**skin\_training.ipynb**

­

**1. Importing Dependencies:**

Import necessary Python libraries and modules, including TensorFlow, Keras, Matplotlib, and scikit-learn metrics for later evaluation.

**2. Setting Constants:**

Constants like BATCH\_SIZE, IMAGE\_SIZE, CHANNELS, and EPOCHS are defined. These constants determine various parameters for data preprocessing and model training.

**3. Importing and Preprocessing Data:**

The code uses tf.keras.preprocessing.image\_dataset\_from\_directory to load image data from a specified directory.

It sets the seed for reproducibility, shuffles the dataset, and specifies the image size and batch size.

class\_names is a list of class names extracted from the dataset.

**4. Data Visualization:**

This section visualizes a batch of images from the dataset using Matplotlib.

It displays 12 images with their corresponding class labels.

**5. Splitting the Dataset:**

The code calculates the size of the dataset and splits it into training, validation, and test sets using a custom function get\_dataset\_partitions\_tf.

The dataset is divided into portions based on the specified split ratios (train\_split, val\_split, test\_split).

**6. Data Caching and Prefetching:**

The training, validation, and test datasets are cached, shuffled, and prefetched to optimize data loading during model training.

**7. Model Building:**

The neural network model for skin cancer detection is defined using Keras Sequential API.

The model includes various layers, such as resizing, rescaling, convolution, max-pooling, and dense layers.

The final layer uses softmax activation for multiclass classification.

**8. Model Compilation:**

The model is compiled with an optimizer (Adam), a loss function (SparseCategoricalCrossentropy), and metrics (accuracy) for training.

**9. Model Training:**

The model is trained using the fit method.

Training data (train\_ds) and validation data (val\_ds) are provided.

Training progresses over a specified number of epochs (100 in this case).

The history variable stores training history, including loss and accuracy for each epoch.

**10. Plotting Training Metrics:**

The code plots training and validation accuracy and loss curves using Matplotlib.

These curves provide insights into the model's learning progress and potential issues like overfitting.

**11. Model Evaluation:**

The trained model is evaluated on the test dataset using the evaluate method.

Test accuracy and loss are obtained and printed.

**12. Accuracy Visualization:**

The code plots the accuracy scores for training, validation, and test datasets.

It provides a visual representation of the model's performance.

**13. Precision, Recall, and F1-Score:**

The code calculates precision, recall, and F1-score for each class in the test dataset using scikit-learn metrics.

These metrics are important for evaluating classification models.

**14. ROC Curve and AUC Score:**

Calculates the ROC curve and AUC score for binary classification (assuming a binary problem).

The ROC curve visually represents the model's ability to distinguish between classes, and AUC quantifies its performance.

**15. Model Saving:**

The trained model is saved in both Keras HDF5 format (.h5) and TensorFlow SavedModel format (.pb) for model prediction or future use.

**Skin\_ResNet.ipynb**

**1. Importing Dependencies and Setting Constants:**

The code starts by importing required libraries, such as TensorFlow, Matplotlib, and scikit-learn.

It also sets several constants, including batch size, image size, channels, and the number of epochs for training.

**2.** **Importing the Dataset:**

It uses tf.keras.preprocessing.image\_dataset\_from\_directory to load the skin cancer dataset.

The dataset is loaded from the "skin" directory, with specified image dimensions and batch size.

Class names are extracted from the dataset.

**3.** **Splitting the Dataset:**

A function get\_dataset\_partitions\_tf is defined to split the dataset into training, validation, and test sets.

The data is shuffled before splitting.

**4. Caching, Shuffling, and Prefetching:**

Training, validation, and test datasets are cached, shuffled, and prefetched to optimize data loading during training.

**5. Building the Model:**

A ResNet50 model is loaded from tensorflow.keras.applications.

The code builds a custom model by adding layers on top of the pre-trained ResNet50 model.

It includes resizing and rescaling layers for input preprocessing.

A Global Average Pooling layer is added to reduce spatial dimensions.

A Dense layer with softmax activation is added for classification into two classes (benign and malignant).

**6. Compiling the Model:**

The model is compiled with the Adam optimizer, sparse categorical cross-entropy loss, and accuracy as the metric.

**7. Model Training:**

The model is trained using the fit method.

Training data (train\_ds) and validation data (val\_ds) are provided.

Training progresses over a specified number of epochs.

Training history is stored in the history variable.

**8. Saving the Model:**

The trained model is saved in HDF5 format (".h5").

**9.Plotting Training Metrics:**

Training and validation accuracy and loss curves are plotted using Matplotlib based on the training history.

**10. Run Prediction on Sample Images:**

A function predict is defined to make predictions on sample images.

Predictions are generated for a batch of sample images from the test dataset.

Actual and predicted class labels, along with confidence scores, are displayed.

