Introduction

Skin cancer is one of the most common cancers worldwide, and early detection is critical for effective treatment. Traditional diagnostic methods rely on visual examination and biopsy, which can be time-consuming and require specialist expertise. Recent advancements in deep learning have shown promise in automating and improving the accuracy of skin cancer detection. In this project, we explore the application of convolutional neural networks (CNNs) and transfer learning using EfficientNetB7 to classify different types of skin cancer from dermoscopic images. This research is particularly interesting as it has the potential to assist dermatologists and improve patient outcomes through faster and more accurate diagnoses.

Methods

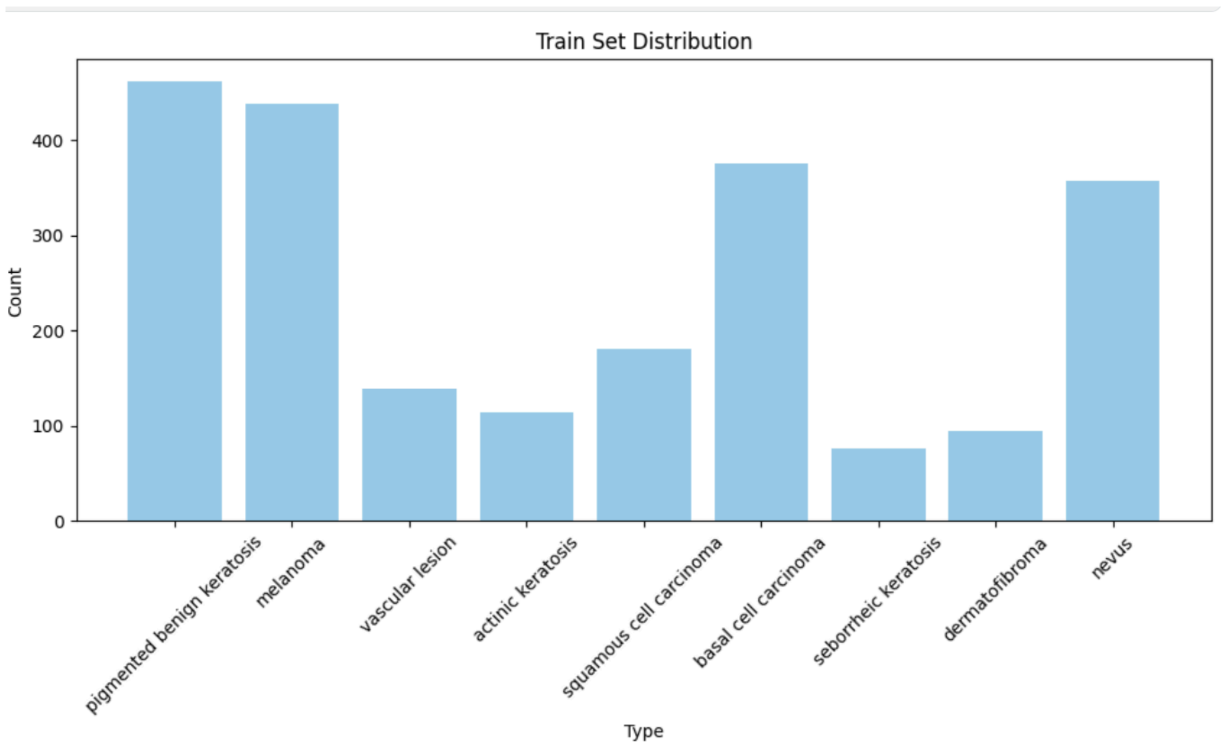
- Dataset Description
    - The dataset used in this study is from the International Skin Imaging Collaboration (ISIC). It was obtained from the Kaggle website, where it was made available to the public. It consists of dermoscopic images categorized into different types of skin lesions, including melanoma, basal cell carcinoma, and others. The dataset is already split into training and testing sets, ensuring that the model's performance can be evaluated on unseen data.
    - Number of classes: 9
      Total images in training set: [number of images]
      Total images in test set: [number of images]

