

✓ Practical Image Processing Project

Galaxy10 DECaLS Image Classification

Ahmad Alkahlout

Dataset: Galaxy10 DECaLS

Model Used: CNN and EfficientNetB0

✓ Step 1: Download Dataset from Kaggle

In this step, I downloaded the Galaxy10 DECaLS image dataset from Kaggle. The dataset is used for galaxy image classification.

```
# Download Galaxy10 DECaLS dataset from Kaggle
!kaggle datasets download -d ijlalahmed10/galaxy10-decals
```

```
Warning: Looks like you're using an outdated `kaggle` version (installed: 2.0.2), please consider upgrading to the latest version
Dataset URL: https://www.kaggle.com/datasets/ijlalahmed10/galaxy10-decals
License(s): unknown
Downloading galaxy10-decals.zip to /content
100% 2.55G/2.55G [02:14<00:00, 20.4MB/s]
```

```
# Unzip the downloaded dataset
!unzip -q galaxy10-decals.zip -d galaxy10_decals

print("Dataset unzipped successfully")
```

```
Dataset unzipped successfully
```

```
# Check files inside the dataset folder
import os

for file in os.listdir("galaxy10_decals"):
    print(file)
```

```
Galaxy10_DECaLS.h5
```

✓ Step 2: Read Dataset File

In this step, I read the H5 dataset file and checked the images and labels information.

```
# Read H5 dataset file and check its content
import h5py

file_path = "galaxy10_decals/Galaxy10_DECaLS.h5"

with h5py.File(file_path, "r") as file:
    print("Keys inside the file:", list(file.keys()))
    print("Images shape:", file["images"].shape)
    print("Labels shape:", file["ans"].shape)
```

```
Keys inside the file: ['ans', 'dec', 'images', 'pxscale', 'ra', 'redshift']
Images shape: (17736, 256, 256, 3)
Labels shape: (17736,)
```

✓ Step 3: Load Images and Labels

In this step, I loaded the galaxy images and their labels from the H5 file.

This step helps to check the dataset shape and the number of classes before splitting the data.

```
# Load images and labels from the H5 file
import h5py
import numpy as np

file_path = "galaxy10_decals/Galaxy10_DECaLS.h5"

with h5py.File(file_path, "r") as file:
    images = np.array(file["images"])
```

```

labels = np.array(file["ans"])

print("Images loaded successfully")
print("Images shape:", images.shape)
print("Labels shape:", labels.shape)
print("Number of classes:", len(np.unique(labels)))
print("Classes:", np.unique(labels))

```

```

Images loaded successfully
Images shape: (17736, 256, 256, 3)
Labels shape: (17736,)
Number of classes: 10
Classes: [0 1 2 3 4 5 6 7 8 9]

```

Step 4: Display Sample Images

In this step, I displayed some sample images from the dataset to understand the data before training.

```

# Display random sample images from the dataset
import matplotlib.pyplot as plt
import numpy as np

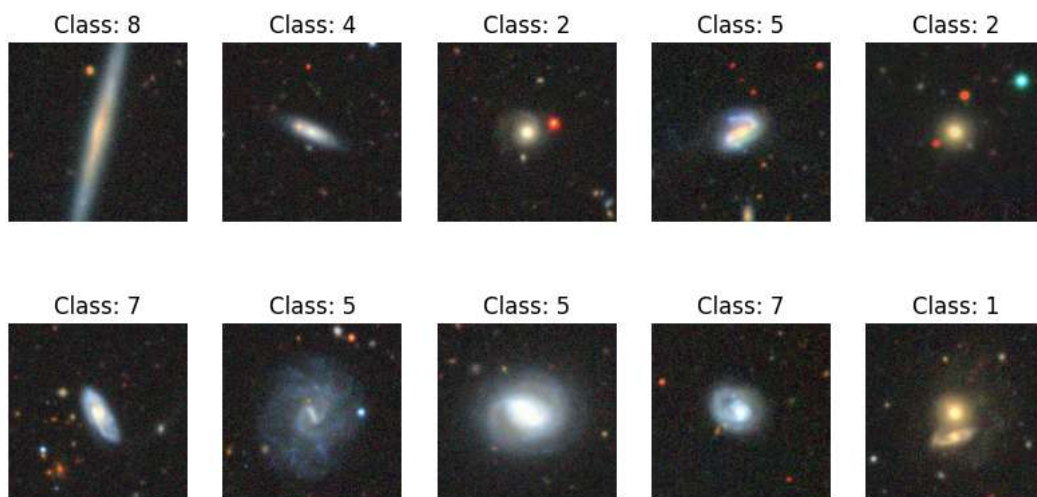
random_index = np.random.choice(len(images), 10, replace=False)

plt.figure(figsize=(10, 5))

for i, index in enumerate(random_index):
    plt.subplot(2, 5, i + 1)
    plt.imshow(images[index])
    plt.title("Class: " + str(labels[index]))
    plt.axis("off")

plt.show()

