Skin Cancer Detection Using Diversified CNN Architectures

PROFESSOR: GRIGORIEV, ANDREI

12.11.2024

By:
Niyat Habtom Seghid: 19967
Belsabel Teklemariam Woldemichael: 20052
Feven Belay Araya: 20027

Project Description

Problem Statement

The problem we will be solving in the project is human skin cancer detection using deep learning. We chose this specific idea because we found it interesting, and more importantly since skin cancer is one of the most common cancers worldwide, with millions of new cases diagnosed annually. Hence, early detection of this cancer is very important as it significantly improves survival rates, especially for melanoma, which is the deadliest form of skin cancer. So applying advanced deep learning solutions to tackle this problem is of huge importance.

Solution/Approach

The approach we followed is that we considered a minimum of three scholarly papers on the topic and applied them ourselves (built the models) and then we evaluated and compared the performance of these models to suggest the best performing one. For each of the papers the methods and the tools the authors have used are below.

1. **Analysis of Skin Lesion Images with Deep Learning:**

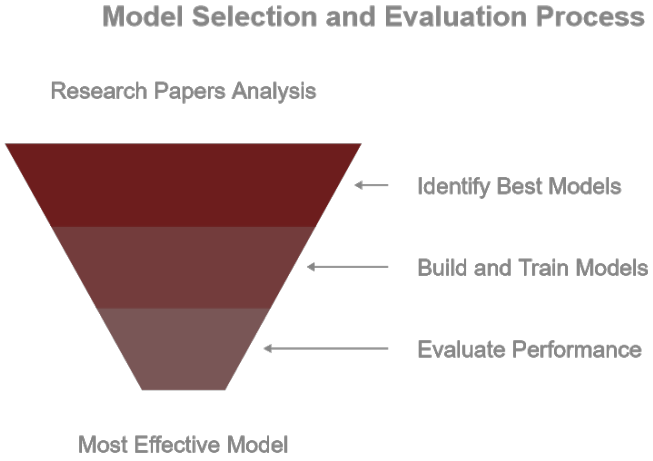
This research uses multiple datasets. The ISIC-2019 Challenge dataset (25,331 images) is used as the primary source, along with BCN 20000, HAM10000, and MSK datasets, totaling in 32,748 training images. They also used supporting datasets which included PH2 database, SKINL2, SD-198, 7-point criteria database, and MED-NODE dataset to increase diversity.These datasets contain dermatoscopic images of 7 different skin lesion categories which names are melanocytic nevi, melanoma, benign keratosis, basal cell carcinoma, actinic keratoses, vascular lesions, and dermatofibroma.For the implementation they used several tools and libraries, primarily Python with TensorFlow, and Keras for deep learning implementations, along with supporting libraries like NumPy, Pandas, OpenCV, and Scikit-learn for data handling and processing. Moreover the paper evaluates five major CNN architectures which are new to us: Inception-ResNet-v2, SE-ResNeXt-101 (32x4d), NASNet-A-Large, EfficientNet-B4, and EfficientNet-B5. The authors pre trained all the models on ImageNet and then fine-tuned each for the specific task of skin lesion classification.

2. **An Interpretable Deep Learning Approach for Skin Cancer Categorization:**

In this study, the authors used four pre-trained Convolutional Neural Networks (CNNs) which are InceptionResNetV2, Xception,EfficientNetV2 Small, and EfficientNetV2 Medium.These models were trained specifically for skin cancer detection, which was done using more advanced AI methods of SmoothGrad and Faster Score-CAN. Adam optimizer was used with ReduceLROnPlateau callback to automatically control the learning rate for the best performance. The model's training and the image preprocessing part was done in TensorFlow/Keras and OpenCV, respectively.The authors used the HAM10000 dataset with 10,015 dermoscopic images belonging to seven skin lesion classes.

3. **Methods to Classify Skin Lesions using Demroscopic images:**

This research paper utilized Convolutional Neural Networks (CNNs), specifically EfficientNet and U-Net to enhance diagnostic accuracy. They used the ISIC 2019 and HAM10000 datasets, enriched with additional smaller datasets for training diversity. TensorFlow and Keras are used for model development, with Python, OpenCV, and Scikit-learn supporting image processing and performance evaluation. Also, to improve model generalization Data augmentation techniques like rotation and scaling were applied, and to enhance skin cancer diagnostics a 10-fold cross-validation strategy was used to ensure reliable model evaluation, leveraging advanced technologies.
Note: since in each paper the authors train several models we took the best performing model from each of their work and built that one specifically by following their methodologies.



Implementation Details

The three CNN model architectures that we selected for implementation are the following.

A. XceptionNet
B. EfficientNet-B5
C. Dual-Path CNN

The main objective was to classify skin lesions into two main categories based on their potential severity:

- Melanoma (Malignant): A serious form of skin cancer that can spread to other parts of the body, requiring early detection and treatment.
- Non-Melanoma (Benign): Less aggressive skin lesions that are generally not life-threatening and less likely to spread.

Moreover, in some models we have further classified the lesions upto 9 classes categories which include Melanoma, Melanocytic nevus, Basal cell carcinoma, Actinic keratosis, Benign keratosis (solar lentigo / seborrheic keratosis / lichen planus-like keratosis), Dermatofibroma, Vascular lesion, Squamous cell carcinoma, and None of the others.

I. First Approach – XceptionNet

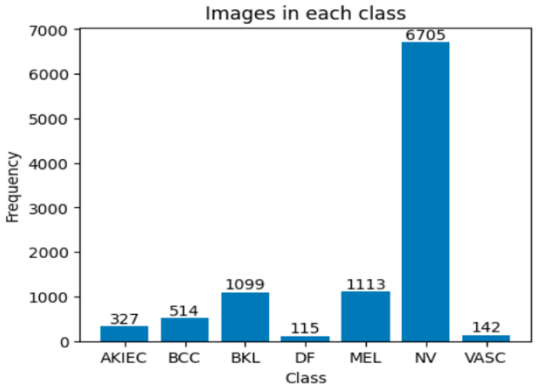
Xception is a kind of neural network architecture for image recognition. The name X-Inception means Extreme Inception because it goes even further than the Inception models of Google and makes the design even better and more effective.
A specific type of a convolution, called depthwise separable, which is an effective approach to processes images is used in Xception. Think of it as breaking down a complex task into smaller, simpler steps rather than performing spatial and feature computations simultaneously together. It divides the two into two steps: one for using the spatial dimensions (the filter) and one for integrating the features (the channel). This greatly reduces the number of parameters but the model is still accurate and this makes it fast and lightweight.
Unlike some other models for convolutional neural networks, Xception is designed to solve large-scale problems and complex image recognition at that. It's generally good for transfer learning because its weights learned on large data such as ImageNet can be used for many different tasks.