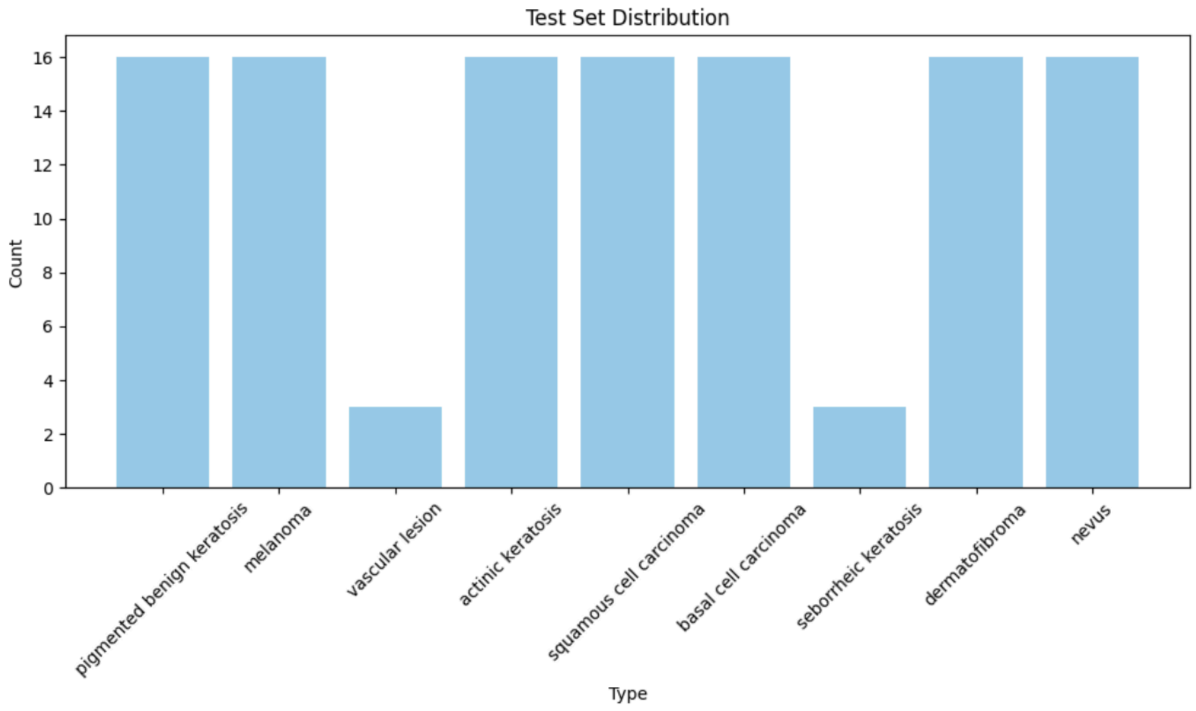
- Data Loading and Preprocessing

  The dataset used in this project was organized into 'Test' and 'Train' folders, each containing subfolders for different classes of skin lesions. Within these subfolders, images corresponding to each lesion type were stored. The paths to these images were captured in a Python dictionary, with class names as key-value pairs, which was then used throughout the project for various operations.

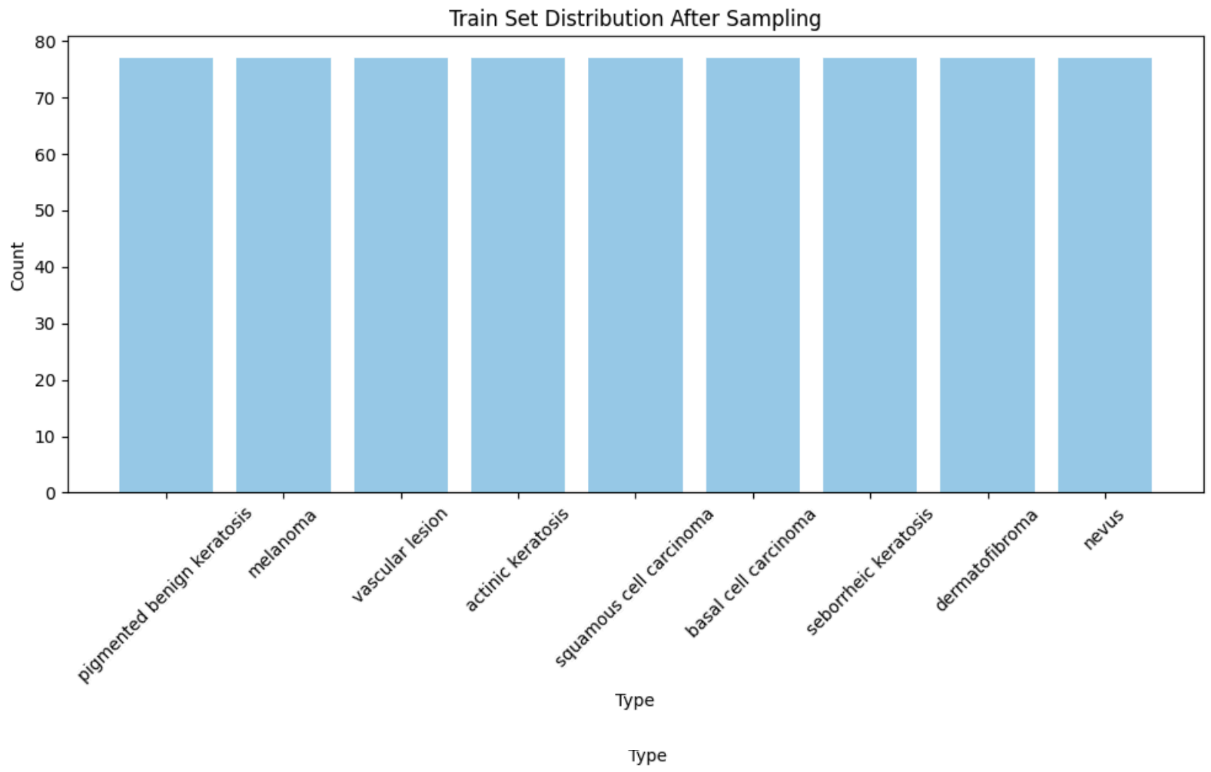
  Upon examining the dataset, it was clear that there was a class imbalance, both in terms of the unequal distribution of images among different classes and the inconsistent ratio of class counts between the test and training sets (Figure 1). To address this, a sampling approach was employed, where the sample size was set to the minimum count of images available in any class. This ensured that each class had an equal number of images, thereby mitigating the class imbalance issue.

  A


Train Set Distribution
Count
Type
400
300
200
100
0
pigmented benign keratosis
melanoma
vascular lesion
actinic keratosis
squamous cell carcinoma
basal cell carcinoma
seborrheic keratosis
dermatofibroma
nevus


Test Set Distribution
Count
Type
16
14
12
10
8
6
4
2
0
pigmented benign keratosis
melanoma
vascular lesion
actinic keratosis
squamous cell carcinoma
basal cell carcinoma
seborrheic keratosis
dermatofibroma
nevus

