Shoe Classification Task

Amin Ranjbar

Abstract: In developing an artificial intelligence (AI) shoe classification system, three methods were employed based on the principle of parsimony to determine the most suitable AI model for the task. These methods included fine-tuning an end-to-end pre-trained convolutional neural network (CNN) classifier, feature extraction via a pre-trained CNN with a separated classification pipeline, and training a model from scratch. In our implementation, we use both EfficientNetB0 and MobileNetV2 which are renowned CNN architectures for image classification. EfficientNetB0 excels in balancing accuracy and model size through innovative scaling methods. Meanwhile, MobileNetV2 prioritizes efficiency for mobile and edge devices with depthwise separable convolution. The shoe classification model was trained on a diverse dataset encompassing six classes of shoes: boots, sneakers, flip-flops, loafers, sandals, and soccer shoes. Each category consisted of 249 JPEG images, showcasing a wide array of shoe variations, styles, and designs. This comprehensive dataset aimed to enrich the training data and enhance the model's ability to generalize and accurately classify shoes across different categories. In the selection of machine learning algorithms for the AI model, the principle of parsimony guided the decision-making process. This principle emphasized choosing the most straightforward model that effectively addresses the classification task before considering more complex alternatives. The aim was to strike a balance between model simplicity and performance, opting for a model that delivers satisfactory results without unnecessary complexity. By applying these methods based on the principle of parsimony, the AI system aimed to achieve optimal classification accuracy and efficiency in identifying and categorizing different types of shoes. The integration of diverse training data, meticulous dataset curation, and thoughtful algorithm selection underscored a methodical and pragmatic approach to developing an effective AI shoe classification system. The results show that simpler model i.e. MobileNet_v2 combined with a fully connected network achieves better performance compared rest of the models for this dataset.

Keywords: Shoe dataset, Classification, Fine-tuning, Convolutional neural network

1. Introduction

1.1. Description of the AI system

The AI system being developed is a shoe classification model trained on a dataset comprising six classes of shoes: boots, sneakers, flip-flops, loafers, sandals, and soccer shoes. Each shoe category is represented by 249 images, with varying sizes but all images are in JPEG format. The dataset encapsulates a broad range of shoe variations, capturing the intricacies of each shoe type. From different perspectives and environments to various styles and designs, the dataset offers a comprehensive representation of shoes, enriching the training data for the classification model. This diversity ensures that the AI system is exposed to a wide array of shoe attributes, enhancing its ability to generalize and accurately classify shoes across different categories.

1.2. Data sources

The dataset was meticulously compiled by scraping images from Yahoo Images and unsplashed using a custom Python script created by the dataset publisher. The script interacted with the APIs of these platforms to retrieve images based on specific queries related to each shoe type. This process enabled the collection of a diverse set of shoe images encompassing different angles, scenarios, and variations commonly encountered in real-world settings.

2. Methods

2.1. Principle of parsimony in selecting machine learning algorithms

In the context of choosing the right AI model in a parsimonious manner, the **principle of parsimony** refers to selecting the simplest, most straightforward model that adequately solves the problem at hand before considering more complex or sophisticated models. The goal is to strike a balance between model complexity and performance, choosing the simplest model that provides satisfactory results without unnecessarily complicating the analysis.

Here are some key points to consider when applying the principle of parsimony in selecting the right AI model:

1. Start with Simple Models: Begin by exploring simpler, interpretable models like linear regression, logistic regression, or decision trees. These models are easy to understand, implement, and interpret, making them suitable for initial analysis.
2. Consider Model Complexity: Evaluate the complexity of each model concerning the complexity of the problem. A more complex model may offer higher accuracy, but it also comes with increased computational costs and potential overfitting.

3. Evaluate Performance: Assess the performance of each model using appropriate metrics and validation techniques. Compare how well each model generalizes to unseen data and makes predictions accurately.
4. Incremental Complexity: Gradually increase the complexity of the models if simpler models fail to meet the desired performance standards. Consider adding features, increasing model capacity, or exploring more advanced algorithms as needed.
5. Domain Knowledge: Using domain knowledge and insights to guide the model selection process. Understand the characteristics of the data and the problem domain to choose models that are well-suited to the task at hand.

Figure 1. Principle of parsimony in selecting the right machine learning algorithms

2.2. Right machine learning models

By following these guidelines and prioritizing simplicity and effectiveness in model selection, we can apply the principle of parsimony to choose the right AI model for our specific problem while avoiding unnecessary complexity and over-engineering. Thus, to address the classification task for this dataset in a parsimonious manner, where simpler solutions are considered before more complex models, we could start with the following classification methods:

Simplest Solution: Fine-tunning an end-to-end pre-trained CNN classier (FTCNN)

CNNs are specifically designed for image classification tasks and have shown state-of-the-art performance in various domains. As more complex models, they can capture intricate patterns and variations in the shoe images. Here, we use fine-tuning an end-to-end pre-trained CNN classifier which involves taking a pre-trained CNN model, usually trained on a large dataset like ImageNet, and adapting it to a specific classification task by retraining the model on a new dataset with a different set of targets classes. This process allows us to use the learned features from the pre-trained model while customizing the final layers of the network to suit the new classification problem.

Here's an explanation of the key steps involved in fine-tuning an end-to-end pre-trained CNN classifier:

1. Select a Pre-trained CNN Model: Choosing a pre-trained CNN model that has been trained on a large image dataset for tasks like image classification. Popular choices include models like VGG, ResNet, Inception, or MobileNet that have achieved state-of-the-art performance on various benchmarks.

2. Modify the Output Layers: Remove the original classification layer(s) from the pre-trained model and replace them with new fully connected layers that match the number of classes in the new dataset. These new layers will be randomly initialized and trained from scratch to adapt the model to the specific classification task.

3. Freeze Base Layers: Depending on the size of the new dataset and the similarity between the original dataset and the new task, we may choose to freeze some of the earlier layers of the pre-trained model to prevent them from being updated during training. This can help retain the learned features while focusing on fine-tuning the higher-level features.

4. Fine-tuning Process: Train the modified CNN model on the new dataset using techniques like backpropagation with a suitable optimizer (e.g., Adam or SGD) and a loss function appropriate for the classification task (e.g., categorical cross-entropy). The model learns to extract features that are relevant to the new classes while retaining the valuable knowledge gained from the original training.