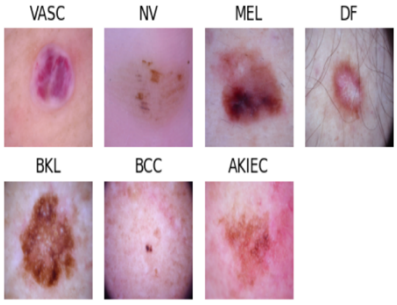
I. Dataset

The dataset is designed for a multi-class image classification task, consisting of 10,015 images.It contains 7 distinct classes:

1. AKIEC
2. BCC
3. BKL
4. DF
5. MEL
6. NV
7. VASC

Each image is labeled with its corresponding category in a CSV file (GroundTruth.csv). The images vary in size and content.





I. Data Preprocessing

For the Data preparation of this project began with the combining of the images with their corresponding labels. The images were kept in one file while labels for these images were in a CSV form (GroundTruth.csv). The labels for each class were saved in a separate CSV file, where only one entry was destined for each class, and during the preprocessing process, each image name was linked to its particular label to create the chosen dataset. Then we used a smaller subset (1/4th) of the full dataset for computation efficiency.

A. Input Shape

Resizes them to **224x224 pixels**, the required input size for the Xception model. Then stores images and labels in NumPy arrays.

```
[ ]  # Use 1/4 th of the dataset
     num_samples = len(image_names)
     samples = num_samples // 4
```

```
[ ]  # Subset of data
     image_names_subset = image_names[:half_samples]
     labels_subset = labels[:half_samples]
```

```
[ ]  samples
```

```
2503
```

```
[ ]  # Load images
     images = []
     for image_name in image_names_subset:
         image_path = os.path.join(image_dir, image_name + '.jpg')
         img = cv2.imread(image_path)
         if img is not None:
             img = cv2.resize(img, (224, 224))  # Resize to match the input size
             images.append(img)
```

B. Data Splitting

- Training (64%), Validation (16%), and Test (20%) subsets.
- Maintains class balance using stratified sampling.

```
[ ]  # Split dataset into training, validation, and test sets
     train_images, test_images, train_labels, test_labels = train_test_split(
         images, labels_subset, test_size=0.2, stratify=labels_subset, random_state=42
     )
     train_images, val_images, train_labels, val_labels = train_test_split(
         train_images, train_labels, test_size=0.2, stratify=train_labels, random_state=42
     )
```

C. Normalization and Data Augmentation

- We Scaled the pixel values to the range [0, 1] to help the model train efficiently.
- We also applied random transformations like rotation, zoom, flipping, and shifting to create varied training examples, improving generalization.

```
[ ] # Normalize pixel values
    train_images = train_images / 255.0
    val_images = val_images / 255.0
    test_images = test_images / 255.0
```

```
[ ] # Data augmentation
    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
    )
```

II. Model Architecture

Base Model:

- Uses the Xception architecture pre-trained on ImageNet.
- include_top=False: Removes the original classification head to adapt to the task.

Custom Layers:

- Adds a **GlobalAveragePooling2D** layer to reduce spatial dimensions.
- Adds a **Dense layer (128 neurons)** with ReLU activation for feature learning.
- Adds a **Dropout layer (50%)** to prevent overfitting.
- Final **Dense layer** with softmax activation for multi-class classification.

```
# Build XceptionNet model
base_model = Xception(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(128, activation='relu')(x)
x = Dropout(0.5)(x)
predictions = Dense(labels.shape[1], activation='softmax')(x)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/xception/xception_weights_tf_dim_ordering_tf_kernels_notop.h5
83683744/83683744 ━━━━━━━━━━━━━━━━━━━━ 3s 0us/step
```

Model Assembly:

- Combines the base Xception model with custom layers.

Freezing Layers:

- Freezes the base model layers to retain pre-trained features initially.

```
model = Model(inputs=base_model.input, outputs=predictions)
```

```
# Freeze base layers for transfer learning
for layer in base_model.layers:
    layer.trainable = False
```

Compilation

- **Optimizer:** Adam, suitable for adaptive learning rates.
- **Loss Function:** Categorical cross-entropy for multi-class classification.
- **Metrics:** Accuracy.

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

IV. Model Training:

- **Callbacks:**
  - ReduceLROnPlateau: Reduces the learning rate if validation accuracy stagnates.
  - ModelCheckpoint: Saves the best-performing model during training.
- **Training Process:**
  - Runs for 30 epochs with a batch size of 16.

- Uses augmented training data and normalized validation data.

```
callbacks = [
    ReduceLROnPlateau(monitor='val_accuracy', factor=0.5, patience=5, min_lr=1e-4, verbose=1),
    ModelCheckpoint('best_model.keras', monitor='val_accuracy', save_best_only=True, verbose=1)
]
```

```
# Train the model
history = model.fit(
    datagen.flow(train_images, train_labels, batch_size=16),
    validation_data=(val_images, val_labels),
    epochs=30,
    callbacks=callbacks
)
```

Fine-Tuning

- **Unfreezing Layers:**
  - Unfreezes the last 10 layers of the base model for fine-tuning, allowing them to adapt to the dataset.
- **Re-training:**
  - Continues training for 10 more epochs with the same setup.

```
# Unfreeze some layers and fine-tune
for layer in base_model.layers[-10:]:  # Unfreeze last 10 layers for fine-tuning
    layer.trainable = True
```

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

```
history_fine_tune = model.fit(
    datagen.flow(train_images, train_labels, batch_size=16),
    validation_data=(val_images, val_labels),
    epochs=10,
    callbacks=callbacks
)
```

V. Hyperparameters

Batch Size: 16
Epochs (Initial Training): 30
Epochs (Fine-Tuning): 10
Optimizer: Adam
Loss Function: Categorical Crossentropy
Initial Learning Rate: Default (adaptive in Adam)
Learning Rate Reduction Factor: 0.5 (ReduceLROnPlateau)

Minimum Learning Rate: 1e-4
Validation Split: Validation data is 16% of the total dataset.
Data Augmentation: Enabled with rotation, zoom, shearing, and flipping.