B



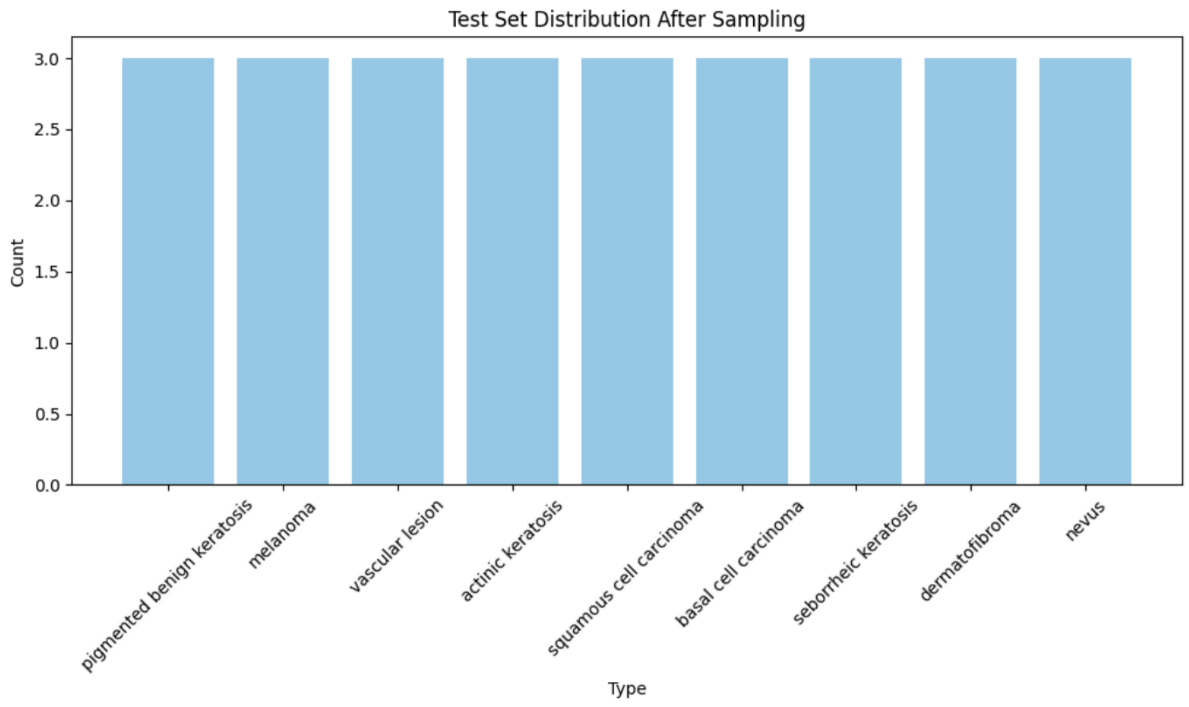


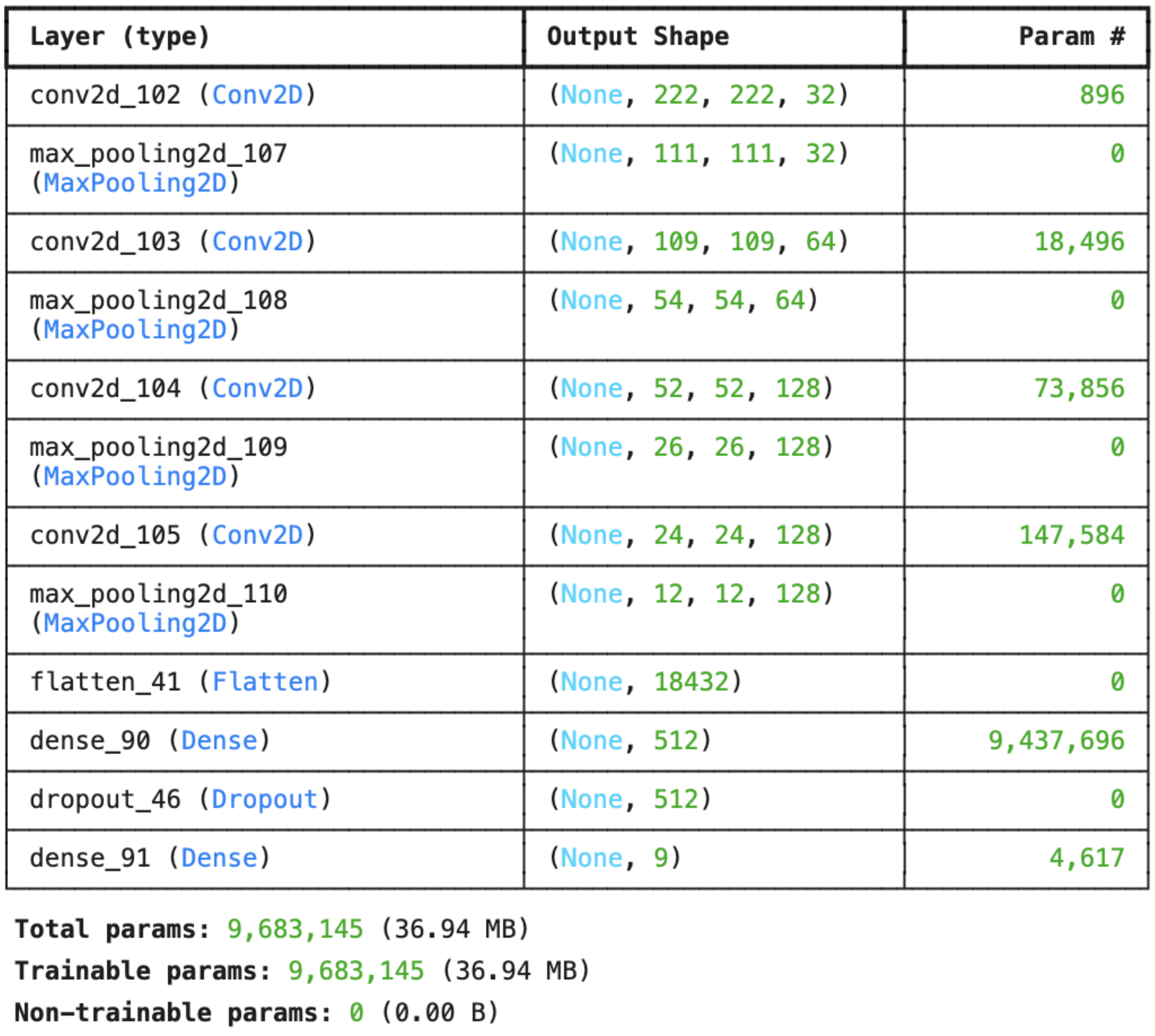
Figure 1: The Frequency of each type of lesion in the test and training subsets. (A) is before sampling and (B) is after.

The test and training subsets then underwent a preprocessing pipeline. All images were resized to 224x224 pixels and normalized to have pixel values between 0 and 1. Data augmentation techniques, including rotation, width and height shifts, shear, zoom, and horizontal flips, were applied to the training images. This increased variability within the dataset and helped to prevent overfitting.

