Skin Vision

Reference : <https://www.kaggle.com/code/malakalaabiad/skin-tone-classification#4.-Model-Creation-Using-Transfer-Learning-(MobileNetV2)>

Data Augmentation and Preprocessing

Crucial for training a more robust and generalized model.

Why augment data?

Prevents overfitting by making the model see slightly modified versions of images.

Helps the model generalize better by learning invariant features (like skin tones from different angles or lighting).

train\_datagen = ImageDataGenerator(

rescale=1./255,

rotation\_range=20,

zoom\_range=0.2,

width\_shift\_range=0.2,

height\_shift\_range=0.2,

horizontal\_flip=True

)

rescale=1./255 - Normalizes pixel values from 0–255 to 0–1 (MobileNetV2 expects normalized input).

rotation\_range=20 - Randomly rotates images by up to 20 degrees.

zoom\_range=0.2 - Randomly zooms in or out by up to 20%.

width\_shift\_range=0.2 - Horizontally shifts images by up to 20%.

height\_shift\_range=0.2 - Vertically shifts images by up to 20%.

horizontal\_flip=True - Randomly flips images horizontally.

Validation Data

valid\_datagen = ImageDataGenerator(rescale=1./255)

Only rescaling is applied — no augmentation.

Why?

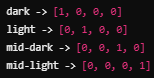
Validation data should reflect real-world, unaltered images.

It tests how well the model performs on untouched data.

Data loaded as batches. Batch size = 32 images

Class Mode — class\_mode='categorical' because:

Your labels are one-hot encoded — like this,



Model creation using MobileNetV2

base\_model = MobileNetV2(weights='imagenet', include\_top=False, input\_shape=(224, 224, 3))

MobileNetV2 — loads a pre-trained MobileNetV2 model.

weights='imagenet' — uses weights trained on the ImageNet dataset (1,000 classes like dogs, cats, etc.).

include\_top=False — removes the final classification layer (the one for 1,000 ImageNet classes).

input\_shape=(224, 224, 3) — sets the input size to 224x224 RGB images (3 channels for color).

Why remove the top layer?

ImageNet’s final layer outputs 1,000 classes (like dogs, cats, etc.).

Your task only has 4 classes (skin tones), so you remove the top layer and add your own custom layers.

Freeze base layers

for layer in base\_model.layers:

layer.trainable = False

freezing layers means their weights won’t be updated during training.

The convolutional layers of MobileNetV2 already learned to detect basic features (edges, textures, shapes) during ImageNet training.

This lets you reuse those features for skin tone classification without re-learning from scratch.

Why freeze layers?

Transfer learning works best when you reuse low-level features from pre-trained models.

Early layers detect edges, shapes, and patterns — useful for any image task (like skin textures).

You only train the new layers (which learn skin tone-specific patterns).

If you want to **fine-tune** the base model, you can unfreeze some layers:

for layer in base\_model.layers[-10:]:

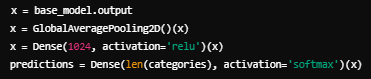
layer.trainable = True

This unfreezes the last 10 layers of MobileNetV2 - so the model can fine-tune deeper patterns.

Why fine-tune?

If your skin tone dataset is complex, you may want the model to adjust some pre-trained filters.

New layers add by me.



The output layer will give something like this for a given image:

A screen shot of a computer

AI-generated content may be incorrect.

The class with the highest probability is the predicted class.

Then we want to define the **full model** with connecting them.

[Input (224, 224, 3)] → [MobileNetV2 layers (frozen)] → [GlobalAveragePooling2D] → [Dense 1024 ReLU] → [Dense 4 Softmax]

model.compile(optimizer=Adam(learning\_rate=0.001), loss='categorical\_crossentropy', metrics=['accuracy'])

Adam optimizer — updates the model’s weights during training using the Adam algorithm (adaptive learning rate optimization).

Learning rate (lr=0.001) — controls how fast/slow the model updates weights.

categorical\_crossentropy — loss function for multi-class classification with one-hot labels (like [1, 0, 0, 0] for "dark").

metrics=['accuracy'] tracks accuracy during training.

We can add other matrics.

import torch.optim as optim

from torchmetrics.classification import Accuracy, Precision, Recall, AUROC, F1Score

model.compile(optimizer=Adam(learning\_rate=0.001), loss='categorical\_crossentropy', metrics=['accuracy', Precision(name='precision'), Recall(name='recall'), AUC(name='auc'), F1Score(name='f1\_score')])

Early Stopping’s Role

early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

Monitors val\_loss - if it stops improving for 3 epochs:

Training halts to prevent overfitting (model learning noise instead of patterns).

restore\_best\_weights= True — rolls back the model to the epoch with the lowest val\_loss.

This ensures you don’t over-train your model.