```



Step 5A: Splitting Dataset

In this step, I split the dataset into training, validation, and testing sets.

The training set is used to train the model, the validation set is used during training, and the test set is used for final evaluation.

```

# Splitting dataset into train, validation, and test sets
from sklearn.model_selection import train_test_split

train_ratio = 0.7
validation_ratio = 0.1
test_ratio = 0.2

# Split data into training and temporary data
X_train, X_temp, y_train, y_temp = train_test_split(
    images, labels, test_size=(1 - train_ratio), random_state=42
)

# Split temporary data into validation and testing data
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=test_ratio / (validation_ratio + test_ratio), random_state=42
)

```

```
print("Training images:", X_train.shape)
print("Validation images:", X_val.shape)
print("Testing images:", X_test.shape)
```

```
Training images: (12415, 256, 256, 3)
Validation images: (1773, 256, 256, 3)
Testing images: (3548, 256, 256, 3)
```

✓ Step 5B: Check Class Distribution

In this step, I checked the number of images in each class before and after splitting the dataset. This helps to understand if the data is balanced between classes.

```
# Check class distribution before and after splitting
import numpy as np

print("Original dataset distribution:")
classes, counts = np.unique(labels, return_counts=True)
for c, n in zip(classes, counts):
    print("Class", c, ":", n)

print("\nTraining dataset distribution:")
classes, counts = np.unique(y_train, return_counts=True)
for c, n in zip(classes, counts):
    print("Class", c, ":", n)

print("\nValidation dataset distribution:")
classes, counts = np.unique(y_val, return_counts=True)
for c, n in zip(classes, counts):
    print("Class", c, ":", n)

print("\nTesting dataset distribution:")
classes, counts = np.unique(y_test, return_counts=True)
for c, n in zip(classes, counts):
    print("Class", c, ":", n)
```

```
Original dataset distribution:
Class 0 : 1081
Class 1 : 1853
Class 2 : 2645
Class 3 : 2027
Class 4 : 334
Class 5 : 2043
Class 6 : 1829
Class 7 : 2628
Class 8 : 1423
Class 9 : 1873
```

```
Training dataset distribution:
Class 0 : 762
Class 1 : 1295
Class 2 : 1857
Class 3 : 1443
Class 4 : 226
Class 5 : 1393
Class 6 : 1284
Class 7 : 1832
Class 8 : 990
Class 9 : 1333
```

```
Validation dataset distribution:
Class 0 : 102
Class 1 : 188
Class 2 : 262
Class 3 : 211
Class 4 : 31
Class 5 : 227
Class 6 : 177
Class 7 : 248
Class 8 : 151
Class 9 : 176
```

```
Testing dataset distribution:
Class 0 : 217
Class 1 : 370
Class 2 : 526
Class 3 : 373
Class 4 : 77
Class 5 : 423
Class 6 : 368
Class 7 : 548
Class 8 : 282
Class 9 : 364
```

✓ Step 6: Applying Data Augmentation

In this step, I applied data augmentation to the training images.