Subsequently, the image arrays were converted into tensors. The use of the expand_dims function in this context was to add an additional dimension to the array, effectively converting it from a 2D to a 3D tensor (height, width, channels). This step was crucial as the deep learning model expected input data in the form of 3D tensors. This comprehensive preprocessing resulted in a set of images ready for training and testing the model (X_train and X_test).

The image paths were used to derive class labels. The class names (lesion names) were first converted into numerical form and then mapped to the image paths, maintaining the order. This process generated y_train and y_test. These labels were then one-hot encoded to match the format required by the model. The X_train and X_test datasets were concatenated to form a complete dataset for further processing. Finally, X_train, X_test, y_train, and y_test were formatted as numpy arrays to be compatible with the model training and evaluation processes.

Model Development



Layer (type)	Output Shape	Param #
conv2d_102 (Conv2D)	(None, 222, 222, 32)	896
max_pooling2d_107 (MaxPooling2D)	(None, 111, 111, 32)	0
conv2d_103 (Conv2D)	(None, 109, 109, 64)	18,496
max_pooling2d_108 (MaxPooling2D)	(None, 54, 54, 64)	0
conv2d_104 (Conv2D)	(None, 52, 52, 128)	73,856
max_pooling2d_109 (MaxPooling2D)	(None, 26, 26, 128)	0
conv2d_105 (Conv2D)	(None, 24, 24, 128)	147,584
max_pooling2d_110 (MaxPooling2D)	(None, 12, 12, 128)	0
flatten_41 (Flatten)	(None, 18432)	0
dense_90 (Dense)	(None, 512)	9,437,696
dropout_46 (Dropout)	(None, 512)	0
dense_91 (Dense)	(None, 9)	4,617

```
Total params: 9,683,145 (36.94 MB)
Trainable params: 9,683,145 (36.94 MB)
Non-trainable params: 0 (0.00 B)
```

Figure 2: The architecture of the custom CNN model.

Two models were used in this project. First, a custom Convolutional Neural Network (CNN) model was developed (Figure 2). This architecture included pooling layers, convolutional layers, dense layers, and a flatten layer. Pooling layers were used to reduce the spatial dimensions of the feature maps, thereby decreasing computational complexity and mitigating the risk of overfitting. Convolutional layers applied filters to the input data to extract relevant features, while dense layers, which are fully connected, performed the final classification. The flatten layer transformed the 2D feature maps into a 1D vector, making it suitable for input into the dense layers.

To prevent overfitting, dropout layers were incorporated, which randomly set a fraction of the input units to zero at each update during training. The Rectified Linear Unit (ReLU) was chosen as the activation function for its ability to introduce non-linearity without suffering from the vanishing gradient problem, unlike sigmoid or tanh. For the output layer, the softmax activation function was used because it is well-suited for multiclass classification problems, as it outputs a probability distribution over the classes.

The model employed categorical cross-entropy as the loss function, appropriate for multiclass classification tasks. The training was conducted over 10 epochs using the Adam optimizer, with a learning rate set to 0.001, to ensure efficient and adaptive learning.

Due to the limited performance of the custom convolutional neural network (CNN), a transfer learning approach using EfficientNetB7 was implemented. EfficientNetB7 is a pre-trained model on the ImageNet dataset, renowned for its efficiency and high performance. Transfer learning involves using a pre-trained CNN model instead of building one from scratch, which is advantageous because the model already has optimized weights from extensive training on a large dataset. This pre-trained model can leverage its learned features and apply them to the new dataset, enhancing performance and reducing the time required for training.

EfficientNetB7 was selected for its demonstrated effectiveness in previous applications. The process began by freezing all layers of EfficientNetB7 to retain the pre-trained weights and prevent them from being altered during initial training. Custom layers were then added to tailor the model for the specific task of lesion classification. These included a Flatten layer to transform the 3D tensor outputs from EfficientNetB7 into a 1D tensor, followed by Dense layers to perform classification.

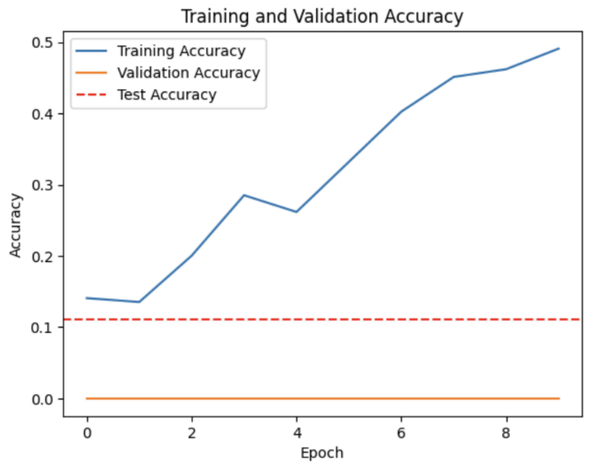
After the initial training with the frozen layers, fine-tuning was performed by unfreezing the top layers of EfficientNetB7. This allowed the model to adjust the pre-trained weights slightly to adjust to the characteristics of the new dataset. During fine-tuning, a lower learning rate was used to ensure that the pre-trained weights were only minimally adjusted, preserving the valuable features learned from the ImageNet dataset.

For training and evaluation, the Adam optimizer was used with an initial learning rate of 0.001, which was further reduced during fine-tuning. The categorical cross-entropy loss function was chosen to measure the performance of the model in multi-class classification tasks. Accuracy was used as the primary metric to evaluate the model's performance.

Results

The results of the model training and evaluation are presented below.