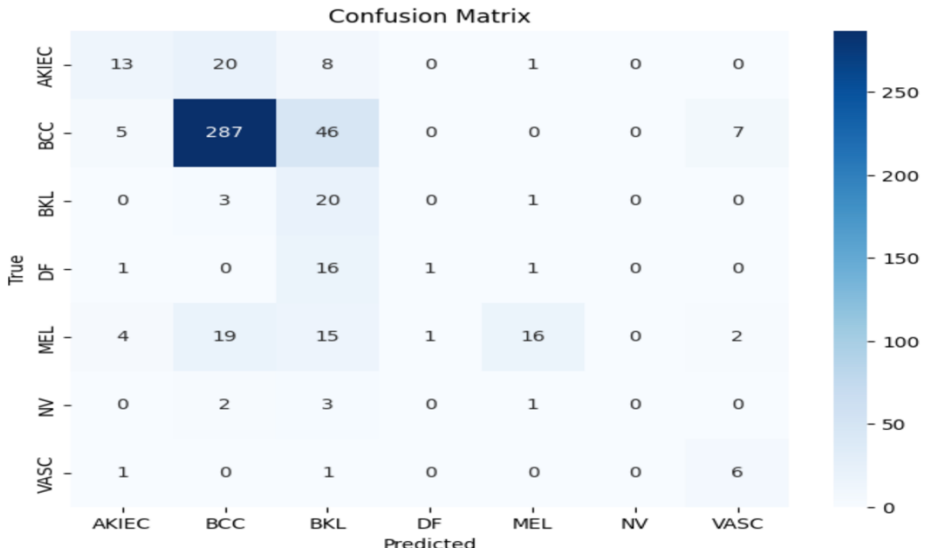
VI. Evaluation Metrics

- **Classification Report:**
  - Displays precision, recall, and F1-score for each class.

```
# Classification report
print("Classification Report:")
print(classification_report(true_labels, predicted_labels, target_names=class_names))
```

```
Classification Report:
              precision    recall  f1-score   support

       AKIEC       0.54      0.31      0.39        42
         BCC       0.87      0.83      0.85       345
         BKL       0.18      0.83      0.30        24
          DF       0.50      0.05      0.10        19
         MEL       0.80      0.28      0.42        57
          NV       0.00      0.00      0.00         6
        VASC       0.40      0.75      0.52         8

    accuracy                           0.68       501
   macro avg       0.47      0.44      0.37       501
weighted avg       0.77      0.68      0.69       501
```

- **Confusion Matrix:**
  - Visualizes true vs. predicted labels using a heatmap.



- **Training History:**
  - Plots accuracy and loss trends for training and validation sets across epochs.

VII . **Libraries Used**

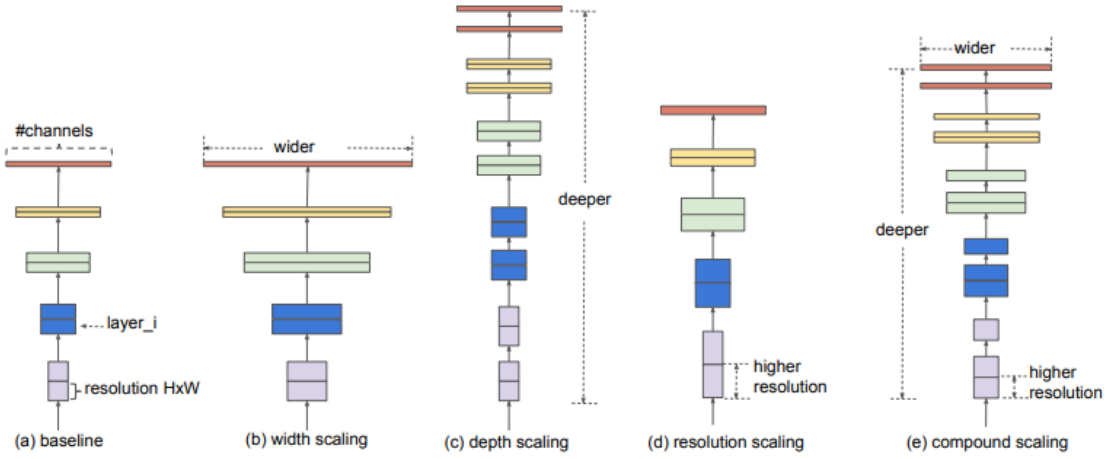
- **NumPy**: For numerical operations and handling image and label arrays.
- **Pandas**: For reading and processing the CSV file with labels.
- **OpenCV (cv2)**: For reading and resizing images.
- **Matplotlib**: For visualizing data distributions and model performance.
- **Seaborn**: For creating enhanced visualizations, like heatmaps for the confusion matrix.
- **Scikit-learn**: For dataset splitting and evaluation metrics (e.g., classification report, confusion matrix).
- **TensorFlow/Keras**:
  - Pre-trained Xception model for transfer learning.
  - Layers, callbacks, and utilities for building and training the neural network.
- **Google Colab Utilities**: For mounting Google Drive to access dataset files.

VIII. Result and Analysis

We got test Accuracy of 68.4 % which is not high performance, however we believe that if we have used a larger data set it would have improved. We got this result because we used only 1/4th part of the data set. We did that because there was no enough GPU to run the whole data set.

II. Second Approach- Training an EfficientNet-B5 CNN Model

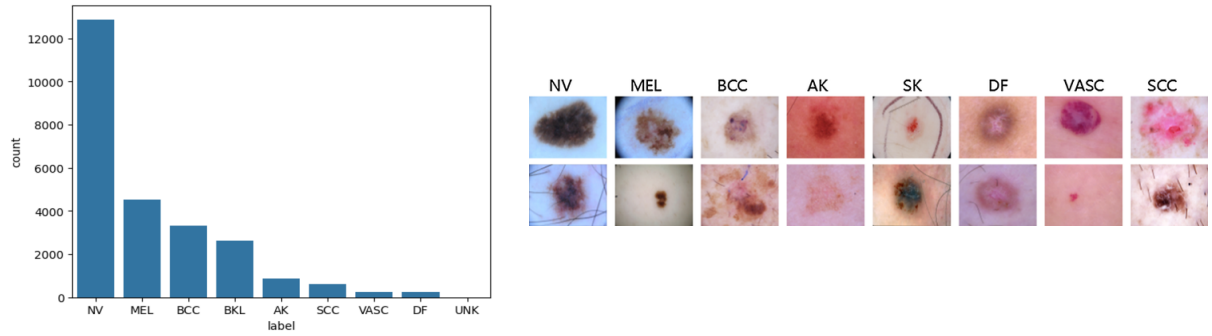
EfficientNet-B5 is a type of neural network and it is designed for image recognition. It's part of the EfficientNet family, which is known for being super smart about balancing size, speed, and accuracy. Instead of just making the network deeper or wider to improve performance, it scales up all aspects—depth, width, and resolution—in a balanced way. Think of it like upgrading a car's engine, tires, and aerodynamics all at once for the best performance. EfficientNet-B5 is a bigger, more powerful version in this family, great for handling complex image tasks while still being efficient with resources like memory and speed. EfficientNet-B5 is named as part of the "B" series in the EfficientNet family, where each "B" level represents a different scale of the model. The numbers (B0, B1, B2, etc.) indicate how much the model has been scaled up in terms of depth, width, and resolution. B5 is a larger version compared to smaller models like B0 or B1, meaning it has more layers (depth), wider layers (more neurons per layer), and higher image resolution inputs. This makes it more powerful and accurate for tasks like image recognition, though it also requires more computational resources.



If we see the above figure (a) is a baseline network example
(b)-(d) are conventional scaling that only increases one dimension of network width, depth, or resolution.
(e) is the architecture we used which is compound scaling method that uniformly scales all three dimensions with a fixed ratio.

III. Dataset

As our primary dataset we used the ISIC-2019 Challenge dataset which consists of 25,331 skill lesion images that are classified in 9 diagnostic categories as we can see in the figure below.