Train the model

When training pretrained model and our model are connected.

history = model.fit(

train\_generator, # Your dataset (images + labels)

epochs=10,

validation\_data=validation\_generator, # For val\_loss monitoring

callbacks=[early\_stopping]

)

Steps in training,

1. train\_generator — batches images and labels from your dataset (flow\_from\_directory points to your folders like this):
2. Images → MobileNetV2:

Each 224x224 image passes through MobileNetV2’s pre-trained filters.

The pre-trained layers detect edges, textures, and basic patterns.

1. MobileNetV2 → Custom Layers:

The filtered outputs go through your GlobalAveragePooling2D → Dense (1024) → Dense (4) layers.

These layers learn the skin tone-specific relationships between the extracted features and your 4 classes.

1. Predictions vs. Labels:

The final layer produces 4 probabilities.

Cross-entropy loss compares these with the true labels (from your dataset).

1. Weight Update:

Only the custom layer weights are updated (because the base model’s layers are frozen).

1. Validation:

At the end of each epoch, the model evaluates on the validation set using validation\_generator.

The early stopping callback watches val\_loss — if it doesn’t improve for 3 epochs, training stops early.

Dataset (train\_generator) → MobileNetV2 → Custom Layers → Loss/Accuracy → Weight Update

Improve the model performance

1. Increase the size of the dataset.
2. Prevent overfitting we used early stopping criteria

We can get the best model by putting a checkpoint on the best model and save that.

from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping

early\_stopping = EarlyStopping(monitor='val\_loss', patience=5, min\_delta=0.001, restore\_best\_weights=True)

checkpoint = ModelCheckpoint(

'best\_model.h5',

monitor='val\_loss',

save\_best\_only=True,

mode='min'

)

model.fit(

train\_generator,

epochs=20,

validation\_data=validation\_generator,

callbacks=[early\_stopping, checkpoint]

)

Ensures you stop training at the right time and keep the best model.

1. Data Augmentation

Add more variety to your training data using ImageDataGenerator:

train\_datagen = ImageDataGenerator(

rescale=1./255,

rotation\_range=30, # More rotation range

zoom\_range=0.3, # Larger zoom range

width\_shift\_range=0.3,

height\_shift\_range=0.3,

horizontal\_flip=True,

brightness\_range=[0.7, 1.3], # Adjust brightness

shear\_range=0.3, # Shearing transformations

fill\_mode='nearest'

)

Helps your model generalize — learns to recognize skin tones from different angles, lighting, and zoom levels.

1. Unfreeze Layers for Fine-Tuning

for layer in base\_model.layers[-10:]:  
 layer.trainable = True

Unfreeze last 10 layers to fine tune the pretrained model.

Why:

Lower layers detect edges/textures (good as-is).

Higher layers learn complex patterns (fine-tuning lets them adjust to skin tones).

Lower learning rate (like 1e-5) avoids "forgetting" pre-trained knowledge.

1. Add Dropout to Prevent Overfitting

x = base\_model.output

x = GlobalAveragePooling2D()(x)

x = Dense(1024, activation='relu')(x)

x = Dropout(0.5)(x) # Drop 50% of neurons

predictions = Dense(4, activation='softmax')(x)

Dropout randomly "turns off" some neurons during training — making the model more robust.

Helps regularize the model — preventing it from memorizing training data.

1. Learning Rate Scheduling

Use a learning rate scheduler — so the model starts learning fast but slows down to fine-tune:

from tensorflow.keras.callbacks import ReduceLROnPlateau

lr\_scheduler = ReduceLROnPlateau(

monitor='val\_loss',

factor=0.5,

patience=3,

min\_lr=1e-6

)

model.fit(

train\_generator,

epochs=20,

validation\_data=validation\_generator,

callbacks=[lr\_scheduler]

)

Gradually reduces learning rate when the model stops improving.

Metrics

* Accuracy
* Precision
* AUC –

AUC measures the ability of a model to separate positive and negative classes across different thresholds. It is the area under the ROC (Receiver Operating Characteristic) curve.

AUC = 1.0 → Perfect model (separates classes perfectly).

AUC = 0.5 → Random guessing (no discrimination).

AUC < 0.5 → Worse than random (flipping predictions may be better).

Usages;

General Classifier Performance → Measures the model’s ability to distinguish between classes.

Medical Tests → Determines the effectiveness of diagnostic tests.

* Recall –

measures how many actual positive samples are correctly predicted as positive. It is important when missing positive cases is costly.

* F1 score –

F1 Score balances precision and recall in cases of class imbalance.

How to check a model

What is validation accuracy?

Training accuracy measures how well the model predicts on the training data (the data it’s learning from).

Validation accuracy measures how well the model predicts on unseen data (the validation set, which the model never trains on).

Formula for accuracy:

High training accuracy but low validation accuracy means overfitting — the model memorizes the training data but can’t handle new data.

Validation accuracy shows how well the model has learned general patterns — not just memorized answers.

Ideal case: Training accuracy ≈ Validation accuracy — good generalization.

Overfitting: Training accuracy high, Validation accuracy low — memorizing, not learning.

Underfitting: Both training and validation accuracies are low — the model is too simple.