A



B

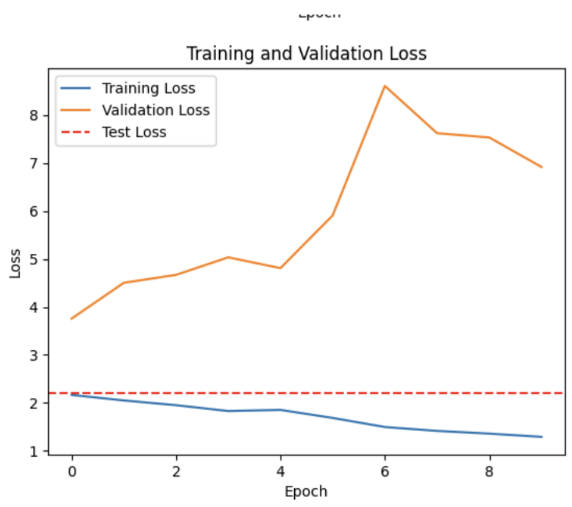
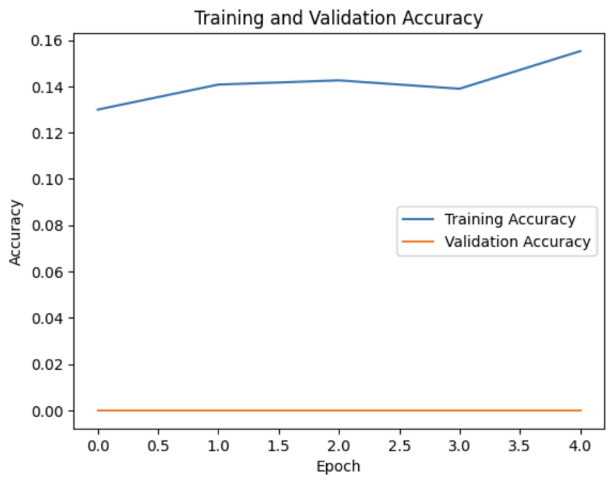


Figure 3: Training and validation metrics over epochs for the custom neural network model. (A) is the accuracy graph and (B) is the loss graph. The orange line is the validation loss and the blue line is the training loss. The red line is the test loss obtained from the model after it was finished training. The loss function is categorical cross-entropy.

A



B

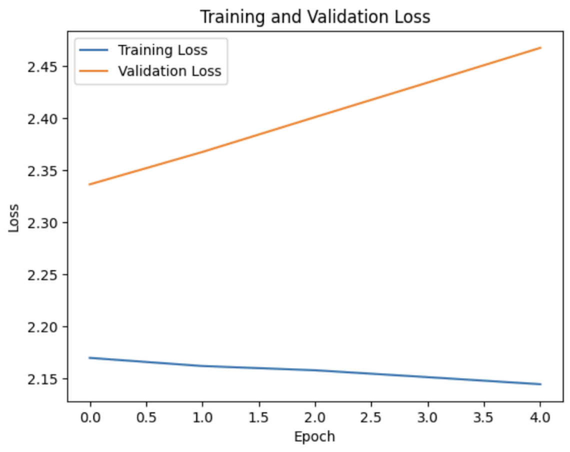
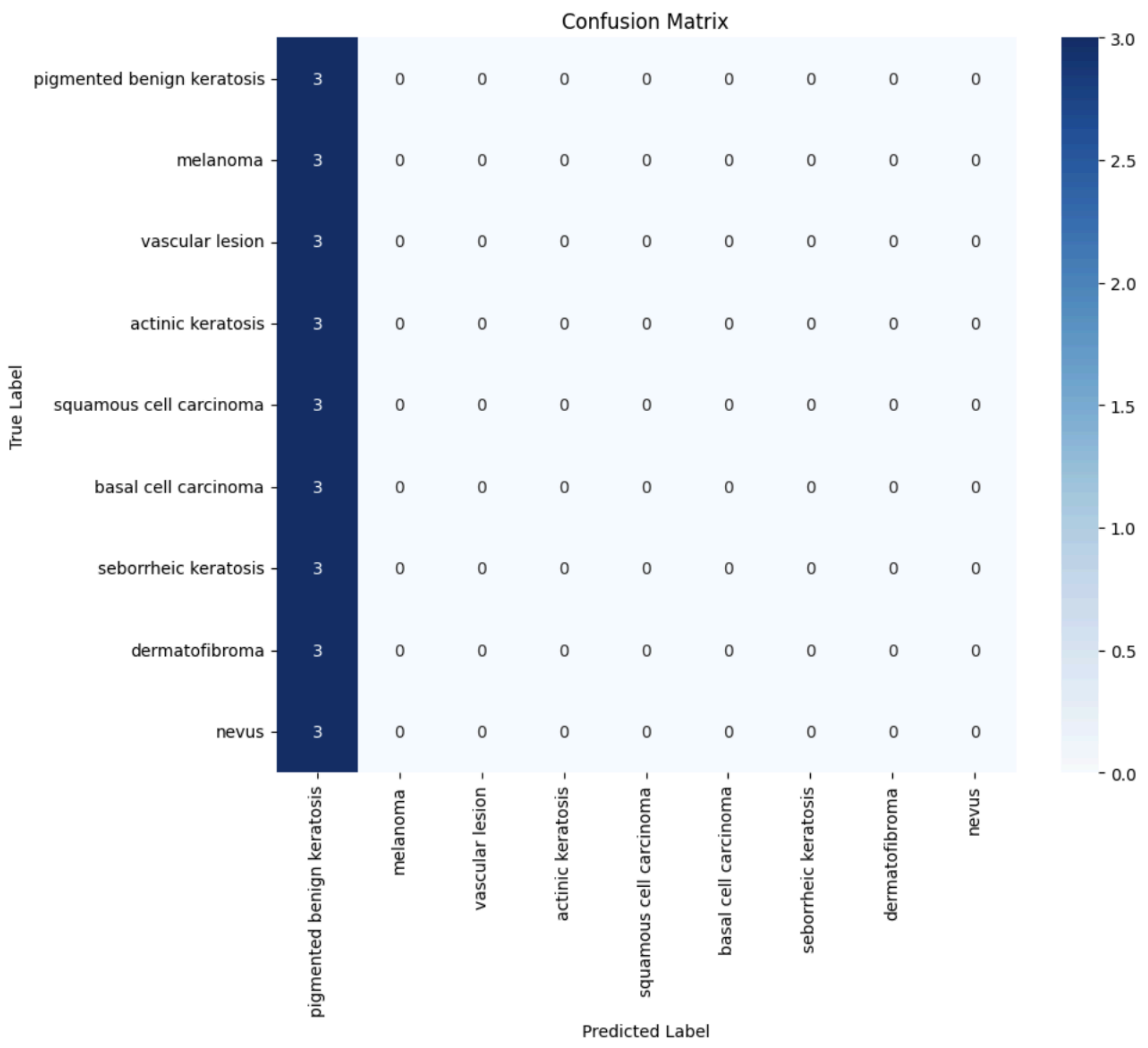