The 9 classification categories of the dataset are:

1. Melanoma
2. Melanocytic nevus
3. Basal cell carcinoma
4. Actinic keratosis
5. Benign keratosis (solar lentigo / seborrheic keratosis / lichen planus-like keratosis)
6. Dermatofibroma
7. Vascular lesion
8. Squamous cell carcinoma
9. None of the others

IV. Data Preprocessing

For the data preparation, the initial step we did was combine the images with their respective labels as the images were downloaded in separate zip file and the labels in a csv file for the dataset (see figure 4). The csv file included the onehot encoded class labels for each row of data and in the preprocessing stage we were able to merge each image path with its label.

```
get_isic_df()
```

	name	label
0	isic_preproc/ISIC_2019_Training_Input/ISIC_000...	NV
1	isic_preproc/ISIC_2019_Training_Input/ISIC_000...	NV
2	isic_preproc/ISIC_2019_Training_Input/ISIC_000...	MEL
3	isic_preproc/ISIC_2019_Training_Input/ISIC_000...	NV
4	isic_preproc/ISIC_2019_Training_Input/ISIC_000...	MEL
...	...	...
25326	isic_preproc/ISIC_2019_Training_Input/ISIC_007...	BCC
25327	isic_preproc/ISIC_2019_Training_Input/ISIC_007...	BKL
25328	isic_preproc/ISIC_2019_Training_Input/ISIC_007...	MEL
25329	isic_preproc/ISIC_2019_Training_Input/ISIC_007...	NV
25330	isic_preproc/ISIC_2019_Training_Input/ISIC_007...	BKL

25331 rows × 2 columns

```
Training samples: 22797
Validation samples: 2534
Testing samples: 8238
                          image  MEL   NV  BCC   AK  BKL   DF  VASC  SCC  UNK  \
12981              ISIC_0053550  0.0  0.0  1.0  0.0  0.0  0.0   0.0  0.0  0.0
18808              ISIC_0062911  0.0  1.0  0.0  0.0  0.0  0.0   0.0  0.0  0.0
12783              ISIC_0034186  0.0  0.0  0.0  0.0  1.0  0.0   0.0  0.0  0.0
1595   ISIC_0012945_downsampled  0.0  1.0  0.0  0.0  0.0  0.0   0.0  0.0  0.0
9125               ISIC_0030528  0.0  0.0  1.0  0.0  0.0  0.0   0.0  0.0  0.0

                                              image_path
12981  /content/ISIC_2019_Training_Input/ISIC_Trainin...
18808  /content/ISIC_2019_Training_Input/ISIC_Trainin...
12783  /content/ISIC_2019_Training_Input/ISIC_Trainin...
1595   /content/ISIC_2019_Training_Input/ISIC_Trainin...
9125   /content/ISIC_2019_Training_Input/ISIC_Trainin...
```

- Then we split the dataset into training, validation, and testing sets.

```
# Load the CSV files
train_df = pd.read_csv(TRAIN_LABELS)
test_df = pd.read_csv(TEST_LABELS)

# Split the training data into training and validation sets
train_df, val_df = train_test_split(train_df, test_size=0.1, random_state=42)

# Confirm the splits
print(f"Training samples: {len(train_df)}")
print(f"Validation samples: {len(val_df)}")
print(f"Testing samples: {len(test_df)}")

# Add full image paths to the DataFrame
train_df['image_path'] = train_df['image'].apply(lambda x: os.path.join(TRAIN_IMAGES, f"{x}.jpg"))
test_df['image_path'] = test_df['image'].apply(lambda x: os.path.join(TEST_IMAGES, f"{x}.jpg"))
val_df['image_path'] = val_df['image'].apply(lambda x: os.path.join(TRAIN_IMAGES, f"{x}.jpg"))
print(train_df.head())
```

A. Input Shape

- EfficientNet-B5 accepts image shapes of 456 by 456. Images are resized to 456x456 which is the input size required by EfficientNet-B5.
- BATCH_SIZE = 32: This was adjusted for training depending on available GPU memory.

B. Normalization and Augmentation

- We applied augmentation techniques like rotations, zoom, flips, lighting changes, and cutouts to the training dataset using the ImageDataGenerator library.
- And for the test and validation sets, they have been normalized without agumentation and rescaled to the same pixel value.

```
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator

IMG_SIZE = 456  # Required input size for EfficientNet-B5
BATCH_SIZE = 32  # Adjust based on available GPU memory

# Data generators
train_datagen = ImageDataGenerator(
    rescale=1.0/255.0,        # Normalize pixel values
    rotation_range=20,        # Random rotation
    width_shift_range=0.2,    # Random horizontal shift
    height_shift_range=0.2,   # Random vertical shift
    zoom_range=0.2,           # Random zoom
    horizontal_flip=True,     # Random horizontal flip
)

val_datagen = ImageDataGenerator(rescale=1.0/255.0)  # Only normalization for testing
test_datagen = ImageDataGenerator(rescale=1.0/255.0)  # Only normalization for validation
```

```
# Define the label columns explicitly
label_columns = ['MEL', 'NV', 'BCC', 'AK', 'BKL', 'DF', 'VASC', 'SCC', 'UNK']

IMG_SIZE = 456  # Reduce image size
BATCH_SIZE = 16  # or 8, or even 4 if necessary

# Update the train_generator
train_generator = train_datagen.flow_from_dataframe(
    dataframe=train_df,
    x_col="image_path",      # Column containing image file paths
    y_col=label_columns,     # Only include the label columns
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=BATCH_SIZE,
    class_mode="raw"         # Use raw labels for one-hot encoding
)

# Update the test_generator (if applicable)
test_generator = test_datagen.flow_from_dataframe(
    dataframe=test_df,
    x_col="image_path",      # Column containing image file paths
    y_col=None,              # No labels for test set
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=BATCH_SIZE,
    class_mode=None          # No labels for inference
)

# Validation data generator
val_generator = val_datagen.flow_from_dataframe(
    dataframe=val_df,
    x_col="image_path",
    y_col=label_columns,  # Label columns
    target_size=(IMG_SIZE, IMG_SIZE),
    batch_size=BATCH_SIZE,
    class_mode="raw"
)
```

C. **Label Columns**: Labels are defined explicitly as ['MEL', 'NV', 'BCC', 'AK', 'BKL', 'DF', 'VASC', 'SCC', 'UNK'].

D. **Data Generators**:

- **Train Generator**: Uses train_datagen for augmentation and one-hot encodes labels (class_mode="raw").
- **Validation Generator**: Similar to the train generator but without augmentation, using val_datagen.
- **Test Generator**: Uses test_datagen for normalization, with no labels (class_mode=None) for inference.

V. Model Architecture

- **Loading Pre-Trained EfficientNet-B5 Base Model:**

When loading the model from tensorflow.keras.applications, the following were also considered.

- **weights='imagenet':** Loads weights pre-trained on the ImageNet dataset, allowing the model to utilize general features learned during pre-training like edges and textures.
- **include_top=False:** Removes the default classification layer of EfficientNet-B5

- **input_shape=(IMG_SIZE, IMG_SIZE, 3):** This specifies the input image dimensions and color channels (3 for RGB images) because EfficientNet-B5 receives image input size 456X456 only.