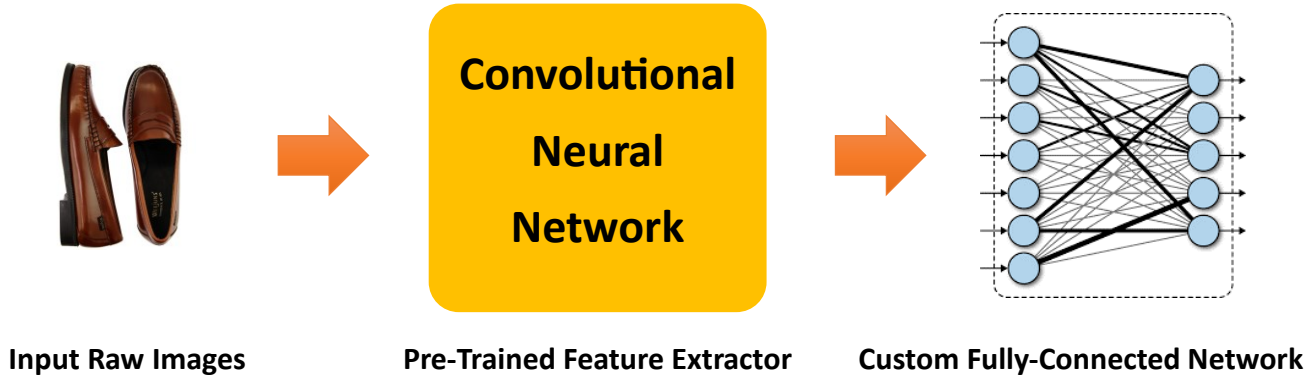

Figure 2. Block diagram of fine-tunning an end-to-end pre-trained CNN model

Thus, by adjusting the model's weights through fine-tuning, we can customize the model to the nuances and characteristics of the shoe images, improving its performance on the target task. This approach offers several advantages and benefits:

Faster Training: Since the pre-trained models have already learned generic features from diverse datasets, we can significantly reduce the training time and computational resources required to train a model from scratch. **This can be particularly advantageous when working with limited data**.

Improved Generalization: Transfer learning helps in transferring knowledge and representations learned from one domain (general images) to another domain (shoe images). This can lead to improved generalization and performance on the shoe dataset, **especially when the dataset is small and may not capture the full variability of shoe images**.

Regularization: Transfer learning with pre-trained models can act as a form of regularization by starting with a model that has already learned to generalize well on diverse data. This can help prevent overfitting, especially in scenarios where the dataset is small and prone to overfitting.

By employing transfer learning with pre-trained models and fine-tuning them on the shoe dataset, we can harness the power of deep learning and benefit from the prior learning on large-scale image datasets to improve the classification accuracy and efficiency of the shoe classification task.

Moderate Solution: Feature extraction via a pre-trained CNN and separated classification pipeline (FECNN)

Using processed features from a pre-trained CNN feature extractor and an offline classifier refers to leveraging the extracted features from a pre-trained CNN model as input to a separate classifier that is trained outside the CNN architecture. This approach can be beneficial in scenarios where we want to decouple the feature extraction process from the classification task, allowing for flexibility and efficiency in the overall model design.

Here's an explanation of the key steps involved in using processed features from a pre-trained CNN feature extractor and an offline classifier:

1. Feature Extraction with Pre-trained CNN: Start by using a pre-trained CNN model to extract meaningful features from images. The CNN model serves as a feature extractor, capturing high-level representations of input images in the form of feature vectors. Popular pre-trained CNN models like VGG, ResNet, Inception, or MobileNet are commonly used for this purpose.
2. Processed Feature Extraction: Once the pre-trained CNN model has extracted features from the input images, the extracted features are processed to create a feature representation for each image. These features capture the important characteristics of the input data in a lower-dimensional space, making them suitable for downstream classification tasks.
3. Offline Classifier Training: After extracting the processed features, an offline classifier is trained using these features as input. This classifier can be any machine learning algorithm, such as Support Vector Machines (SVM) and Logistic Regression. The classifier learns the patterns and relationships between the extracted features and the target labels in the training data.

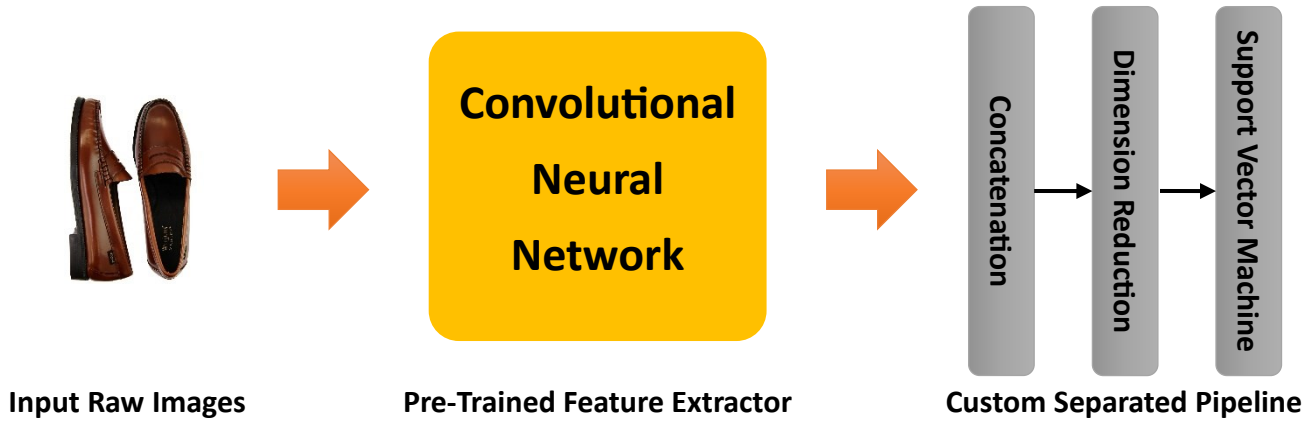

Figure 3. The block diagram of the model uses processed features from a pre-trained CNN and a separated classier

Here, we aim to use the Support Vector Machines (SVM) model as a classifier for post-processed CNN feature maps for the following reasons:

- SVMs are effective for high-dimensional data like images and handle non-linear decision borders.
- They are good choices for multi-class classifications and offer good generalization performance.