Data augmentation helps the model learn better by creating different versions of the images.

```
# Applying data augmentation
from tensorflow.keras.preprocessing.image import ImageDataGenerator

batch_size = 32

train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=30,
    horizontal_flip=True,
    vertical_flip=True
)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow(X_train, y_train, batch_size=batch_size)
val_generator = test_datagen.flow(X_val, y_val, batch_size=batch_size)
test_generator = test_datagen.flow(X_test, y_test, batch_size=batch_size, shuffle=False)

print("Data augmentation applied successfully")
```

Data augmentation applied successfully

✓ Step 7: Build CNN Model

In this step, I built a simple CNN model for galaxy image classification.

The model contains convolutional layers, pooling layers, and dense layers.

```
# Build CNN model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(256, 256, 3)),
    MaxPooling2D(2, 2),

    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),

    Conv2D(128, (3, 3), activation='relu'),
    MaxPooling2D(2, 2),

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])

model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

model.summary()
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pass an `input`  
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 254, 254, 32)	896
max_pooling2d (MaxPooling2D)	(None, 127, 127, 32)	0
conv2d_1 (Conv2D)	(None, 125, 125, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 62, 62, 64)	0
conv2d_2 (Conv2D)	(None, 60, 60, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 30, 30, 128)	0
flatten (Flatten)	(None, 115200)	0
dense (Dense)	(None, 128)	14,745,728
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290

Total params: 14,840,266 (56.61 MB)
Trainable params: 14,840,266 (56.61 MB)
Non trainable params: 0 (0.00 MB)

✓ Step 8: Train CNN Model

In this step, I trained the CNN model using the training data.

The validation data is used to check the model performance during training.

```
# Train CNN model  
history = model.fit(  
    train_generator,  
    epochs=5,  
    validation_data=val_generator  
)
```

```
Epoch 1/5  
388/388 ————— 170s 414ms/step - accuracy: 0.2139 - loss: 2.0994 - val_accuracy: 0.2781 - val_loss: 1.8938  
Epoch 2/5  
388/388 ————— 152s 392ms/step - accuracy: 0.2845 - loss: 1.8869 - val_accuracy: 0.3553 - val_loss: 1.7578  
Epoch 3/5  
388/388 ————— 153s 394ms/step - accuracy: 0.3172 - loss: 1.8176 - val_accuracy: 0.3723 - val_loss: 1.6735  
Epoch 4/5  
388/388 ————— 152s 391ms/step - accuracy: 0.3460 - loss: 1.7417 - val_accuracy: 0.4270 - val_loss: 1.5683  
Epoch 5/5  
388/388 ————— 152s 392ms/step - accuracy: 0.3905 - loss: 1.6360 - val_accuracy: 0.4659 - val_loss: 1.4554
```

✓ Step 9: Testing CNN Model

In this step, I tested the trained CNN model using the testing dataset.

The test data was not used during training, so it helps to evaluate the model performance.

```
# Test CNN model  
test_loss, test_accuracy = model.evaluate(test_generator)  
  
print("Test Loss:", test_loss)  
print("Test Accuracy:", test_accuracy)
```

```
111/111 ————— 3s 28ms/step - accuracy: 0.4828 - loss: 1.4406  
Test Loss: 1.4406383037567139  
Test Accuracy: 0.4828072190284729
```

✓ Step 10: Evaluation Metrics

In this step, I calculated the accuracy, precision, recall, and F1-score.

I also displayed the classification report for all classes.