- **Freeze the Base Model's Layers:**

Then we Froze the Base Model's Layers to prevent updating weights in the pre-trained layers during training, so that we preserve their learned features for now. This was essential to do for transfer learning when training on a smaller dataset so that we avoid overfitting and speed up the training.
So initially these frozen layers act as a fixed feature extractor, and only the custom classification head is trained.

- **Add Custom Classification Head:**

When implementing the model, the important thing we did was to add a custom classification head to the base EfficientNet-B5 model as part of a Sequential model.

Why did we add a Custom Classification Head?

This is because we are using a pretrained EfficientNet-B5 model that was trained on ImageNet and the output layer is built for 1000 classifications (as required for ImageNet which has 1000 classification classes). Hence, since our ISIC dataset has a different number of classes, it needs to be replaced to have an output layer that matches the number of classes in our dataset which is 9. And this will allow the model to adapt to the specific task while leveraging the pre-trained feature extractor.

The custom classification head is appended to the base EfficientNet model during the construction of the Sequential model. This is done before compiling and training the model. The base EfficientNet acts as the backbone (feature extractor), and the custom head tunes these features for our specific dataset.

```python
from tensorflow.keras.applications import EfficientNetB5
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Dropout
from tensorflow.keras.optimizers import Adam

# Load pre-trained EfficientNet-B5 base model
base_model = EfficientNetB5(weights='imagenet', include_top=False, input_shape=(IMG_SIZE, IMG_SIZE, 3))

# Freeze the base model's layers
base_model.trainable = False

# Add custom classification head
model = Sequential([
    base_model,             # Pre-trained EfficientNet-B5 without the classification layer
    GlobalAveragePooling2D(),  # Reduces spatial dimensions while preserving information
    Dropout(0.5),              # Introduces randomness to reduce overfitting
    Dense(128, activation='relu'),  # Fully connected layer to learn higher-level features
    Dropout(0.5),                  # Additional Dropout layer for regularization
    Dense(len(label_columns), activation='softmax')  # Output layer for final predictions
])

# Compile the model
model.compile(
    optimizer=Adam(learning_rate=1e-4),  # Optimizer with a low learning rate for transfer learning
    loss="categorical_crossentropy",    # Suitable for multi-class classification
    metrics=["accuracy"]
)
```

Apart from the base model the custom classification head includes the following:

1. **GlobalAveragePooling2D (GAP)**: Since EfficientNet has already applied MaxPooling or stride convolutions during feature extraction, we dont need further MaxPooling. So instead of using a fully connected layer directly after the feature extractor, GAP reduces each feature map into a single value by averaging all activations from the base model.
2. **Dropout Layers (0.5)**: 2 dropout layers have also been added to reducing overfitting by randomly deactivating 50% of the neurons during training, because we have used a smaller portion of the data for training due to resource issues.
3. **Dense Layer (128, activation='relu')**: We chose a 128 units dense layer because we already have a very large base model with millions of parameters, so we believed making the last/head layer light to control the model size from becoming bigger.
4. **Output layer:** Here the number of units corresponds to the number of target classes (len(label_columns)) and we chose softmax activation because it's suited for multi-class classification tasks.

- **Compiling the Model:**
  - Adam Optimizer with a low learning rate (1e-4) is used which is suitable for fine-tuning pre-trained models.
  - Loss Function: categorical_crossentropy is used for multi-class classification problems.
  - Metrics: Accuracy is tracked to monitor performance during training.
- **Callbacks**:
  - **EarlyStopping** has been added to monitors val_loss and it stops training after 3 epochs without improvement and restores the best weights.
  - **ModelCheckpoint** saves the model with the lowest val_loss as (efficientnetb5_isic.keras).

IV. Hyperparameters

Batch Size: 16
Epochs: 5
Optimizer: Adam
Loss Function: Categorical Crossentropy
Learning Rate: (1e-4)
Validation Split: 10% of the training data used for validation within each epoch.

V. Model Training:

- The model is finally trained using train_generator with validation on val_generator.
- It runs for up to 5 epochs with validation every 2 epochs.
- Early_stopping and model_checkpoint are used for optimization and checkpointing respsctively.

```
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

# Callbacks for training
early_stopping = EarlyStopping(monitor="val_loss", patience=3, restore_best_weights=True)
model_checkpoint = ModelCheckpoint("efficientnetb5_isic.keras", save_best_only=True, monitor="val_loss")


# Train the model
history = model.fit(
    train_generator,
    validation_data=val_generator,
    epochs=5,
    callbacks=[early_stopping, model_checkpoint],
    validation_freq=2  # Validate less frequently
)
```

- **Unfreeze the Base Model's Layers:**

Initially, the base model was frozen to leverage the pre-trained weights without modifying them. After the initial training phase, we unfreeze the base model to allow fine-tuning, and make the pre-trained features adapt to the specifics of out ISIC dataset.

Why?

When the base model is frozen, only the custom classification head's weights are updated. Unfreezing the base model ensures the entire network is fine-tuned. This allows all layers of the base model to be updated during backpropagation.

Hence in this step the fine-tuning involves updating the weights of the pre-trained base model alongside the custom classification head. And this:

1. Refines the general features learned during pre-training to make them more specific to skin lesion classification instead of ImageNet categories.
2. Improves performance metrics like accuracy.
3. Helps the model generalize better.

VI. Evaluation Metrics

The model's performance was evaluated using accuracy, and the maximum accuracy achieved after the only the custom head was trained is 35%. This is mainly because we used only less than half of the data for training due to computational resource restrictions. And for the last model trained which also included the weights of the EfiiecentNet-B5 base model, it was too complex and too resource demanding, which made it difficult for us to even run it using colab GPU runtime. Hopefully as part of our feature work, we can cover that aspect provided that we have the required resources.

```
Epoch 1/5
109/109 ━━━━━━━━━━━━━━━━━━━━ 213s 2s/step - accuracy: 0.3271 - loss: 1.7198
Epoch 2/5
/usr/local/lib/python3.10/dist-packages/keras/src/callbacks/early_stopping.py:155: UserWarning: Early stopping conditioned or
  current = self.get_monitor_value(logs)
/usr/local/lib/python3.10/dist-packages/keras/src/callbacks/model_checkpoint.py:206: UserWarning: Can save best model only wi
  self._save_model(epoch=epoch, batch=None, logs=logs)
109/109 ━━━━━━━━━━━━━━━━━━━━ 0s 2s/step - accuracy: 0.3145 - loss: 1.7116/usr/local/lib/python3.10/dist-packages/keras/src/tr
  self._warn_if_super_not_called()
109/109 ━━━━━━━━━━━━━━━━━━━━ 304s 2s/step - accuracy: 0.3145 - loss: 1.7116 - val_accuracy: 0.3177 - val_loss: 1.6836
Epoch 3/5
109/109 ━━━━━━━━━━━━━━━━━━━━ 220s 2s/step - accuracy: 0.2929 - loss: 1.7246
Epoch 4/5
109/109 ━━━━━━━━━━━━━━━━━━━━ 274s 2s/step - accuracy: 0.3236 - loss: 1.7021 - val_accuracy: 0.3177 - val_loss: 1.6837
Epoch 5/5
109/109 ━━━━━━━━━━━━━━━━━━━━ 214s 2s/step - accuracy: 0.3257 - loss: 1.7044
```

III. Third Approach- Dual-Path CNN Model

The **Dual-Path CNN Model** is a convolutional neural network designed to leverage two different types of input data which are the original images and their segmentation masks. This approach is important for tasks like skin lesion classification**,** where the image's structure and specific regions of interest play important role in detection. The Advantages of Dual-Path CNN is to enhance feature extraction which combines global context of the images and localized information of masks and for improved accuracy that applies segmentation data to focus on critical regions, reducing noise.