By starting with logistic regression and progressively moving towards more complex models, we can follow the principle of parsimony while exploring the classification of the 6-class shoe dataset.

Most Complex Solution: Training a model from scratch (TFS)

Training a model from scratch refers to the process of building and training a neural network or machine learning model without leveraging pre-trained models or feature extractors. It is often necessary when working with new or specific datasets for which pre-trained models may not be directly applicable. While it can be more computationally intensive and may require a larger dataset to prevent overfitting, training a model from scratch provides the flexibility to tailor the model specifically to the problem at hand. Here, we also try this method of learning model for our shoe dataset.

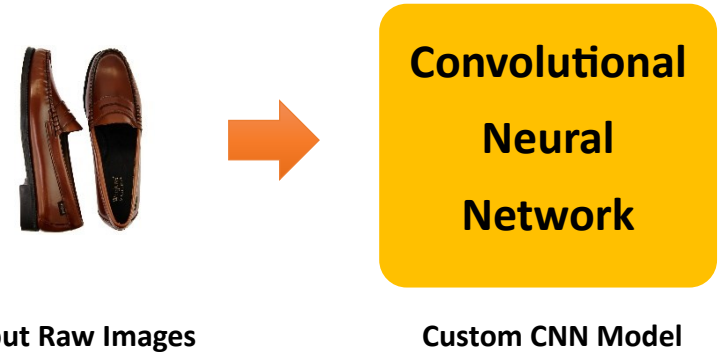

Figure 4. The block diagram of the model uses processed features from a pre-trained CNN and a separated classier

2.3. Detailed description of system architecture

In the realm of image classification, the CNN architectures of EfficientNetB0 and MobileNetV2 stand out as powerful and efficient models. EfficientNetB0, part of the EfficientNet family of models, is known for its exceptional performance in balancing model size and accuracy. Its innovative architecture incorporates a compound scaling method to optimize model depth, width, and resolution, leading to superior classification capabilities while maintaining computational efficiency.

On the other hand, MobileNetV2 represents a lightweight yet high-performing CNN architecture designed for mobile and edge devices. With its emphasis on efficiency and speed, MobileNetV2 utilizes depth-wise separable convolution and linear bottleneck layers to achieve impressive accuracy in image classification tasks while minimizing computational resources and model size.

We use these two models in the implementation of this AI system, based on the principle of parsimony described in the previous section:

- **Simplest solution**: Finetuning CNN classifier uses MobileNet_v2 and also EffifcientNetB0 vision transformer as a feature extractor

The first architecture, i.e. fine-tuned MobileNet_v2 showcases a neural network model constructed using TensorFlow's Keras API, featuring a pretrained MobileNetV2 as the base network. In the customization phase, a sequential model is constructed to extend the base MobileNetV2 architecture. This extension includes a dropout layer with a 0.1 dropout rate to mitigate overfitting by randomly deactivating a fraction of neurons during training. Subsequently, a dense layer with 64 units and a ReLU activation function has been added to introduce non-linearity and enhance the model's capacity to capture complex patterns within the data.

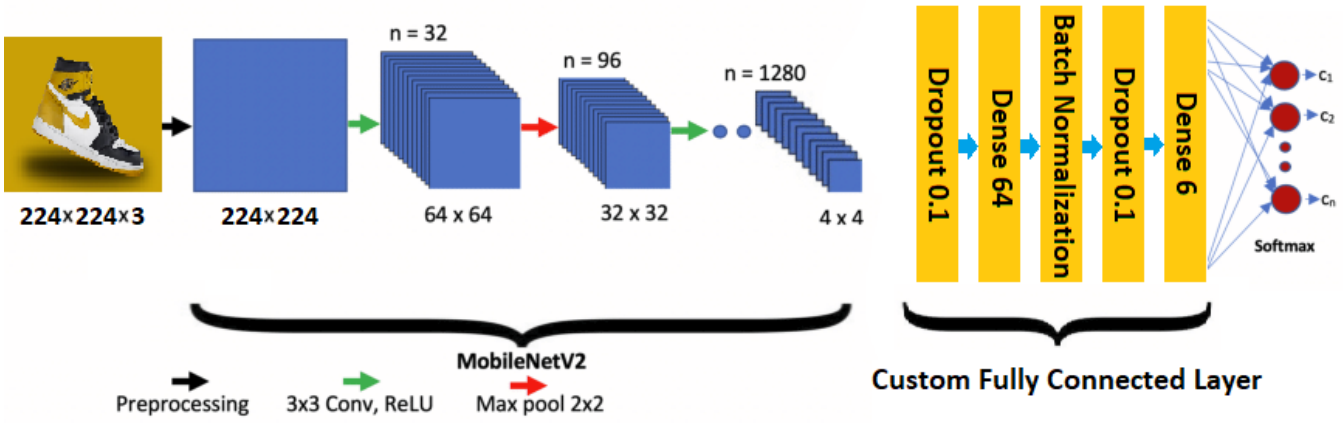

Figure 5. Schematic diagram of the fine-tuned MobileNet_v2 with custom FCN

Notably, this dense layer incorporates kernel regularizers (L1 and L2) to impose regularization constraints on the layer's weights, thereby enhancing the model's generalization capabilities. Furthermore, a batch normalization layer has been integrated to normalize the activations from the preceding layer, helping stabilize the model's learning process and reduce internal covariate shifts. Another dropout layer with a 0.1 dropout rate follows the batch normalization, providing additional regularization to prevent overfitting.

The second architecture, i.e. EfficientNetB0 architecture, the EfficientNetB0 model is utilized from TensorFlow's Keras applications, initializing it with pre-trained weights from the ImageNet dataset. By setting trainable to False, I have frozen the weights of the pre-trained EfficientNetB0 model, preventing them from being updated during the training process and preserving the pre-learned features. To customize the inference part of the model, a Global Average Pooling 2D layer is added to the output of the pre-trained EfficientNetB0 model. Global Average Pooling reduces the spatial dimensions of the feature map, summarizing the features captured by the model across the spatial dimensions. This pooling layer is efficient in reducing model complexity and preventing overfitting by aggregating spatial

information. Additionally, a Dense layer with a softmax activation function is appended to the output of the Global Average Pooling layer. This final Dense layer contains the number of classes in the classification task and applies the softmax function to generate class probabilities for multi-class classification.