```
# Evaluation metrics  
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, classification_report  
import numpy as np
```

```

# Predict classes for test data
predictions = model.predict(test_generator)
y_pred = np.argmax(predictions, axis=1)

# True labels
y_true = y_test

# Calculate metrics
accuracy = accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred, average='weighted', zero_division=0)
recall = recall_score(y_true, y_pred, average='weighted', zero_division=0)
f1 = f1_score(y_true, y_pred, average='weighted', zero_division=0)

print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)

print("\nClassification Report:")
print(classification_report(y_true, y_pred, zero_division=0))

```

```

111/111 ————— 3s 23ms/step
Accuracy: 0.48280721533258175
Precision: 0.45628634587409156
Recall: 0.48280721533258175
F1-score: 0.4366476324303949

Classification Report:

```

	precision	recall	f1-score	support
0	0.00	0.00	0.00	217
1	0.60	0.34	0.43	370
2	0.55	0.65	0.59	526
3	0.45	0.48	0.46	373
4	0.00	0.00	0.00	77
5	0.37	0.06	0.10	423
6	0.43	0.35	0.39	368
7	0.35	0.78	0.48	548
8	0.65	0.88	0.75	282
9	0.68	0.65	0.66	364
accuracy			0.48	3548
macro avg	0.41	0.42	0.39	3548
weighted avg	0.46	0.48	0.44	3548

Step 11: Confusion Matrix

In this step, I displayed the confusion matrix to show the correct and incorrect predictions for each class.

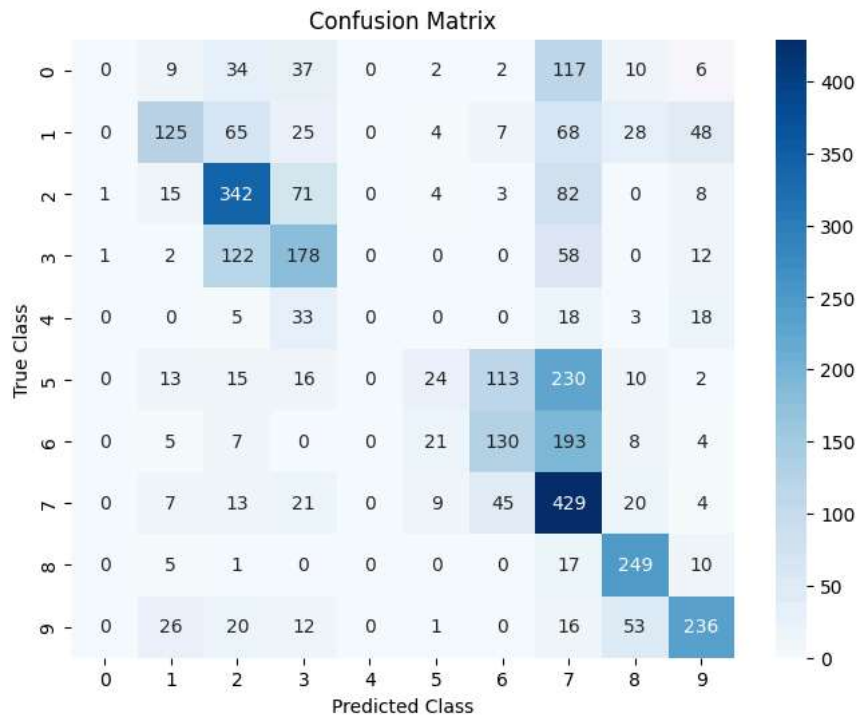
```

# Confusion matrix
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

cm = confusion_matrix(y_true, y_pred)

plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.xlabel("Predicted Class")
plt.ylabel("True Class")
plt.title("Confusion Matrix")
plt.show()

```



Step 12: Build EfficientNetB0 Model

In this step, I used EfficientNetB0 as a pre-trained model for galaxy image classification.

This model was used to improve the classification accuracy compared with the simple CNN model.