I. Data Overview

In our project, 10,000-data-size train and test datasets from the ISIC SBI2016 ISIC are used. The dataset consists of

- Pairs of high-resolution skin images
- the matching segmentation masks.

The Labels are Binary classification of

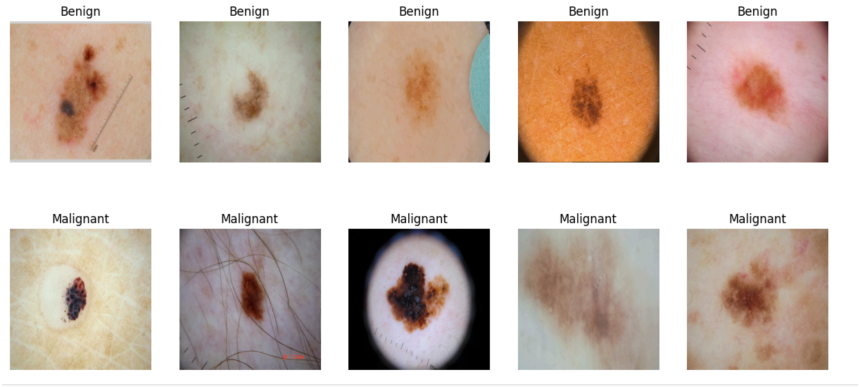
- 0 as Benign (non-cancerous)
- 1 as Malignant (cancerous).

The training dataset with the original and segmentation masks are loaded as follows.

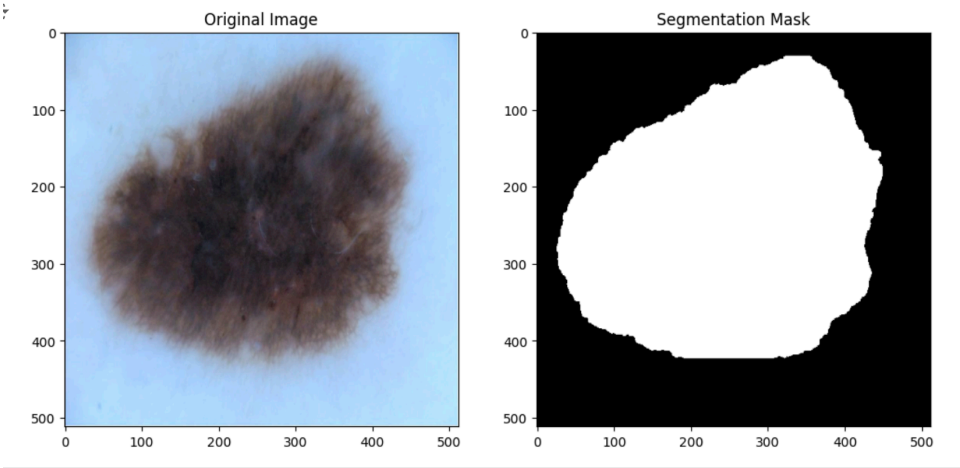
```python
# Define paths to dataset and CSV
DATASET_PATH = '/content/ISBI2016_ISIC_Part3B_Training_Data'
CSV_PATH = '/content/ISBI2016_ISIC_Part3B_Training_GroundTruth.csv'

# Function to load images and masks
def load_images_and_masks(data_path, labels_df, img_size=(512, 512), max_images=60):
    images = []
    masks = []
    labels = []
    if max_images is not None:
        labels_df = labels_df.iloc[:max_images]
    for _, row in labels_df.iterrows():
        img_path = os.path.join(data_path, f"{row['image_id']}.jpg")
        mask_path = os.path.join(data_path, f"{row['image_id']}_Segmentation.png")
        label = row['label']
        if os.path.exists(img_path) and os.path.exists(mask_path):
            img = load_img(img_path, target_size=img_size)
            img = img_to_array(img) / 255.0
            mask = load_img(mask_path, color_mode='grayscale', target_size=img_size)
            mask = img_to_array(mask) / 255.0
            images.append(img)
            masks.append(mask)
            labels.append(label)
        else:
            print(f"Warning: Image or mask not found for {row['image_id']}")
    return np.array(images), np.array(masks), np.array(labels)

# Load the CSV file and encode labels
labels_df = pd.read_csv(CSV_PATH, header=None)
labels_df.columns = ['image_id', 'label']
label_encoder = LabelEncoder()
labels_df['label'] = label_encoder.fit_transform(labels_df['label'])

# Load images and masks
X_img, X_mask, y = load_images_and_masks(DATASET_PATH, labels_df, max_images=5000)
```

To visualize the 5 random benign and 5 random malignant skin lesion images from the dataset, displaying them in a 2x5 grid with appropriate labels, we used the code below.

```python
# Visualize a few images from each class
fig, ax = plt.subplots(2, 5, figsize=(15, 7))
ax = ax.ravel()

benign_images = labels_df[labels_df['label'] == 0].sample(5)  # Random benign samples
malignant_images = labels_df[labels_df['label'] == 1].sample(5)  # Random malignant samples

# Load and display sample images
for i, row in enumerate(benign_images.iterrows()):
    img_path = os.path.join(DATASET_PATH, f"{row[1]['image_id']}.jpg")
    img = load_img(img_path, target_size=(512, 512))
    img = img_to_array(img) / 255.0
    ax[i].imshow(img)
    ax[i].set_title('Benign')
    ax[i].axis('off')

for i, row in enumerate(malignant_images.iterrows(), 5):
    img_path = os.path.join(DATASET_PATH, f"{row[1]['image_id']}.jpg")
    img = load_img(img_path, target_size=(512, 512))
    img = img_to_array(img) / 255.0
    ax[i].imshow(img)
    ax[i].set_title('Malignant')
    ax[i].axis('off')

plt.show()
```



The following also shows the first skin lesion image from the dataset alongside its corresponding segmentation mask, displaying them side-by-side to highlight the lesion area.



II. Preprocessing

Before feeding data into the model

- **Images** are loaded and resized to 512x512 pixels for consistent input size.
- **Segmentation Masks** highlight the lesion areas and are also resized and normalized to match the image size.
- Both inputs (images and masks) are normalized by dividing pixel values by 255 to scale them between 0 and 1, improving training stability.