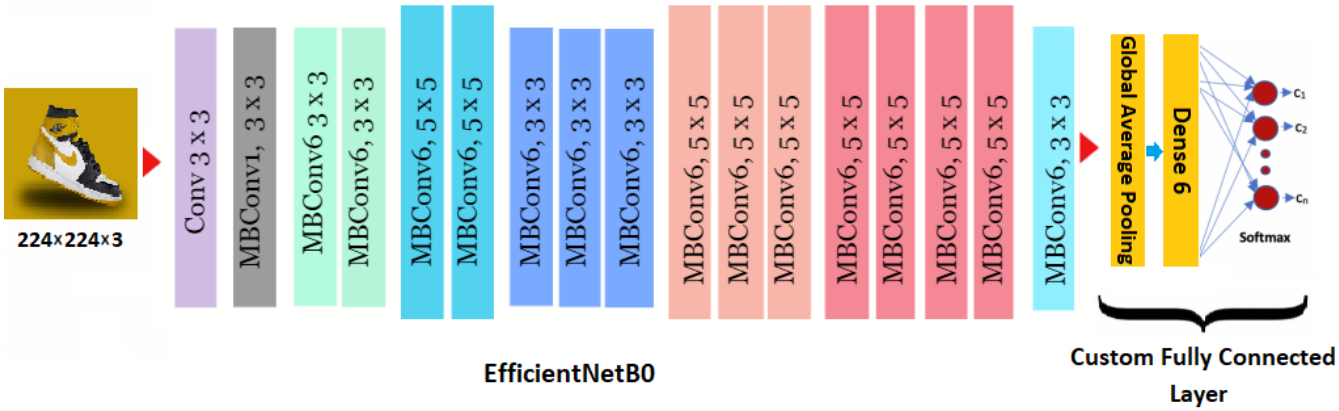

Figure 6. Schematic diagram of the fine-tuned EfficientNetB0 with custom FCN

- **Moderate solution**: Separate classifier based on MobileNet_v2 as a feature extractor

This model utilizes a pre-trained model for the feature extraction phase. This feature extraction model is created using the last convolutional layer named 'block_16_project_BN' based on the summary of the pre-trained model. The feature extractor model is then frozen to ensure its weights are not updated during training. In the offline pipeline, Principal Component Analysis (PCA) is used for feature extraction with a variance-based selection of principal components. This reduces the dimensionality of the input data while retaining relevant information. Subsequently, an SVM classifier with a nonlinear Radial Basis Function (RBF) kernel is employed for classification tasks.

To optimize the model's performance, a Grid Search with cross-validation is conducted to determine the best hyperparameters for the SVM classifier. The parameter grid includes 'C' for regularization and 'gamma' for the kernel coefficient of the RBF. The Grid Search is performed with the specified number of cross-validation folds to ensure robust model evaluation. After hyperparameter tuning, the best model is selected based on the optimized parameters. This approach allows for fine-tuning of the SVM classifier within the PCA-SVM pipeline to enhance classification performance on the dataset. The iterative process of parameter optimization and model selection results in a well-tailored model capable of effectively handling feature extraction and classification tasks.

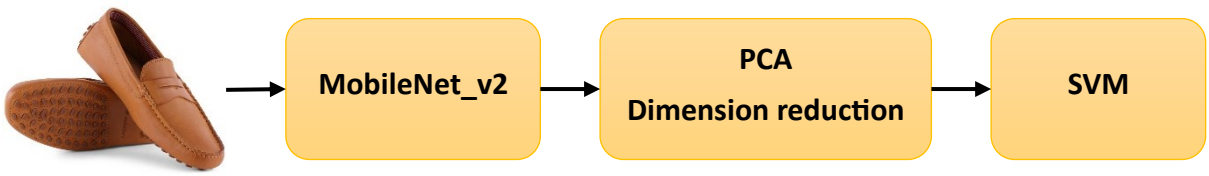

Figure 7. Schematic diagram of the separate classifier based on MobileNet base model and custom pipeline

2.4. Pre-processing Description

Our AI system employs a robust preprocessing pipeline to enhance the quality and effectiveness of the training data before feeding it into the classification model. The preprocessing steps in my implementation include **random rotation**, **random horizontal flip**, **resizing**, and **normalization**, designed to augment and standardize the input images for optimal learning and classification performance.

1. Random Rotation: Our system incorporates Random Rotation as a data augmentation technique to introduce variability and increase the diversity of the training dataset. By randomly rotating the images within a specified range, the model becomes more robust to different orientations of the shoes, enabling it to learn invariant features and generalize better to unseen instances. This augmentation strategy enhances the model's ability to cope with variations in the orientation of shoes present in the dataset, ultimately improving its classification accuracy.

2. Random Horizontal Flip: To further enhance the dataset's diversity and prevent overfitting, our AI system utilizes Random Horizontal Flip. This technique flips the images horizontally at random during training, creating additional variations while preserving the essential characteristics of the shoes. By introducing mirrored versions of the images, the model learns to recognize shoes irrespective of their orientation, leading to improved generalization and robustness in classifying shoe types accurately.

3. Resizing: The images in the dataset are resized to a standardized format to ensure uniformity in input dimensions across all samples. Resizing facilitates the efficient processing of images by fixing their dimensions to a predefined size, enabling the model to handle inputs of consistent shapes. This step prepares the images for seamless integration into the neural network architecture, streamlining the training process and enhancing the model's ability to learn discriminative features without being constrained by varying image sizes.

4. Rescaling: Before training, the pixel values of the resized images are scaled to a common range, typically ranging between 0 and 1 or -1 and 1. Scaling standardizes the pixel intensities across images, mitigating discrepancies in pixel values that could hinder the convergence of the model during training. By normalizing the input data, the system ensures that the neural network receives inputs in a consistent and optimized format, facilitating effective learning and improving the model's performance in accurately classifying shoe categories.