Figure 4: Training and validation metrics over epochs for the EfficientNetB7 model. (A) is the accuracy graph and (B) is the loss graph. The orange line is the validation loss and the blue line is the training loss. The loss function is categorical cross-entropy.



	pigmented benign keratosis	melanoma	vascular lesion	actinic keratosis	squamous cell carcinoma	basal cell carcinoma	seborrheic keratosis	dermatofibroma	nevus
pigmented benign keratosis	3	0	0	0	0	0	0	0	0
melanoma	3	0	0	0	0	0	0	0	0
vascular lesion	3	0	0	0	0	0	0	0	0
actinic keratosis	3	0	0	0	0	0	0	0	0
squamous cell carcinoma	3	0	0	0	0	0	0	0	0
basal cell carcinoma	3	0	0	0	0	0	0	0	0
seborrheic keratosis	3	0	0	0	0	0	0	0	0
dermatofibroma	3	0	0	0	0	0	0	0	0
nevus	3	0	0	0	0	0	0	0	0

Figure 5: Confusion matrix for the custom neural network model. This result is after the model has learned for 10 epochs.

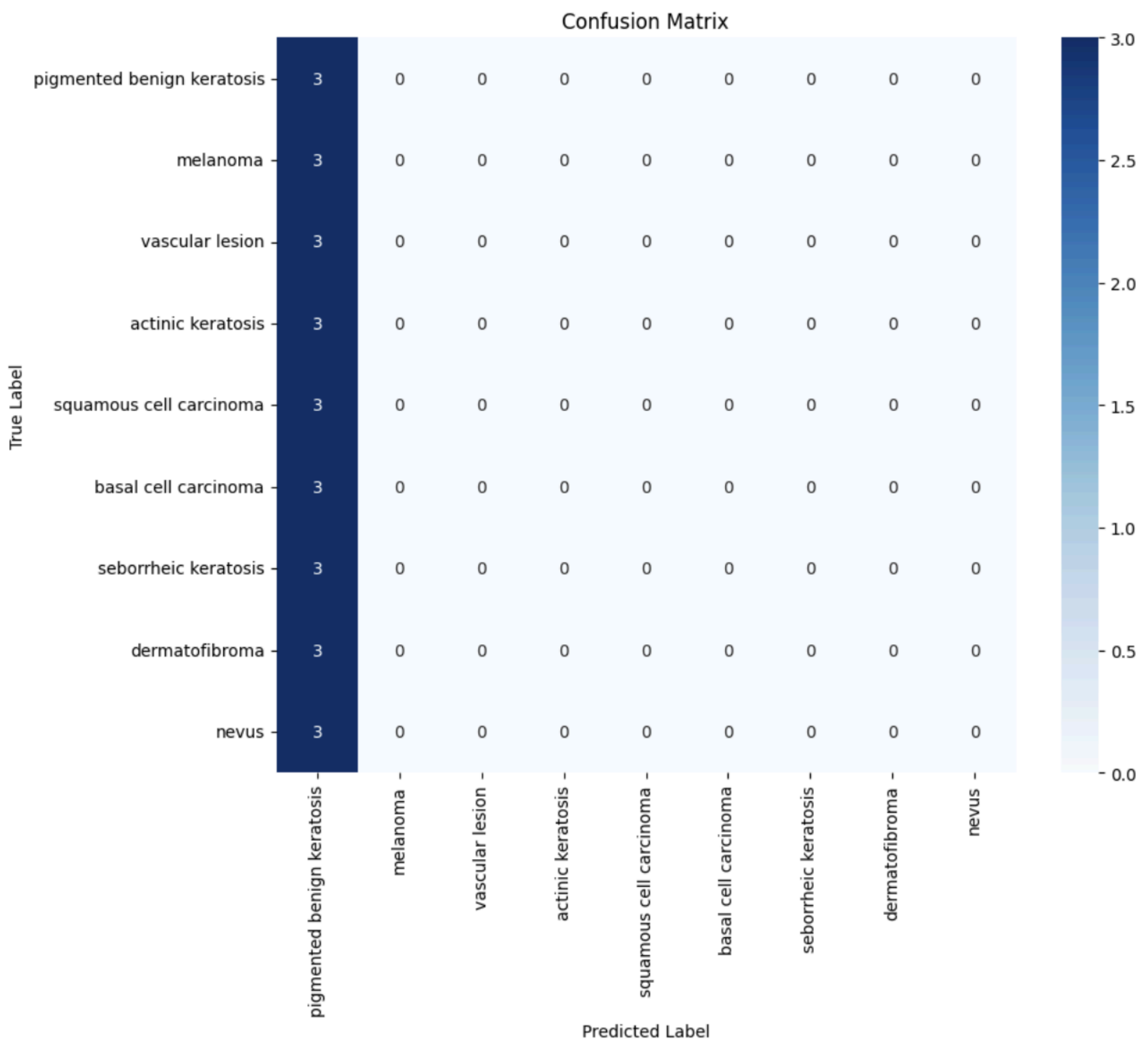


Figure 6: Confusion matrix for the EfficientNetB7 model. This result is after the model has learned for 5 epochs.

Discussion

The accuracy for the custom neural network model after 10 epochs was 48.23%, while the accuracy for the transfer model after 5 epochs was 14.03%. Nevertheless, this does not indicate that the model is performing well. According to Figures 5 and 6, both models are classifying all the images as "pigment benign keratosis", incorrectly. Furthermore, the validation accuracy for both models remains at 0 (Figure 3, 4). When the validation accuracy remains at 0 while the validation loss increases per epoch, it indicates that the model is struggling to learn effectively. This situation suggests that the model is failing to generalize well to unseen data. Several factors could be contributing to this issue. Firstly, the model might not be learning meaningful representations from the training data due to its architecture being too simple or too complex. Additionally, the model may not be suitable for the given task or dataset, and adjusting hyperparameters such as learning rate and batch size might be necessary. Furthermore, overfitting might be occurring, where the model is memorizing the training data instead of learning to generalize from it. Or, most likely, the preprocessing could have occurred incorrectly, hence affecting the performance of both models in the same manner (Figures 5 and 6).

