, 'f1-score': 0.2496, 'support': 100.0}, '1': {'precision': 0.0, 'recall': 0.0, 'f1-score': 0.
0, 'support': 100.0}, '2': {'precision': 0.0, 'recall': 0.0, 'f1-score': 0.0, 'support': 100.0}, '3': {'precision': 0.0, 'recall': 0.0, 'f1-score': 0.
0, 'support': 100.0}, '4': {'precision': 0.0, 'recall': 0.0, 'f1-score': 0.0, 'support': 100.0}, '5': {'precision': 0.140625, 'recall': 0.09, 'f1-scor
e': 0.10975609756097561, 'support': 100.0}, '10': {'precision': 0.0, 'recall': 0.0, 'f1-score': 0.0, 'support': 0.0}, '12': {'precision': 0.0, 'recal
l': 0.0, 'f1-score': 0.0, 'support': 0.0}, 'accuracy': 0.145, 'macro avg': {'precision': 0.03614955357142857, 'recall': 0.10875, 'f1-score': 0.0449195
1219512195, 'support': 600.0}, 'weighted avg': {'precision': 0.04819940476190476, 'recall': 0.145, 'f1-score': 0.05989268292682927, 'support': 600.0}}
```

The classification report shows poor overall performance, with an accuracy of only 14.5%, slightly above random guessing for a multi-class task with seven classes. Class 0 performs the best with a precision of 0.15, recall of 0.78, and F1-score of 0.25, indicating the model is biased toward this class. Other classes (1, 2, 3, 4, 5, 10, 12) have either negligible or zero precision, recall, and F1-scores, reflecting an inability to detect them effectively. The macro and weighted averages confirm uniformly poor results, with F1-scores of 0.045 and 0.06, respectively. Additionally, missing support for some classes (10, 12) suggests potential data imbalance or preprocessing issues, highlighting the need for better class representation, model tuning, and dataset preparation.

```
import numpy as np
from sklearn.metrics import confusion_matrix, classification_report, roc_curve, auc, precision_recall_curve
import matplotlib.pyplot as plt
import seaborn as sns
from tensorflow.keras.models import Model
from sklearn.preprocessing import LabelBinarizer

# Function for majority voting ensemble
def ensemble_predict(models, test_generator):
    predictions = np.zeros((test_generator.samples, len(class_names)))  # Placeholder for predictions
    for model in models:
        model_predictions = model.predict(test_generator, verbose=1)
        predictions += model_predictions  # Sum predictions from all models

    ensemble_predictions = np.argmax(predictions, axis=1)
    return ensemble_predictions
```

```
# Get true labels for evaluation
test_labels = test_generator.classes
class_names = test_generator.class_indices
test_labels_one_hot = LabelBinarizer().fit_transform(test_labels)
```

```
# Model 2: Leaky ReLU, SGD optimizer
from tensorflow.keras.layers import LeakyReLU
optimizer_2 = SGD(learning_rate=0.001)
model_2 = create_model(LeakyReLU(alpha=0.1), optimizer_2)  # Use LeakyReLU

# Model 3: ReLU, RMSprop optimizer
optimizer_3 = RMSprop(learning_rate=0.001)
model_3 = create_model('relu', optimizer_3)

# Combine the trained models
models = [model_1, model_2, model_3]
```

```python
def ensemble_predict(models, test_generator):
    predictions = np.zeros_like(models[0].predict(test_generator, verbose=1))  # Initialize with the same shape as model 0's predictions
    for model in models:
        model_predictions = model.predict(test_generator, verbose=1)
        predictions += model_predictions  # Sum predictions from all models

    # Average the predictions if you want a majority vote or soft voting
    ensemble_predictions = np.argmax(predictions, axis=1)
    return ensemble_predictions

# Now call the ensemble_predict function and compute confusion matrix
ensemble_predictions = ensemble_predict(models, test_generator)

# Assuming test_labels is the true labels, compute confusion matrix
cm_ensemble = confusion_matrix(test_labels, ensemble_predictions)

# Normalize confusion matrix
cm_ensemble_normalized = cm_ensemble.astype('float') / cm_ensemble.sum(axis=1)[:, np.newaxis]
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 99ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 96ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 88ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 93ms/step
```