2.5. Performance metrics and evaluation

When evaluating trained models, we aim to use **accuracy**, **precision**, **recall**, and the **F1 score** as key metrics to assess their performance. These evaluation metrics provide valuable insights into the effectiveness and accuracy of the trained models in classification tasks. We also use a **confusion matrix**

to evaluate the performance of the classification model. Let's explore why these specific metrics were chosen and how they reflect the system's performance in real-world scenarios:

1. Accuracy: The accuracy metric in classification tasks measures the percentage of correctly predicted instances out of the total instances in the dataset. It indicates how well the model predicts class labels. While straightforward, accuracy may not account for imbalanced datasets. Other metrics like precision, recall, F1 score, or AUC may offer more comprehensive evaluations of model performance.
2. Precision: Precision is crucial in scenarios where the cost of false positives is high. For example, in medical diagnostics or fraud detection, misclassifying a negative sample as positive can have severe consequences. High precision indicates that the classifier correctly identifies positive samples while minimizing false positives, ensuring high confidence in the predicted positives. In real-world applications, precision helps assess the reliability and accuracy of the classifier in labeling positive instances.
3. Recall: Recall, also known as sensitivity, is vital when it is essential to capture all positive samples, even at the expense of misclassifying some negatives. For instance, in disease diagnosis, missing a positive case can be more detrimental than false alarms. High recall signifies the classifier's ability to detect the majority of positive instances, highlighting its sensitivity to true positives. In real-world scenarios, recall provides insights into the model's effectiveness in capturing all relevant positive cases.
4. F1-score: The F-measure, incorporating both precision and recall, offers a balanced evaluation of a model's performance. By considering the harmonic mean of precision and recall, the F1-score provides a single metric that reflects the trade-off between precision and recall. In real-world applications, the F1 score helps in determining an optimal balance between minimizing false positives (high precision) and capturing all positive instances (high recall), offering a comprehensive assessment of the model's effectiveness.
5. Confusion matrix: Confusion matrix is a key tool in evaluating classification model performance in machine learning. It breaks down predictions into true positives, true negatives, false positives, and false negatives, enabling analysis of model behavior and calculation of performance metrics like accuracy, precision, recall, and F1 score. This information helps in identifying model errors, biases, and areas for improvement, leading to better decision-making and model optimization.

2.6. Deployment and maintenance

I have used a robust deployment and maintenance strategy for my AI software code by leveraging **Docker** and **Git**. Docker allows us to containerize our AI applications, packaging all the necessary dependencies, and also Git allows us to track changes to my code, collaborate with team members, and easily revert to previous versions if needed.

1. **Automated Deployment using Docker**:
   - Containerized AI applications to package dependencies into lightweight, portable containers
   - Ensured consistency in deployment processes across different environments
   - Simplified management of dependencies for efficient deployment and maintenance
2. **Version Control using Git**:
   - Tracked changes, collaborated with team members, and easily reverted to previous versions
   - Improved development workflow and ensured team members worked on the latest code version
   - Enhanced code management and version tracking for effective maintenance and updates.

3. Results

3.1. Implementation details

The implementation combines **Python 3.9** and **TensorFlow Keras 2.12** for the development and training of artificial neural networks. Each model is encapsulated within a single class, providing a structured and organized approach to model development. By defining each model within a dedicated class, we promote code modularity and encapsulation, making it easier to manage and maintain individual models separately. This design allows for clear delineation between different models, facilitating easier tracking and modifications specific to each model. Each class can contain the model architecture, training routine, evaluation methods, and any other relevant functionalities, ensuring a cohesive and self-contained structure for each model.

In this model implementation, we have utilized a **YAML file** to control and manage all the parameters used in the feature extraction and classification pipeline. By storing configuration parameters in a YAML file, we centralize the management of hyperparameters, file paths, model settings, and other variables, making it easier to modify and experiment with different configurations without changing the code directly. This approach enhances the flexibility and maintainability of the code base. The YAML file serves as a configuration file where key parameters such as input shape, variance ratio for PCA, SVM regularization parameter (C), SVM kernel coefficient (gamma), cross-validation fold value, and other parameters related to the model setup can be defined and modified. This allows for quick adjustments to the model settings without the need to manually search through the code.

3.1.1. Configuring the learning process

In the learning process of the implementation, the classification model is compiled using the **Adam optimizer** with a **learning rate of 0.001**. The Adam optimizer is known for its efficiency in optimizing neural networks by adjusting learning rates for each parameter individually. The loss function used in this implementation is **Categorical Cross-entropy**, which is commonly used in multi-class classification tasks. It measures the dissimilarity between the true class distribution and the predicted class distribution, guiding the model towards better classification accuracy. As well, the training settings are

configured as follows: validation split is set to 0.2 for validation data, the input image size is 224×224 for the input data resolution, a batch size of 8 is used for processing samples, and training occurs over 30 epochs to enhance model learning and prediction accuracy. These settings collectively influence the training process and model performance.

Besides, saving the best model checkpoint during training is a key practice in machine learning. By using the **ModelCheckpoint callback** in Keras to monitor criteria like validation loss, we saved the best version of the model observed during training. This helps prevent overfitting, track performance, and provide a reliable model for deployment or sharing. Lastly, the chosen evaluation metric is **accuracy**, which calculates the proportion of correct predictions made by the model. It is a fundamental metric for assessing the overall performance of the model in correctly classifying the input data.

3.1.2. Enhanced training efficiency with early stopping

Early stopping is a technique used in machine learning training to prevent overfitting and improve model performance. In our model training, we have implemented early stopping with the following parameters: monitoring the validation loss, setting a minimum change threshold of 0.0001, allowing for patience of 5 epochs before stopping, and restoring the best weights of the model. By monitoring the validation loss, the early stopping algorithm can identify when the model's performance starts to degrade on unseen data. The min delta parameter ensures that the algorithm only stops when there is a significant performance improvement. The patience parameter allows the model to continue training for a few more epochs to see if the performance improves before stopping. Lastly, restoring the best weights ensures that the model retains the best set of parameters when training is halted.

3.2. Classification results

3.2.1. End-to-end finetuning MobileNet_v2 classifier

This section presents the outcomes of for **simpler solution** i.e. training, evaluation, and testing the model MobileNet_v2 classifier. The training parameters are configured as discussed in previous sections, and the model's performance is assessed using distinct test data.