```
# Build EfficientNetB0 model
from tensorflow.keras.applications import EfficientNetB0
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense, Dropout

efficientnet_model = Sequential([
    EfficientNetB0(weights='imagenet', include_top=False, input_shape=(256, 256, 3)),
    GlobalAveragePooling2D(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')
])

efficientnet_model.layers[0].trainable = False

efficientnet_model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)

efficientnet_model.summary()
```

Downloading data from https://storage.googleapis.com/keras-applications/efficientnetb0_notop.h5
 16705208/16705208 — 2s 0us/step
 Model: "sequential_1"

Layer (type)	Output Shape	Param #
efficientnetb0 (Functional)	(None, 8, 8, 1280)	4,049,571
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	0
dense_2 (Dense)	(None, 128)	163,968
dropout_1 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 10)	1,290

Total params: 4,214,829 (16.08 MB)
 Trainable params: 165,258 (645.54 KB)

Step 13: Prepare Data for EfficientNetB0

In this step, I prepared image generators for EfficientNetB0.

The training data uses augmentation, while validation and testing data are used without augmentation.

```
# Prepare data for EfficientNetB0
from tensorflow.keras.preprocessing.image import ImageDataGenerator

batch_size = 32

efficient_train_datagen = ImageDataGenerator(
    rotation_range=30,
    horizontal_flip=True,
    vertical_flip=True
)

efficient_test_datagen = ImageDataGenerator()

efficient_train_generator = efficient_train_datagen.flow(
    X_train, y_train,
    batch_size=batch_size,
    shuffle=True
)

efficient_val_generator = efficient_test_datagen.flow(
    X_val, y_val,
    batch_size=batch_size,
    shuffle=False
)

efficient_test_generator = efficient_test_datagen.flow(
    X_test, y_test,
    batch_size=batch_size,
    shuffle=False
)

print("EfficientNetB0 data prepared successfully")
```

EfficientNetB0 data prepared successfully

```
# Train EfficientNetB0 model
efficientnet_history = efficientnet_model.fit(
    efficient_train_generator,
    epochs=5,
    validation_data=efficient_val_generator
)
```

```
Epoch 1/5
388/388 ————— 241s 524ms/step - accuracy: 0.3062 - loss: 1.8919 - val_accuracy: 0.4354 - val_loss: 1.5601
Epoch 2/5
388/388 ————— 152s 392ms/step - accuracy: 0.3885 - loss: 1.6640 - val_accuracy: 0.4676 - val_loss: 1.4995
Epoch 3/5
388/388 ————— 152s 392ms/step - accuracy: 0.4231 - loss: 1.5808 - val_accuracy: 0.4879 - val_loss: 1.3949
Epoch 4/5
388/388 ————— 152s 392ms/step - accuracy: 0.4368 - loss: 1.5370 - val_accuracy: 0.5020 - val_loss: 1.3922
Epoch 5/5
388/388 ————— 153s 394ms/step - accuracy: 0.4486 - loss: 1.5027 - val_accuracy: 0.5014 - val_loss: 1.3724
```

▼ Step 14: Fine-tuning EfficientNetB0 Model

In this step, I fine-tuned the EfficientNetB0 model by training the last layers of the pre-trained model.

This step helps the model learn more specific features from the galaxy images.

```
# Fine-tuning EfficientNetB0 model
import tensorflow as tf

base_model = efficientnet_model.layers[0]
base_model.trainable = True

# Freeze most layers and train only the last layers
for layer in base_model.layers[:-20]:
    layer.trainable = False

efficientnet_model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

```

fine_tune_history = efficientnet_model.fit(
    efficient_train_generator,
    epochs=5,
    validation_data=efficient_val_generator
)

```

```

Epoch 1/5
388/388 ————— 210s 473ms/step - accuracy: 0.4929 - loss: 1.3935 - val_accuracy: 0.5747 - val_loss: 1.1708
Epoch 2/5
388/388 ————— 153s 393ms/step - accuracy: 0.5697 - loss: 1.2076 - val_accuracy: 0.6390 - val_loss: 1.0306
Epoch 3/5
388/388 ————— 152s 391ms/step - accuracy: 0.6071 - loss: 1.1130 - val_accuracy: 0.6734 - val_loss: 0.9453
Epoch 4/5
388/388 ————— 151s 390ms/step - accuracy: 0.6405 - loss: 1.0374 - val_accuracy: 0.6813 - val_loss: 0.9014
Epoch 5/5
388/388 ————— 152s 391ms/step - accuracy: 0.6607 - loss: 0.9907 - val_accuracy: 0.7157 - val_loss: 0.8252