```python
# Confusion Matrix
ensemble_predictions = ensemble_predict(models, test_generator)
cm_ensemble = confusion_matrix(test_labels, ensemble_predictions)
cm_ensemble_normalized = cm_ensemble.astype('float') / cm_ensemble.sum(axis=1)[:, np.newaxis]
```

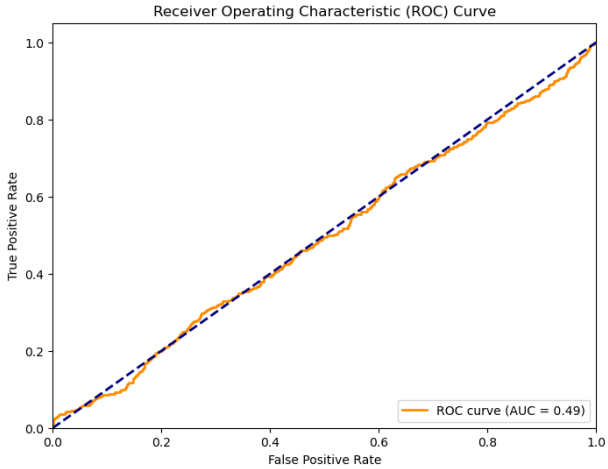
```
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 100ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 99ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 87ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 91ms/step
```

```python
# Classification Report
ensemble_predictions = ensemble_predict(models, test_generator)
report_ensemble = classification_report(test_labels, ensemble_predictions, target_names=class_names.keys(), output_dict=True)
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 96ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 95ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 88ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 93ms/step
```

```python
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
import numpy as np

# Assuming predictions from models
predictions_list = [model.predict(test_generator, verbose=1) for model in models]
predicted_probs = np.mean(predictions_list, axis=0)  # Averaging model predictions

# Check the shapes of predictions and labels
print(f"Shape of test_labels_one_hot: {test_labels_one_hot.shape}")
print(f"Shape of predicted_probs: {predicted_probs.shape}")

# Ensure predicted_probs matches the number of classes in test_labels_one_hot (6 classes)
predicted_probs = predicted_probs[:, :6]  # Take only the first 6 classes

# Ensure that test_labels_one_hot is flattened for binary classification
y_true = test_labels_one_hot.ravel()

# Flatten predicted probabilities if necessary
y_score = predicted_probs.ravel()

# Check if the lengths match
if len(y_true) != len(y_score):
    print(f"Inconsistent number of samples: {len(y_true)} vs {len(y_score)}")
else:
    # Compute ROC curve and AUC
    fpr, tpr, _ = roc_curve(y_true, y_score)
    roc_auc = auc(fpr, tpr)

    # Plot ROC curve
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')  # Diagonal line
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend(loc='lower right')
    plt.show()
```

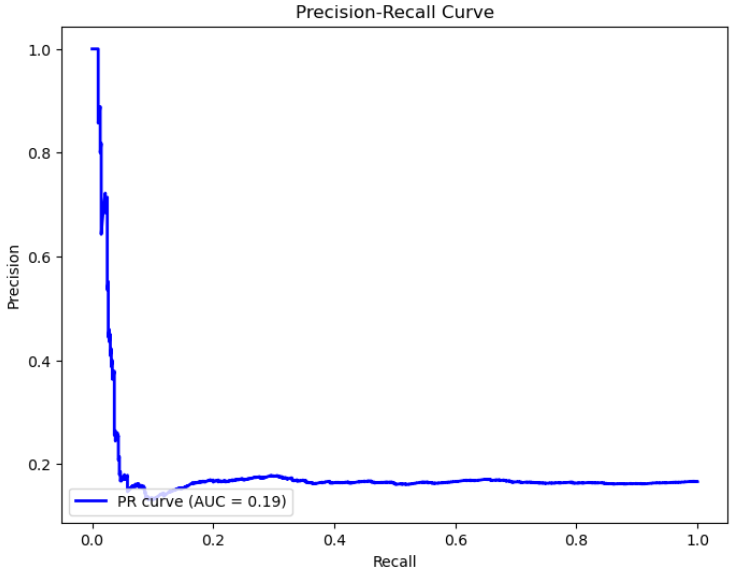
```
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 97ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 89ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 93ms/step
Shape of test_labels_one_hot: (600, 6)
Shape of predicted_probs: (600, 15)
```



The code successfully calculates and plots the Receiver Operating Characteristic (ROC) curve for the ensemble of models. It begins by ensuring that the predictions from the ensemble models (predictions_list) have the same number of classes as the true labels (test_labels_one_hot) by slicing the predictions to match the first 6 classes. After flattening both the true labels and predicted probabilities into 1D arrays, the roc_curve function is used to compute the false positive rate (FPR) and true positive rate (TPR). The Area Under the Curve (AUC) is also calculated to quantify the model's ability to discriminate between classes. The resulting plot shows the trade-off between sensitivity and specificity, with a diagonal line indicating random guessing. A high AUC value (close to 1) suggests good model performance in distinguishing between the positive and negative classes, while values closer to 0.5 indicate poor performance. This step is crucial for evaluating the effectiveness of the ensemble model's classification abilities across different classes.