Table 1. The architecture of the re-trained MobileNet v2

Layer (type)	*Output Shape*	*# Parameters*
mobilenetv2_1.00_224 (Functional)	(None, 1280)	2257984
Dropout	(None, 1280)	0
Dense	(None, 64)	81984
Batch Normalization	(None, 64)	256
Dropout	(None, 64)	0
Dense	(None, 6)	390

The results obtained in terms of accuracy and loss for training and validation are as follows:

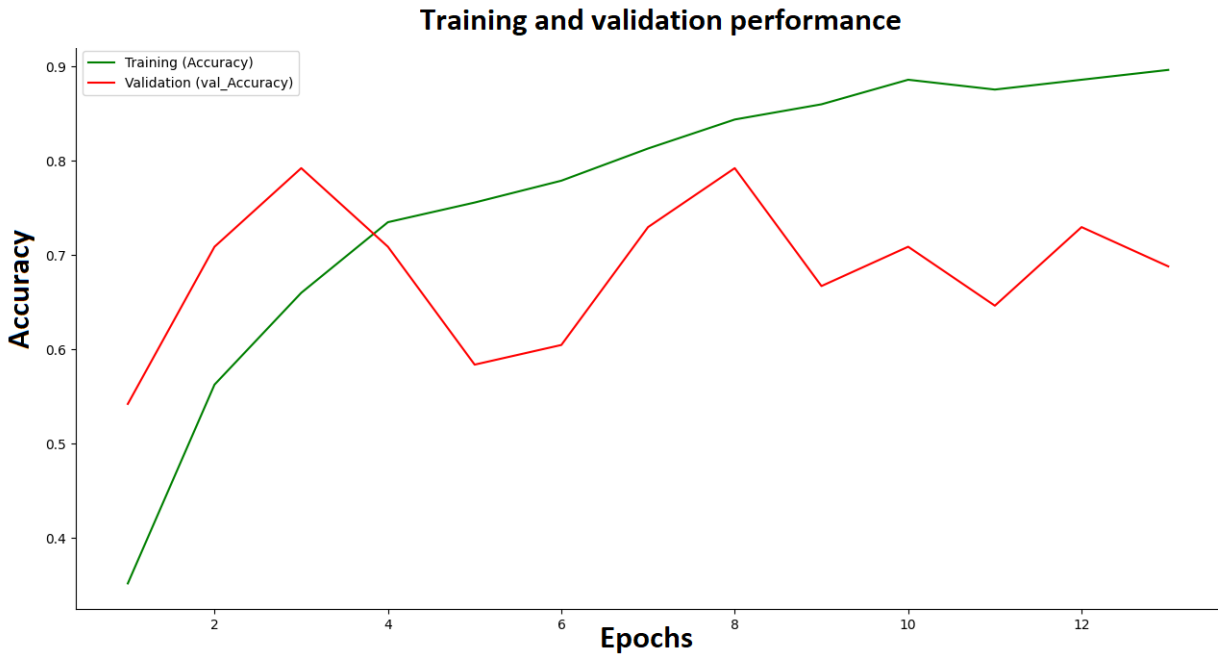

Figure 8. Training and validation performance during consecutive epochs

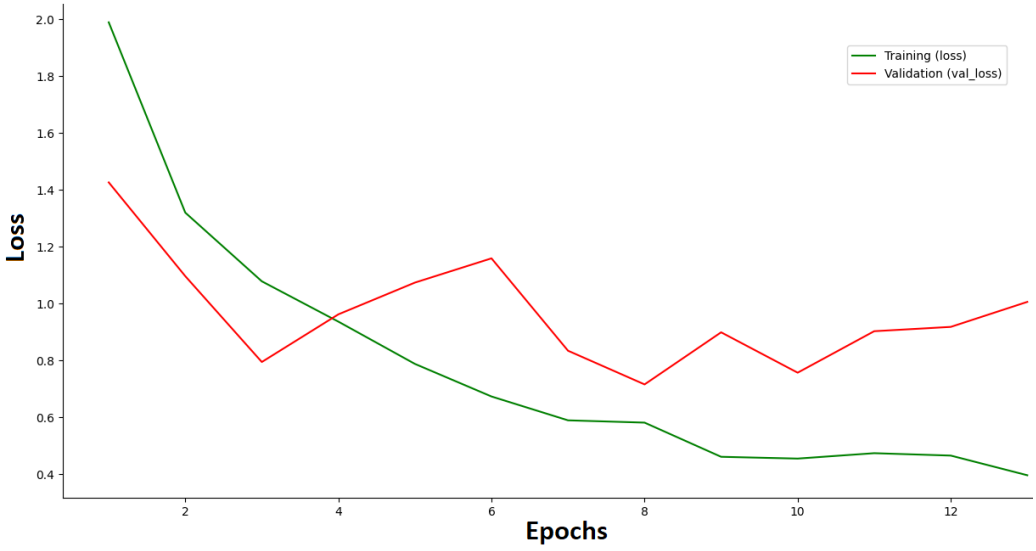

Figure 9. Training and validation loss during consecutive epochs

The results obtained in terms of accuracy, confusion matrix, precision, f1 score, and recall for testing on separate datasets are as follows:

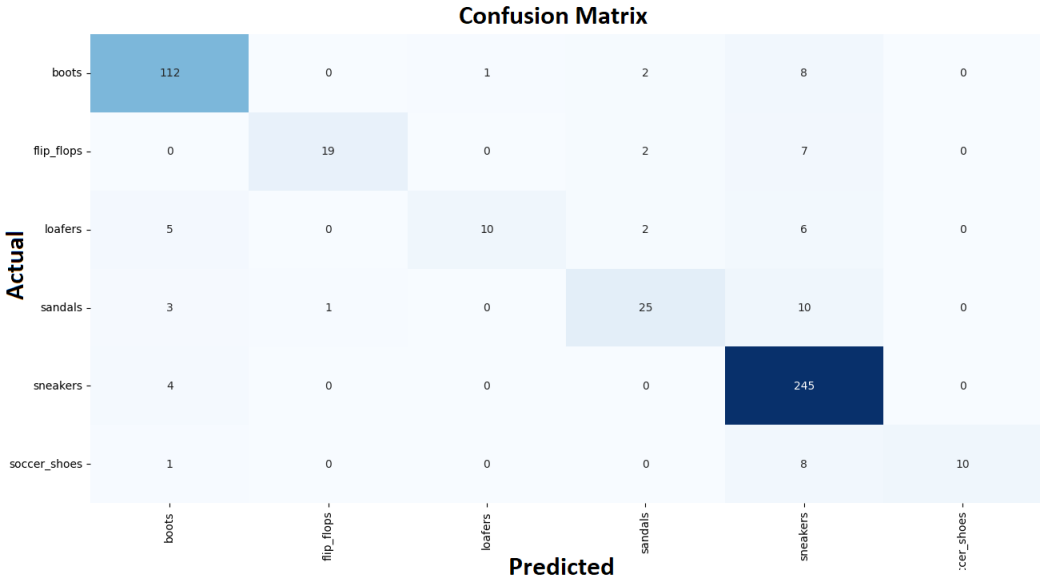

Figure 10. Confusion matrix for all test samples across different predicted classes