III. Model Architecture of the Dual-Path CNN

The model has two different input paths:

1. **Image Input Path:** Processes the raw images.
2. **Segmentation Mask Input Path:** Focuses on the lesion areas

a. Image Input Path

- Input shape: (512, 512, 3) for RGB images.
- **Layers:**
  - Conv2D: Extracts spatial features such as edges and textures.
  - MaxPooling2D: Reduces the size of feature maps.

b. Mask Input Path

- Input shape: (512, 512, 1) for grayscale segmentation masks.
- **Layers:**
  - Process single-channel (grayscale) masks.

c. Merging the Two Paths

- **Concatenation Layer:** Combines the features extracted from both input paths.
- Additional Conv2D and MaxPooling2D layers further process the combined features.
- **Dense Layers:**
  - Flatten: Converts the feature maps into a 1D vector.
  - Fully connected (Dense) layers with dropout regularization prevent overfitting.
- **Output Layer:**
  - A single neuron with a sigmoid activation function outputs the probability of the lesion being malignant.

```python
def create_dual_path_cnn(input_shape):
    input_orig = Input(shape=input_shape, name='original_input')
    x1 = Conv2D(32, (3, 3), activation='relu', padding='same')(input_orig)
    x1 = MaxPooling2D((2, 2))(x1)
    x1 = Conv2D(64, (3, 3), activation='relu', padding='same')(x1)
    x1 = MaxPooling2D((2, 2))(x1)
    input_shape_mask = (input_shape[0], input_shape[1], 1)
    input_seg = Input(shape=input_shape_mask, name='segmentation_input')
    x2 = Conv2D(32, (3, 3), activation='relu', padding='same')(input_seg)
    x2 = MaxPooling2D((2, 2))(x2)
    x2 = Conv2D(64, (3, 3), activation='relu', padding='same')(x2)
    x2 = MaxPooling2D((2, 2))(x2)
    combined = concatenate([x1, x2])
    x = Conv2D(128, (3, 3), activation='relu', padding='same')(combined)
    x = MaxPooling2D((2, 2))(x)
    x = Flatten()(x)
    x = Dense(128, activation='relu')(x)
    x = Dropout(0.5)(x)
    output = Dense(1, activation='sigmoid')(x)
    model = Model(inputs=[input_orig, input_seg], outputs=output)
    return model
```

IV. Training

a. Data Splitting

- The dataset is split into training and validation sets using **K-Fold Cross-Validation** (3 folds), which ensures robust evaluation and reduces overfitting.

b. Hyperparameters

- **Optimizer:** Adam (adaptive learning rate for efficient training).
- **Loss Function:** Binary Crossentropy (suitable for binary classification).
- **Metrics:** Accuracy to evaluate performance.

c. Callbacks

- **EarlyStopping:** Stops training if validation loss doesn't improve for 10 epochs.
- **ReduceLROnPlateau:** Reduces the learning rate if validation loss plateaus.

```python
# Prepare for k-folds cross-validation
n_splits = 3
kf = KFold(n_splits=n_splits, shuffle=True, random_state=42)
train_acc = []
val_acc = []

# Training loop for k-folds cross-validation
for train_index, val_index in kf.split(X_img):
    X_img_train, X_img_val = X_img[train_index], X_img[val_index]
    X_mask_train, X_mask_val = X_mask[train_index], X_mask[val_index]
    y_train, y_val = y[train_index], y[val_index]

    model = create_dual_path_cnn((512, 512, 3))
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

    early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
    reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.001)

    history = model.fit(
        [X_img_train, X_mask_train], y_train,
        validation_data=([X_img_val, X_mask_val], y_val),
        epochs=15,
        batch_size=3,
        verbose=1,
        callbacks=[early_stopping, reduce_lr]
    )
    train_acc.append(history.history['accuracy'])
    val_acc.append(history.history['val_accuracy'])
```

Training reaches high accuracy (~100%) for most folds, but validation accuracy stabilizes around **70-80%**.

```
Epoch 8/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 60ms/step - accuracy: 0.9767 - loss: 0.1479 - val_accuracy: 0.7500 - val_loss: 0.7201 - learning_rate: 0.0010
Epoch 9/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 59ms/step - accuracy: 1.0000 - loss: 0.0309 - val_accuracy: 0.8000 - val_loss: 1.7073 - learning_rate: 0.0010
Epoch 10/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 63ms/step - accuracy: 1.0000 - loss: 0.0039 - val_accuracy: 0.8000 - val_loss: 2.5417 - learning_rate: 0.0010
Epoch 11/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 60ms/step - accuracy: 0.9482 - loss: 0.0771 - val_accuracy: 0.8500 - val_loss: 1.3836 - learning_rate: 0.0010
Epoch 12/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 60ms/step - accuracy: 1.0000 - loss: 0.0015 - val_accuracy: 0.8000 - val_loss: 1.6392 - learning_rate: 0.0010
Epoch 13/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 64ms/step - accuracy: 1.0000 - loss: 7.8338e-04 - val_accuracy: 0.8000 - val_loss: 2.1663 - learning_rate: 0.0010
Epoch 14/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 63ms/step - accuracy: 1.0000 - loss: 2.6932e-04 - val_accuracy: 0.8000 - val_loss: 3.0027 - learning_rate: 0.0010
Epoch 1/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 7s 273ms/step - accuracy: 0.6789 - loss: 3.9115 - val_accuracy: 0.8000 - val_loss: 0.5589 - learning_rate: 0.0010
Epoch 2/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 6s 90ms/step - accuracy: 0.8624 - loss: 0.5917 - val_accuracy: 0.8000 - val_loss: 0.5142 - learning_rate: 0.0010
Epoch 3/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 61ms/step - accuracy: 0.7512 - loss: 0.5441 - val_accuracy: 0.8000 - val_loss: 0.8851 - learning_rate: 0.0010
Epoch 4/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 98ms/step - accuracy: 0.8427 - loss: 0.6017 - val_accuracy: 0.8000 - val_loss: 0.5096 - learning_rate: 0.0010
Epoch 5/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 67ms/step - accuracy: 0.9065 - loss: 0.2972 - val_accuracy: 0.8000 - val_loss: 2.4549 - learning_rate: 0.0010
Epoch 6/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 70ms/step - accuracy: 0.8597 - loss: 0.5013 - val_accuracy: 0.5500 - val_loss: 0.6726 - learning_rate: 0.0010
Epoch 7/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 59ms/step - accuracy: 0.9864 - loss: 0.1702 - val_accuracy: 0.7500 - val_loss: 0.7610 - learning_rate: 0.0010
Epoch 8/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 59ms/step - accuracy: 1.0000 - loss: 0.0259 - val_accuracy: 0.7000 - val_loss: 1.3223 - learning_rate: 0.0010
Epoch 9/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 64ms/step - accuracy: 1.0000 - loss: 0.0036 - val_accuracy: 0.7000 - val_loss: 1.4502 - learning_rate: 0.0010
Epoch 10/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 64ms/step - accuracy: 0.9667 - loss: 0.0233 - val_accuracy: 0.7000 - val_loss: 1.5072 - learning_rate: 0.0010
Epoch 11/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 63ms/step - accuracy: 1.0000 - loss: 0.0016 - val_accuracy: 0.7000 - val_loss: 1.8293 - learning_rate: 0.0010
Epoch 12/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 60ms/step - accuracy: 1.0000 - loss: 1.1208e-04 - val_accuracy: 0.7500 - val_loss: 2.3349 - learning_rate: 0.0010
Epoch 13/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 61ms/step - accuracy: 1.0000 - loss: 3.8176e-05 - val_accuracy: 0.7000 - val_loss: 2.4334 - learning_rate: 0.0010
Epoch 14/15
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 64ms/step - accuracy: 1.0000 - loss: 7.9310e-05 - val_accuracy: 0.7000 - val_loss: 2.3365 - learning_rate: 0.0010
```