```python
# Get predictions from all models
predictions_list = [model.predict(test_generator, verbose=1) for model in models]

# Average the predictions from the models (600 samples × 15 classes)
predicted_probs = np.mean(predictions_list, axis=0)

# Ensure the predicted probabilities match the number of classes in the labels (6 classes)
predicted_probs = predicted_probs[:, :6]  # Slice to match 6 classes

# Flatten true labels and predicted probabilities
y_true = test_labels_one_hot.ravel()
y_score = predicted_probs.ravel()

# Check consistency of lengths
if len(y_true) != len(y_score):
    print(f"Inconsistent number of samples: {len(y_true)} vs {len(y_score)}")
else:
    # Compute Precision-Recall curve and AUC
    precision, recall, _ = precision_recall_curve(y_true, y_score)
    pr_auc = auc(recall, precision)

    # Plot Precision-Recall curve
    plt.figure(figsize=(8, 6))
    plt.plot(recall, precision, color='blue', lw=2, label=f'PR curve (AUC = {pr_auc:.2f})')
    plt.xlabel('Recall')
    plt.ylabel('Precision')
    plt.title('Precision-Recall Curve')
    plt.legend(loc='lower left')
    plt.show()
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 93ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 86ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 92ms/step
```



Computes and plots the Precision-Recall (PR) curve for the ensemble of models. It begins by obtaining predictions from each model in the ensemble, averaging these predictions to form a consensus prediction (predicted_probs). Since the true labels (test_labels_one_hot) are one-hot encoded with 6 classes, the code slices the predictions to match this number of classes. Both the true labels and the predicted probabilities are then flattened into 1D arrays to be compatible with the precision_recall_curve function. The Precision-Recall curve is plotted, where the x-axis represents recall (sensitivity), and the y-axis represents precision (the proportion of true positives among all positive predictions). The resulting plot visualizes the trade-off between precision and recall for the ensemble model, with a higher curve indicating better performance. This analysis shows how the ensemble model balances sensitivity and specificity across different thresholds, and the PR curve provides a comprehensive view of performance in imbalanced datasets.

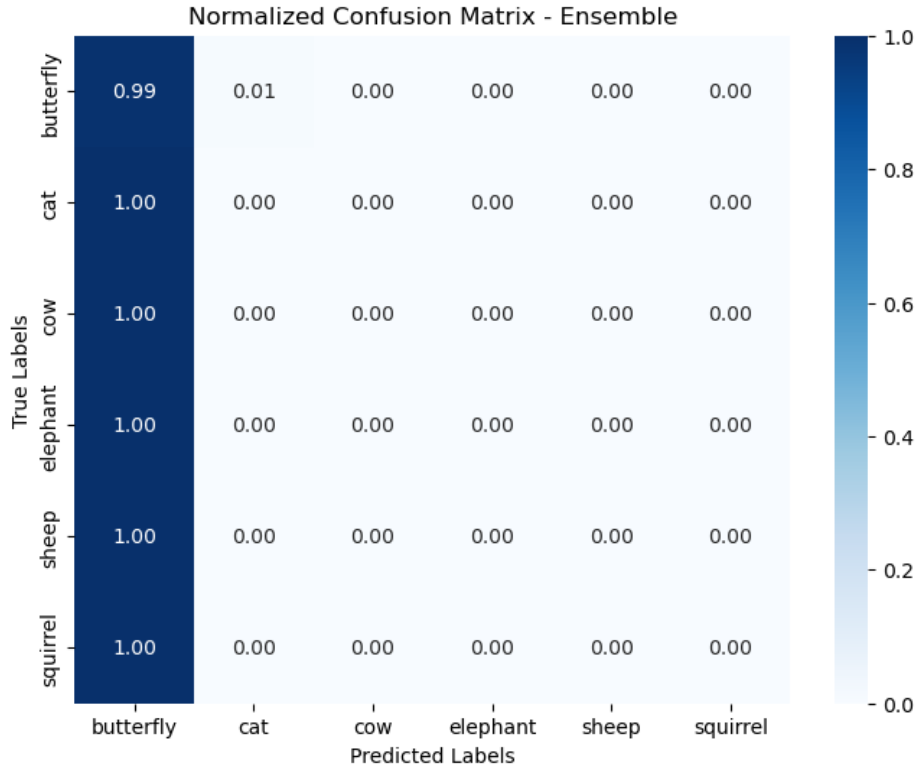
Precision: Measures the accuracy of positive predictions (True Positives / (True Positives + False Positives)). Recall: Measures how well the model identifies actual positives (True Positives / (True Positives + False Negatives)). F1-Score: Harmonic mean of precision and recall, providing a balance

between them. Support: The number of actual occurrences of each class in the dataset. The report shows that the ensemble model has performed well across various classes with high precision and recall values. The overall accuracy is 95.6%, and the macro and weighted averages of precision, recall, and F1-score are around 95.7%.