**11. Model Evaluation:**

The model is evaluated on the test dataset to compute test loss and accuracy.

The results are printed.

**12. Accuracy Visualization:**

Accuracy scores are visualized using bar plots.

**13. Precision, Recall, and F1-Score:**

Precision, recall, and F1-score are computed for each class on the test dataset using scikit-learn metrics.

Bar plots are created to visualize these metrics.

**14. ROC Curve and AUC Score:**

The ROC curve and AUC score are calculated for binary classification (assuming a binary problem).

The ROC curve is plotted, and the AUC score is printed.

**Skin\_cancer\_AlexNet.ipynb**

**Importing Dependencies:**

TensorFlow and other necessary libraries like Matplotlib, NumPy, and scikit-learn are imported.

**Setting Constants:**

Constants such as batch size, image size, number of color channels, and the number of training epochs are defined.

**Importing Data:**

The code uses tf.keras.preprocessing.image\_dataset\_from\_directory to import the skin cancer dataset from a specified directory.

Class names are extracted from the dataset.

**Visualizing Images:**

A few images from the dataset are visualized for a quick overview.

**Splitting the Dataset:**

The dataset is split into training, validation, and test sets.

The split ratios are defined, and the dataset is divided accordingly.

**Custom Dataset Splitting Function:**

A function named get\_dataset\_partitions\_tf is defined to split the dataset into training, validation, and test sets.

The function also supports shuffling the dataset.

**Caching, Shuffling, and Prefetching:**

The code caches, shuffles, and prefetches the training, validation, and test datasets to optimize data loading during training.

**Building the Model:**

An AlexNet-like custom model is built using TensorFlow's Keras API.

It includes convolutional layers, max-pooling layers, and fully connected layers.

The output layer uses softmax activation for classification into two classes.

**Compiling the Model:**

The model is compiled with the Adam optimizer and sparse categorical cross-entropy loss.

Accuracy is used as a metric.

**Training the Model:**

The model is trained on the training dataset using the fit method.

Validation data is also provided for monitoring model performance during training.

**Evaluating the Model:**

The model is evaluated on the test dataset to calculate the test loss and accuracy.

**Plotting Training Metrics:**

Training and validation accuracy and loss curves are plotted using Matplotlib based on the training history.

**Printing Precision, Recall, and F1 Score:**

Precision, recall, and F1-score are computed for each class on the test dataset using scikit-learn metrics.

Bar plots are created to visualize these metrics.

**ROC Curve and AUC Score:**

The Receiver Operating Characteristic (ROC) curve and Area Under the Curve (AUC) score are calculated for binary classification (assuming a binary problem).

The ROC curve is plotted, and the AUC score is printed.

**Saving the Model:**

The trained model is saved in HDF5 format (".h5").

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**svmmodel.ipynb**

1. Import necessary libraries, including joblib for saving and loading the SVM model.
2. Define constants such as IMAGE\_SIZE for resizing images.
3. Define a function preprocess\_and\_extract\_features(image\_path) that reads an image, resizes it, converts it to grayscale, and extracts HOG (Histogram of Oriented Gradients) features from the image.
4. Define a function generate\_segmentation\_mask(image\_path) that reads an image, resizes it, converts it to HSV color space, and generates a binary segmentation mask for detecting skin color regions.
5. Load and preprocess the dataset, including generating segmentation masks for each image. The dataset consists of images categorized into "benign" and "malignant" classes.
6. Split the dataset into training, validation, and test sets using train\_test\_split from scikit-learn.
7. Initialize lists to store training and validation accuracy values for each epoch.
8. Create an SVM (Support Vector Machine) classifier with a linear kernel and set probability=True to enable probability estimates required for ROC curve and AUC calculation.
9. Train the SVM classifier in a loop for a specified number of epochs, calculating and storing training and validation accuracies at each epoch.
10. Print the test accuracy of the trained SVM classifier on the test set.
11. Calculate precision, recall, and F1-score on the test set using scikit-learn's metrics functions and print these metrics.
12. Plot precision, recall, and F1-score in a bar chart using Matplotlib for visualization.
13. Plot training and validation accuracy across epochs to visualize the model's learning progress.
14. Calculate the ROC (Receiver Operating Characteristic) curve and AUC (Area Under the Curve) for the SVM classifier on the test set using roc\_curve and auc functions from scikit-learn.
15. Plot the ROC curve with the AUC score indicated on the plot. The curve shows the trade-off between true positive rate and false positive rate.
16. Print the AUC score, which quantifies the model's ability to discriminate between the two classes.
17. To save the trained SVM classifier for future use, use joblib.dump to save the model as a file named "skin\_cancer\_svm\_classifier.pkl."
18. To load the saved SVM model for future predictions, use joblib.load to load the model from the saved file.