To address this, it's essential to test the code further to see if there is any error with preprocessing. Additionally, it is important to experiment with different model architectures, regularization techniques, and data augmentation methods. This could not be done for this project because of limitations of time and RAM. Having more CPU, or even GPU access, can help make the model run faster and open up oppurtunities for hyper-parameter tuning. Cross-validation can help assess the model's generalization performance while optimizing hyperparameters and applying early stopping can prevent overfitting. Ensuring consistent data preprocessing is also crucial to avoiding issues stemming from data mismatch. Through these steps, it's possible to improve the model's performance.

Python code:

```
# %%
import os
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import random

from PIL import Image
import numpy as np
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import tensorflow as tf

import keras
from keras.utils import to_categorical
from keras.models import Sequential
from tensorflow.keras.models import Model
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.regularizers import L1, L2
from keras.optimizers import SGD, Adam
from sklearn.metrics import confusion_matrix
import seaborn as sns

from tensorflow.keras.applications import EfficientNetB7

# %% [markdown]
# # Import files from dataset.
# ### The directory names have type of mole/cancer

# %%
def get_folder_contents(root_folder):
    folder_contents = {}
    for subdir, _, files in os.walk(root_folder):
        subfolder_name = os.path.basename(subdir)
        if subfolder_name != os.path.basename(root_folder):  # Ignore the root folder itself
            folder_contents[subfolder_name] = []
            for file in files:
                if file.lower().endswith(('.png', '.jpg', '.jpeg', '.gif', '.bmp', '.tiff')):
                    full_path = os.path.join(subdir, file)
                    folder_contents[subfolder_name].append(full_path)
    return folder_contents

# %%
def get_image_paths(image_dict):
    image_paths = []
    for key, value in image_dict.items():
        image_paths.extend(value)
    return image_paths

# %%
def display_images(contents):
    for subfolder, images in contents.items():
        print(f"Subfolder: {subfolder}")
        images_to_display = images[:5]  # Limit to top 5 images
        fig, axes = plt.subplots(1, len(images_to_display), figsize=(15, 5))
        fig.suptitle(subfolder, fontsize=16)
        for ax, image_path in zip(axes, images_to_display):
            img = mpimg.imread(image_path)
            ax.imshow(img)
            ax.axis('off')
        plt.show()

# %%
test_data_path = '/kaggle/input/skin-cancer-dataset/Skin cancer ISIC The International Skin Imaging
Collaboration/Test'
test_contents = get_folder_contents(test_data_path)
test_image_paths = get_image_paths(test_contents)

train_data_path = '/kaggle/input/skin-cancer-dataset/Skin cancer ISIC The International Skin
Imaging Collaboration/Train'
train_contents = get_folder_contents(train_data_path)
train_image_paths = get_image_paths(train_contents)

# print('Test Images')
# for subfolder, images in contents.items():
#     print(f"Subfolder: {subfolder}")
#     for image in images:
#         print(f"  {image}")

# %%
# test_contents

# %%
print('Testing Data--First 5 images')
display_images(test_contents)

# %%
print('Training Data--First 5 images')
display_images(train_contents)

# %%
```

```
def plot_histogram(data, title):
    plt.figure(figsize=(10, 6))
    plt.bar(range(len(data)), list(data.values()), align='center', color='skyblue')
    plt.xticks(range(len(data)), list(data.keys()), rotation=45)
    plt.xlabel('Type')
    plt.ylabel('Count')
    plt.title(title)
    plt.tight_layout()
    plt.show()

# %%
train_histogram = {key: len(value) for key, value in train_contents.items()}
test_histogram = {key: len(value) for key, value in test_contents.items()}

plot_histogram(train_histogram, 'Train Set Distribution')
plot_histogram(test_histogram, 'Test Set Distribution')

# %% [markdown]
# ### There is uneven distribution for some classes. Need to correct.

# %%
def sample_data(data, sample_size):
    sampled_data = {}
    for key, value in data.items():
        sampled_data[key] = random.sample(value, sample_size) if len(value) > sample_size else value
    return sampled_data

# %%
min_count_train = min(len(images) for images in train_contents.values())
min_count_test = min(len(images) for images in test_contents.values())

sampled_train_contents = sample_data(train_contents, min_count_train)
sampled_test_contents = sample_data(test_contents, min_count_test)

# %%
# sampled_train_contents
# %%
class_names = list(sampled_train_contents.keys())
class_names = {v: k for k, v in enumerate(class_names)}

class_names

# %%
train_histogram = {key: len(value) for key, value in sampled_train_contents.items()}
test_histogram = {key: len(value) for key, value in sampled_test_contents.items()}

plot_histogram(train_histogram, 'Train Set Distribution After Sampling')
plot_histogram(test_histogram, 'Test Set Distribution After Sampling')

# %%
for key, value in sampled_train_contents.items():
    print(f"Key: {key}, Number of Values: {len(value)}")

# %%
test_image_paths = get_image_paths(sampled_test_contents)
train_image_paths = get_image_paths(sampled_train_contents)
print('done')

# %% [markdown]
# # Data preprocessing

# %%
def resize_image(image_path, target_size=(224, 224)):
    with Image.open(image_path) as img:
        img = img.resize(target_size)
        return img

def normalize_image(image):
    image_array = np.array(image) / 255.0
    return image_array
```

```
# %%
datagen = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest'
)

def augment_image(image_array):
    image_array = image_array.reshape((1,) + image_array.shape)  # Reshape for ImageDataGenerator
    aug_iter = datagen.flow(image_array)
    return next(aug_iter)[0]

# %%
def preprocess_image(image_paths, target_size=(224, 224)):

    image_tensors = []

    for image_path in image_paths:
        img = Image.open(image_path)
        img = img.resize(target_size)
        img_array = np.array(img) / 255.0
        img_array = augment_image(img_array)

        img_tensor = tf.convert_to_tensor(img_array, dtype=tf.float32)
        img_tensor = tf.expand_dims(img_tensor, axis=0)

        image_tensors.append(img_tensor)

    return image_tensors

# %%
processed_image_train = preprocess_image(train_image_paths)
```

```
processed_image_test = preprocess_image(test_image_paths)

# %%
def assign_class_labels(sampled_contents, class_names):
    y = []
    for class_name, image_paths in sampled_contents.items():
        class_index = class_names.index(class_name)
        y.extend([class_index] * len(image_paths))
    return y

# %%
X_train = processed_image_train
X_test = processed_image_test

class_names = list(sampled_train_contents.keys())
y_train = assign_class_labels(sampled_train_contents, class_names)
y_test = assign_class_labels(sampled_test_contents, class_names)
print(y_train)
print(y_test)

y_train = to_categorical(y_train, num_classes=9)
y_test = to_categorical(y_test, num_classes=9)

print(y_train)
print(y_test)

print('done')

# %%
X_train_concatenated = tf.concat(X_train, axis=0)
X_train = np.array(X_train_concatenated)
#X_train = np.squeeze(X_train, axis=1)
y_train = np.array(y_train)

X_test_concatenated = tf.concat(X_test, axis=0)
X_test = np.array(X_test_concatenated)
```

```
#X_train = np.squeeze(X_train, axis=1)
y_test = np.array(y_test)

X_test = np.array(X_test)
y_test = np.array(y_test)

# %%
print(len(y_test))
print(len(X_test))

print(len(y_train))
print(len(X_train))

# %% [markdown]
# # Model training and fitting

# %%
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 3)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))  # Add dropout layer for regularization
model.add(Dense(9, activation='softmax'))

# Compile the model with Adam optimizer
adam = tf.keras.optimizers.Adam(learning_rate=0.001)  # Use Adam optimizer with a smaller learning
rate
model.compile(optimizer=adam, loss='categorical_crossentropy', metrics=['accuracy'])
```

```
# Compile the model
adam = Adam(learning_rate=0.001)
model.compile(optimizer=adam,
          loss='categorical_crossentropy',
          metrics=['accuracy'])

# Display model summary
model.summary()

# %%
print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)
print("X_train type:", X_train.dtype)
print("y_train type:", y_train.dtype)

# %%
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2, verbose=1,
shuffle=True)

# %%
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=1)
print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)

# %%
# Plot training and validation accuracy per epoch
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.axhline(y=test_accuracy, color='r', linestyle='--', label='Test Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