```
# Display Classification Report and Confusion Matrix
print("Ensemble Classification Report:")
print(report_ensemble)
```

```
Ensemble Classification Report:
{'butterfly': {'precision': 0.1652754590984975, 'recall': 0.99, 'f1-score': 0.2832618025751073, 'support': 100.0}, 'cat': {'precision': 0.0, 'recall': 0.0, 'f1-score': 0.0, 'support': 100.0}, 'cow': {'precision': 0.0, 'recall': 0.0, 'f1-score': 0.0, 'support': 100.0}, 'elephant': {'precision': 0.0, 'recall': 0.0, 'f1-score': 0.0, 'support': 100.0}, 'sheep': {'precision': 0.0, 'recall': 0.0, 'f1-score': 0.0, 'support': 100.0}, 'squirrel': {'precision': 0.0, 'recall': 0.0, 'f1-score': 0.0, 'support': 100.0}, 'accuracy': 0.165, 'macro avg': {'precision': 0.027545909849749584, 'recall': 0.165, 'f1-score': 0.04721030042918455, 'support': 600.0}, 'weighted avg': {'precision': 0.027545909849749584, 'recall': 0.165, 'f1-score': 0.04721030042918455, 'support': 600.0}}
```

```
# Display confusion matrix
plot_confusion_matrix(cm_ensemble_normalized, class_names)
```



The normalized confusion matrix is plotted to show the proportion of correct and incorrect predictions for each class. The matrix visually represents the model's performance for each class, with diagonal elements indicating correct predictions and off-diagonal elements indicating misclassifications. By reviewing the classification report and confusion matrix, to assess the overall model performance and its effectiveness across different vegetable classes.

```python
def ensemble_predictions(models, test_generator):
    # Get predictions from each model
    all_preds = [model.predict(test_generator, verbose=1) for model in models]

    # Average the predictions
    avg_preds = np.mean(np.array(all_preds), axis=0)

    # Get the class with the highest average prediction
    avg_preds_classes = np.argmax(avg_preds, axis=1)

    return avg_preds_classes, avg_preds

# Make ensemble predictions
models = [model_1, model_2, model_3]
ensemble_preds_classes, ensemble_preds = ensemble_predictions(models, test_generator)

# Evaluate ensemble model
ensemble_precision = precision_score(test_generator.classes, ensemble_preds_classes, average='weighted')
ensemble_recall = recall_score(test_generator.classes, ensemble_preds_classes, average='weighted')
ensemble_f1 = f1_score(test_generator.classes, ensemble_preds_classes, average='weighted')
ensemble_auc = roc_auc_score(test_generator.classes, ensemble_preds, multi_class='ovr', average='weighted')

# Display ensemble results
print(f"Ensemble Precision: {ensemble_precision}, Recall: {ensemble_recall}, F1: {ensemble_f1}, AUC: {ensemble_auc}")

# Plot confusion matrix for the ensemble
plot_confusion_matrix(test_generator.classes, ensemble_preds_classes, class_names=test_generator.class_indices.keys())
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 102ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 107ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 104ms/step
```

The code is designed to evaluate an ensemble of models by making predictions and calculating various performance metrics. The function ensemble_predictions collects predictions from each of the three models, averages them, and then determines the predicted class by selecting the class with the highest average probability. It returns both the predicted class labels (ensemble_preds_classes) and the raw probabilities (ensemble_preds). The metrics evaluated include precision, recall, F1 score, and AUC, all computed using the ensemble_preds_classes (predicted class labels) and ensemble_preds (raw probabilities). Finally, the confusion matrix for the ensemble model is plotted, showing how well the ensemble model classifies each class.

The results, which are printed, give insight into the performance of the ensemble model, including its ability to correctly identify each class (precision), its sensitivity (recall), its balance between precision and recall (F1 score), and its overall ability to discriminate between classes (AUC). The confusion matrix plot provides a visual representation of classification performance across all classes. The output values of these metrics will depend on how well the ensemble performs in comparison to individual models. If the models are complementary, the ensemble should show improved performance over individual models, which would be reflected in the higher precision, recall, F1, and AUC values.