```

Step 15: Testing Fine-tuned EfficientNetB0 Model

In this step, I tested the fine-tuned EfficientNetB0 model using the testing dataset. This test data was not used during training, so it gives the final model performance.

```

# Test fine-tuned EfficientNetB0 model
efficient_test_loss, efficient_test_accuracy = efficientnet_model.evaluate(efficient_test_generator)

print("EfficientNetB0 Test Loss:", efficient_test_loss)
print("EfficientNetB0 Test Accuracy:", efficient_test_accuracy)

```

```

111/111 ————— 23s 204ms/step - accuracy: 0.6894 - loss: 0.9019
EfficientNetB0 Test Loss: 0.9018679857254028
EfficientNetB0 Test Accuracy: 0.6894024610519409

```

Step 16: Evaluation Metrics for EfficientNetB0

In this step, I calculated the accuracy, precision, recall, and F1-score for the fine-tuned EfficientNetB0 model. I also displayed the classification report for all galaxy classes.

```

# Evaluation metrics for EfficientNetB0
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, classification_report
import numpy as np

# Predict classes for test data
efficient_predictions = efficientnet_model.predict(efficient_test_generator)
efficient_y_pred = np.argmax(efficient_predictions, axis=1)

# True labels
efficient_y_true = y_test

# Calculate metrics
efficient_accuracy = accuracy_score(efficient_y_true, efficient_y_pred)
efficient_precision = precision_score(efficient_y_true, efficient_y_pred, average='weighted', zero_division=0)
efficient_recall = recall_score(efficient_y_true, efficient_y_pred, average='weighted', zero_division=0)
efficient_f1 = f1_score(efficient_y_true, efficient_y_pred, average='weighted', zero_division=0)

print("EfficientNetB0 Accuracy:", efficient_accuracy)
print("EfficientNetB0 Precision:", efficient_precision)
print("EfficientNetB0 Recall:", efficient_recall)
print("EfficientNetB0 F1-score:", efficient_f1)

print("\nEfficientNetB0 Classification Report:")
print(classification_report(efficient_y_true, efficient_y_pred, zero_division=0))

```

```

111/111 ————— 16s 78ms/step
EfficientNetB0 Accuracy: 0.6894024802705749
EfficientNetB0 Precision: 0.6858413493947797
EfficientNetB0 Recall: 0.6894024802705749
EfficientNetB0 F1-score: 0.6736980132918212

```

```

EfficientNetB0 Classification Report:

```

	precision	recall	f1-score	support
0	0.49	0.10	0.16	217
1	0.64	0.70	0.67	370
2	0.75	0.87	0.80	526
3	0.71	0.72	0.71	373
4	0.77	0.35	0.48	77
5	0.72	0.61	0.66	423
6	0.70	0.56	0.62	368

7	0.52	0.68	0.59	548
8	0.84	0.88	0.86	282
9	0.80	0.91	0.85	364
accuracy			0.69	3548
macro avg	0.69	0.64	0.64	3548
weighted avg	0.69	0.69	0.67	3548

Step 17: Confusion Matrix for EfficientNetB0

In this step, I displayed the confusion matrix for the fine-tuned EfficientNetB0 model.

The confusion matrix shows how many images were correctly and incorrectly classified for each class.

```
# Confusion matrix for EfficientNetB0
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

efficient_cm = confusion_matrix(efficient_y_true, efficient_y_pred)

plt.figure(figsize=(8, 6))
sns.heatmap(efficient_cm, annot=True, fmt="d", cmap="Blues")
plt.xlabel("Predicted Class")
plt.ylabel("True Class")
plt.title("EfficientNetB0 Confusion Matrix")
plt.show()
```