Table 2. Test results for model MobileNet_v2

	Precision	*Recall*	*f1-score*	*Accuracy*
boots	0.90	0.91	0.90	0.89
Flip_flops	0.95	0.68	0.79	
Loafers	0.90	0.43	0.59	
Sandals	0.80	0.64	0.71	
Sneakers	0.86	0.98	0.91	
Soccer_shoes	1.00	0.52	0.69	

3.2.2. End-to-end finetuning EfficientNetB0 classifier

This section presents the outcomes of for **moderate solution** i.e. training, evaluation, and testing of the model EfficientNet_B0 classifier. The training parameters are configured as discussed in previous sections, and the model's performance is assessed using distinct test data.

Table 3. The architecture of the re-trained EfficientNet B0

Layer (type)	*Output Shape*	*# Parameters*
EfficientNet B0 (Functional)	(None, 7, 7, 1280)	-
Dropout	(None, 1280)	0
Dense	(None, 6)	7686

The results obtained in terms of accuracy and confusion matrix for training, validation, and test are as follows:

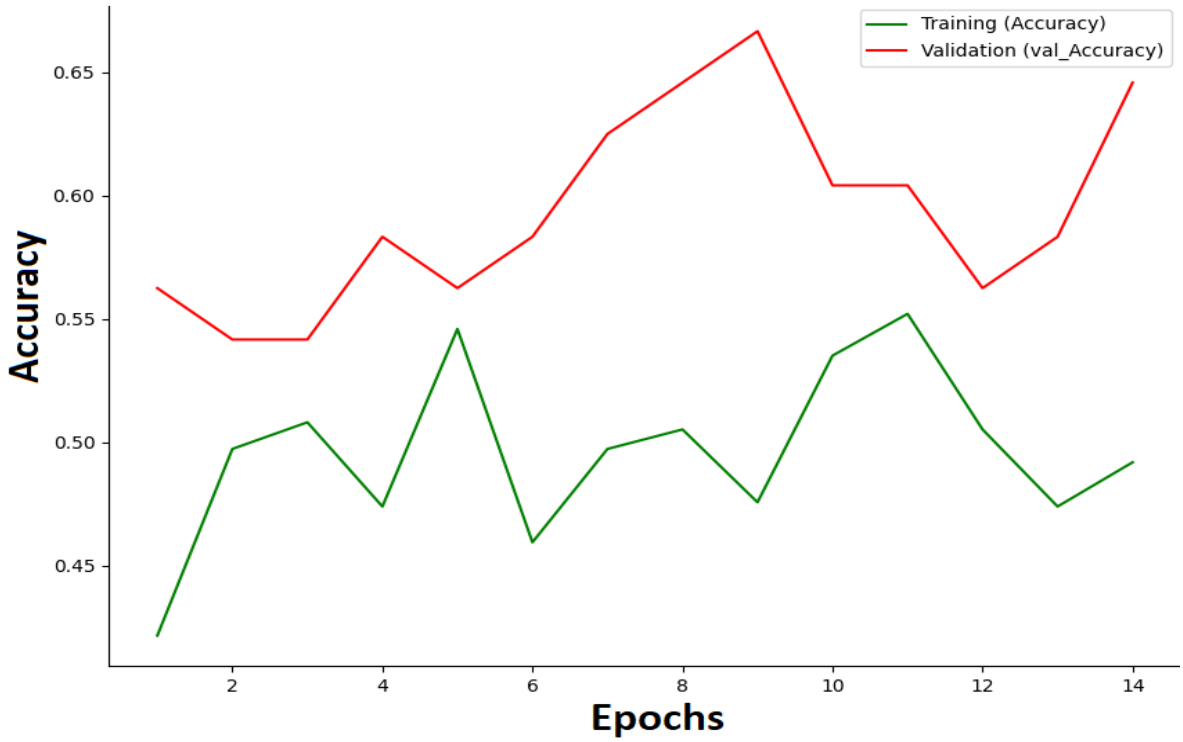

Figure 11. Training and validation performance during consecutive epochs

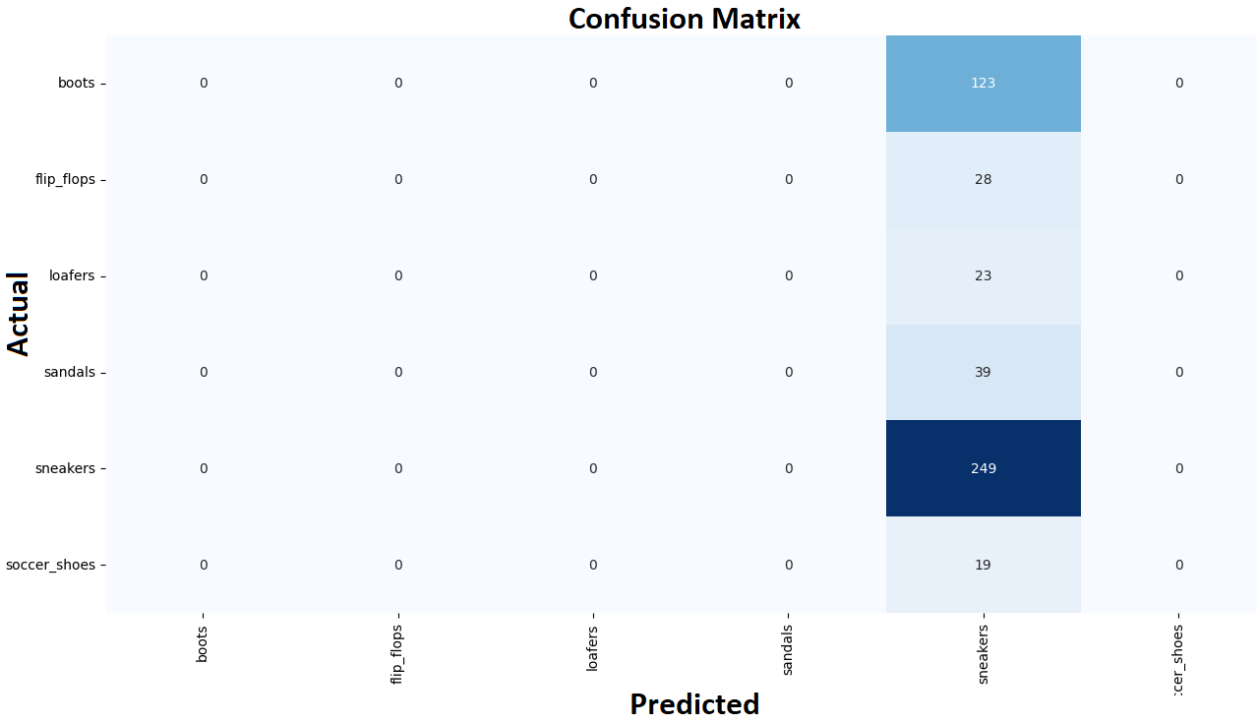

Figure 12. Confusion matrix for all test samples across different predicted classes

3.2.3. Finetuning separated classifier model

In this model design, the MobileNet V2 architecture is employed as a pre-trained classifier to extract features from the last layer. These extracted features are then subjected to dimension reduction techniques to retain **90% of the total variance**, enhancing the computational efficiency while preserving

the critical information contained within the data. Subsequently, a multi-class SVM model is utilized to classify the extracted features. To optimize the SVM model's hyperparameters, a grid search is conducted to find the optimal values for the regularization parameter C and the kernel coefficient gamma. The grid search is performed using a **cross-validation fold of 5**, ensuring robust model evaluation and tuning. The hyperparameter optimization process involves exploring a range of values for C and gamma to identify the combination that maximizes the SVM model's performance. Specifically, the **C values considered range from 0.1 to 2**, while the **gamma values range from 0.1 to 2**, enabling an extensive search across the parameter space to find the most suitable configuration for the SVM model.

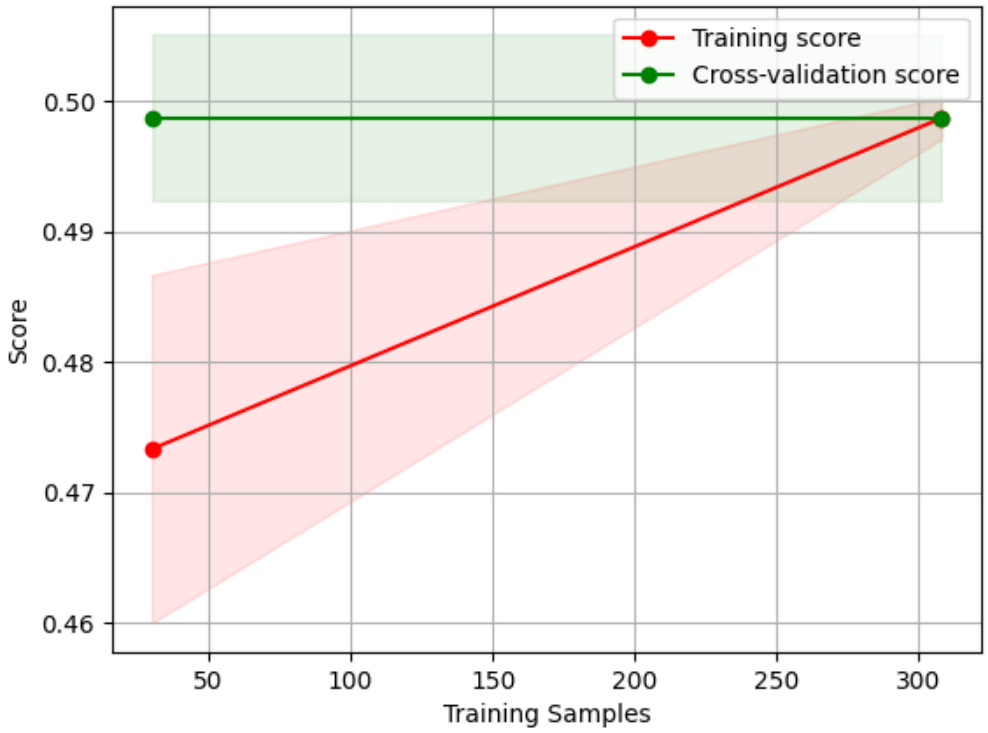

Figure 13. Training and validation performance for training and validation samples

4. Conclusion