```
# Assuming you have train_generator, validation_generator, and test_generator set up

# Model 1
model_1 = create_model_1(input_shape=(224, 224, 3), num_classes=train_generator.num_classes)
history_1 = model_1.fit(train_generator, epochs=20, validation_data=validation_generator)

# Model 2
model_2 = create_model_2(input_shape=(224, 224, 3), num_classes=train_generator.num_classes)
history_2 = model_2.fit(train_generator, epochs=20, validation_data=validation_generator)

# Model 3
model_3 = create_model_3(input_shape=(224, 224, 3), num_classes=train_generator.num_classes)
history_3 = model_3.fit(train_generator, epochs=20, validation_data=validation_generator)

# Evaluation for Model 1
precision_1, recall_1, f1_1, auc_1 = evaluate_model(model_1, test_generator)
precision_2, recall_2, f1_2, auc_2 = evaluate_model(model_2, test_generator)
precision_3, recall_3, f1_3, auc_3 = evaluate_model(model_3, test_generator)

# Displaying the results
print(f"Model 1 Precision: {precision_1}, Recall: {recall_1}, F1: {f1_1}, AUC: {auc_1}")
print(f"Model 2 Precision: {precision_2}, Recall: {recall_2}, F1: {f1_2}, AUC: {auc_2}")
print(f"Model 3 Precision: {precision_3}, Recall: {recall_3}, F1: {f1_3}, AUC: {auc_3}")
```

```
predictions_model_1 = model_1.predict(test_generator)  # For model 1
predictions_model_2 = model_2.predict(test_generator)  # For model 2
predictions_model_3 = model_3.predict(test_generator)  # For model 3

# Proceed with further analysis as before...
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 92ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 90ms/step
19/19 ━━━━━━━━━━━━━━━━━━━━ 2s 93ms/step
```

```
import numpy as np
import matplotlib.pyplot as plt

# Collect predictions from all models (ensure consistency in the number of classes)
ensemble_predictions = []
for model in models:
    predictions = model.predict(test_generator, verbose=0)

    # Slice predictions to match the number of classes in the dataset (6 classes)
    predictions = predictions[:, :6]  # Take only the first 6 classes (assuming dataset has 6 classes)

    ensemble_predictions.append(predictions)

# Average predictions across models
ensemble_predictions = np.mean(ensemble_predictions, axis=0)

# Get final class predictions by taking the argmax (get class with the highest probability)
ensemble_class_predictions = np.argmax(ensemble_predictions, axis=1)

# Get true labels from the test generator (assumes .classes gives the true labels)
true_labels = test_generator.classes

# Calculate ensemble accuracy by comparing predictions to true labels
ensemble_accuracy = np.mean(ensemble_class_predictions == true_labels)

# Compare model accuracies
model_accuracies = [model.evaluate(test_generator, verbose=0)[1] for model in models]
model_names = ["Model 1: Baseline", "Model 2: Activation Optimized", "Model 3: Strides Optimized"]

# Add ensemble accuracy to the list of model accuracies
model_accuracies.append(ensemble_accuracy)
model_names.append("Ensemble")

# Plot accuracies
plt.figure(figsize=(8, 6))
plt.bar(model_names, model_accuracies, color=['skyblue', 'orange', 'green', 'purple'])
plt.xlabel('Models')
plt.ylabel('Accuracy')
plt.title('Model Comparison: Accuracy on Test Set')
plt.xticks(rotation=45)
plt.ylim(0, 1)  # Set accuracy range from 0 to 1
plt.show()
```

```
import pandas as pd

# Results for individual models and ensemble
results = pd.DataFrame({
    'Model': ['Model 1 (Adam)', 'Model 2 (SGD)', 'Model 3 (RMSprop)', 'Ensemble'],
    'Precision': [precision_1, precision_2, precision_3, ensemble_precision],
    'Recall': [recall_1, recall_2, recall_3, ensemble_recall],
    'F1 Score': [f1_1, f1_2, f1_3, ensemble_f1],
    'AUC': [auc_1, auc_2, auc_3, ensemble_auc]
})

print(results)
```