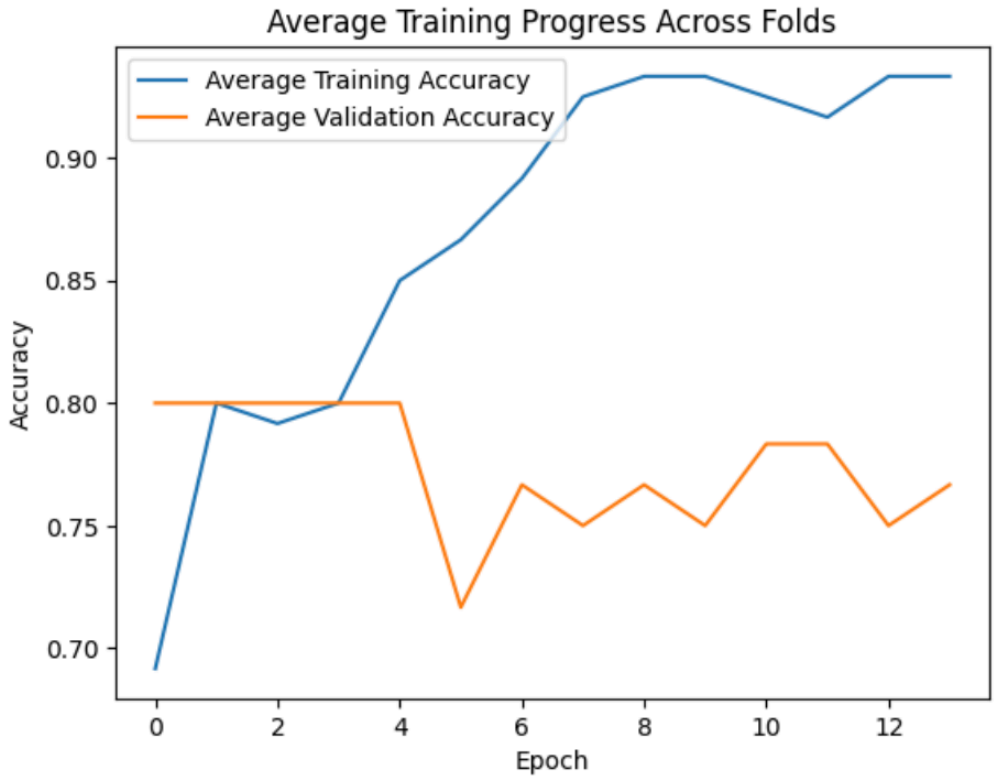
V. Evaluation

Train and Validation Set

The code first truncates training and validation accuracies across all k-folds to the shortest fold length, calculates their averages per epoch, and displays them in a tabular format.

The average training and validation accuracies, showing steady training improvement and stable validation accuracy around 80%. And the training accuracy reaches ~93%.

```
Epoch   Average Training Accuracy       Average Validation Accuracy
1       0.6917                  0.8000
2       0.8000                  0.8000
3       0.7917                  0.8000
4       0.8000                  0.8000
5       0.8500                  0.8000
6       0.8667                  0.7167
7       0.8917                  0.7667
8       0.9250                  0.7500
9       0.9333                  0.7667
10      0.9333                  0.7500
11      0.9250                  0.7833
12      0.9167                  0.7833
13      0.9333                  0.7500
14      0.9333                  0.7667
```



Test Set

The Accuracy is 80% on the test set. Therefore, combining images and segmentation masks improves classification performance by focusing on relevant areas. The model generalizes well but could benefit from additional tuning and larger datasets evaluates the trained model on the test dataset.

```
Classification Report:
              precision    recall  f1-score   support

         0.0       0.80      1.00      0.89       304
         1.0       0.00      0.00      0.00        75

    accuracy                           0.80       379
   macro avg       0.40      0.50      0.45       379
weighted avg       0.64      0.80      0.71       379

Accuracy: 0.8021108179419525
```

VII. Libraries Used

- **os**: For file and directory operations, such as navigating the dataset and checking file paths.
- **zipfile**: For extracting compressed dataset files
- **NumPy**: For numerical operations and handling arrays of images and labels efficiently during preprocessing and model input preparation.
- **Pandas**: For loading and processing the CSV file containing image labels and metadata, enabling data manipulation and organization.
- **Matplotlib.pyplot**: For visualizing images, segmentation masks, and plotting training/validation accuracy or loss curves.
- **TensorFlow**: For building, training, and evaluating the deep learning model, including image preprocessing and designing the CNN architecture.
- **Scikit-Learn (sklearn)**: For data splitting (train-test and k-fold), label encoding, and model evaluation metrics like classification reports and accuracy scores.

Conclusion

In conclusion, our project successfully demonstrated the potential of using diversified CNN architectures for the task of skin cancer detection. The model that has performed the best from all the models is the Dual Path CNN by achieving 80% accuracy. For the other models, despite the challenges we faced due to computational limitations we were able to underatand and achieve some level of accuracy.

As part of our future work, provided we get access to the necessary computing resources, we plan to extend our experiment and build and train the rest of the models which were mentioned in the three scholarly papers we reviewed. By doing so, we aim to not only replicate but also refine these models to achieve better accuracies. This will involve experimenting with more extensive training data, advanced preprocessing techniques, and possibly integrating newer, more efficient neural network architectures.

Codes

1. XceptionNet

https://colab.research.google.com/drive/17msc3wBjO5_PYLtURQqSMflS-3-Et017?usp=sharing

2. EfficientNet-B5

https://colab.research.google.com/drive/172sUZYIPi0DRWGT7EBjq2GT2calPV5jt?usp=sharing

3. Dual Path CNN model

https://colab.research.google.com/drive/1FNvGyt2rqI_Lht_QpJ32XdqoPecjkkUI?usp=sharing

References

1. D. S. Charan, H. Nadipineni, S. Sahayam, and U. Jayaraman, "Method to classify skin lesions using dermoscopic images," Department of Computer Science, Indian Institute of Information Technology, Design and Manufacturing, Chennai, Tamil Nadu, India, 2022.

2. F. Mahmood, D. Desai, and J. Zhou, "An interpretable deep learning approach for skin cancer categorization," IEEE Transactions on Medical Imaging, vol. 42, pp. 1123–1135, 2023.

3. J. Kawahara, S. Daneshvar, G. Argenziano, and G. Hamarneh, "Seven-point checklist and skin lesion classification using multitask multimodal neural nets," IEEE Transactions on Biomedical Engineering, vol. 69, pp. 348–358, 2018.