```
# Plot training and validation loss per epoch
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.axhline(y=test_loss, color='r', linestyle='--', label='Test Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# %%
# history = model.fit(X_train, y_train, epochs=10, batch_size=32, verbose=1, shuffle=True)

# # After each epoch, evaluate the model on the testing data
# for epoch in range(10):
#     model.fit(X_train, y_train, epochs=1, batch_size=32, verbose=1, shuffle=True)
#     loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
#     print("Epoch {}, Testing Loss: {}, Testing Accuracy: {}".format(epoch+1, loss, accuracy))

# %%
# import matplotlib.pyplot as plt

# # Lists to store training and testing loss
# train_loss = []
# test_loss = []

# # Training loop
# for epoch in range(10):
#     history = model.fit(X_train, y_train, epochs=1, batch_size=32, verbose=1, shuffle=True)

#     # Append training loss
#     train_loss.append(history.history['loss'][0])

#     # Evaluate the model on the testing data
#     loss, _ = model.evaluate(X_test, y_test, verbose=0)
#     # Append testing loss
#     test_loss.append(loss)

#     print("Epoch {}, Testing Loss: {}".format(epoch+1, loss))

# # Plotting
# plt.plot(range(1, 11), train_loss, label='Training Loss')
# plt.plot(range(1, 11), test_loss, label='Testing Loss')
# plt.xlabel('Epochs')
# plt.ylabel('Loss')
# plt.title('Training and Testing Loss')
# plt.legend()
# plt.show()

# %%
# Predictions on the test set
y_pred = model_transfer.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_test, axis=1)

# Confusion matrix
conf_matrix = confusion_matrix(y_true_classes, y_pred_classes)

# Plotting the confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=class_names,
yticklabels=class_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()

# %% [markdown]
#

# %% [markdown]
# ### Model is not performing very well. Can use transfer learning to get better accuracy.

# %% [markdown]
# # Transfer Learning

# %%
img_width, img_height = 224, 224
input_shape = (img_width, img_height, 3)
num_classes = 9

base_model = EfficientNetB7(weights='imagenet', include_top=False, input_shape=input_shape)

# Freeze the layers of the pre-trained model
for layer in base_model.layers:
    layer.trainable = False

x = base_model.output
x = Flatten()(x)
x = Dense(128, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dropout(0.5)(x)
predictions = Dense(num_classes, activation='softmax')(x)

model_transfer = Model(inputs=base_model.input, outputs=predictions)

# Compile the model
adam = Adam(learning_rate=0.001)
model_transfer.compile(optimizer=adam, loss='categorical_crossentropy', metrics=['accuracy'])

# %%
model_transfer.summary()

# %%
history_2 = model_transfer.fit(X_train, y_train, epochs=5, batch_size=32, validation_split=0.2,
verbose=1, shuffle=True)
```

```
# %%
test_loss, test_accuracy = model_transfer.evaluate(X_test, y_test, verbose=1)
print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)

# %%
# Plot training and validation accuracy per epoch
plt.plot(history_2.history['accuracy'], label='Training Accuracy')
plt.plot(history_2.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Plot training and validation loss per epoch
plt.plot(history_2.history['loss'], label='Training Loss')
plt.plot(history_2.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# %%
# Predictions on the test set
y_pred = model_transfer.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true_classes = np.argmax(y_test, axis=1)

# Confusion matrix
conf_matrix = confusion_matrix(y_true_classes, y_pred_classes)

# Plotting the confusion matrix
plt.figure(figsize=(10, 8))
```

```
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=class_names,
yticklabels=class_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix For Efficient Net')
plt.show()
```