The effectiveness of using MobileNet as a simpler model with a fully connected network, as opposed to EfficientNetB0 with a fully connected network, or MobileNet with PCA and SVM, showcases the advantages of simplicity in machine learning. By opting for a more streamlined architecture like MobileNet, we can achieve better performance through improved interpretability, faster training times, and reduced risk of overfitting. With MobileNet's efficient design and lightweight architecture, it focuses on crucial patterns in the data, leading to more robust results. This simplicity allows for faster optimization and better generalization on datasets with limited samples or noisy features, making it a practical choice for various applications.

Moreover, supplementing the MobileNet model with advanced training techniques such as data augmentation, regularization, and early stopping can further enhance its performance. These methods help fine-tune the model, prevent overfitting, and improve its ability to capture intricate data patterns effectively.

5. Future Improvements

- Testing new classification architectures

In the future, our implemented AI system can benefit from several enhancements to further improve its performance and flexibility. One key area of improvement involves exploring new classification architectures to enhance model accuracy and efficiency. By experimenting with state-of-the-art architectures such as **Transformer-based models** or **ensemble** methods, we can potentially achieve better results on complex datasets and tasks.

- Handling shared methods using parent classes

Using more object-oriented programming (OOP) principles can enhance the maintainability and scalability of our AI system. Introducing a parent class to handle shared methods and functionalities for different classifiers can streamline code development and reduce redundancy across multiple models. This approach promotes code reusability, facilitates easier model extension or modification, and enhances overall code organization and readability.

- Employing design patterns

Implementing design patterns (e.g., Factory, Strategy, Singleton) to address common programming challenges, enhance code readability, and facilitate future modifications and extensions.

- Performance optimization

Analyzing and implementing optimizations such as model distillation, and architecture search to improve the speed, memory footprint, and accuracy of the AI model.