Deep Learning Concepts and Architectures

[First Cycle of Experiments]

Name: Karunanidhi Ayyamperumal

Reg Number: 3122225002053

Class: IT-A

Github Link: Github Link

Experiment 1: Drawing a Confusion Matrix

Aim

The aim is to understand and visualize the performance of a classification model by computing and plotting its confusion matrix. This provides a detailed breakdown of correct and incorrect predictions for each class.

Python Libraries Needed

- **Scikit-learn:** Primarily for the confusion_matrix function to compute the matrix from true and predicted labels.
- **Matplotlib & Seaborn:** For creating a clear and well-annotated heatmap to visualize the computed matrix.
- **NumPy:** For handling the arrays of true labels, predicted labels, and the resulting matrix itself.

Algorithm/Flowchart

1. **Obtain Labels:** Start with two essential arrays: the **true labels** (y_true) and the **predicted labels** (y_pred) generated by a classification model.
2. **Compute Matrix:** Use the confusion_matrix() function from sklearn.metrics to generate a matrix where rows represent the actual classes and columns represent the predicted classes.
3. **Visualize the Matrix:**
    - Create a heatmap using seaborn.heatmap() or a similar plotting function.
    - Pass the computed matrix as the primary data.
    - Annotate the heatmap to display the numerical count in each cell.
    - Label the X-axis as "Predicted Labels" and the Y-axis as "Actual Labels".
    - Add class names as ticks on both axes for clarity.
4. **Display Plot:** Render the final visualization to analyze the model's performance.

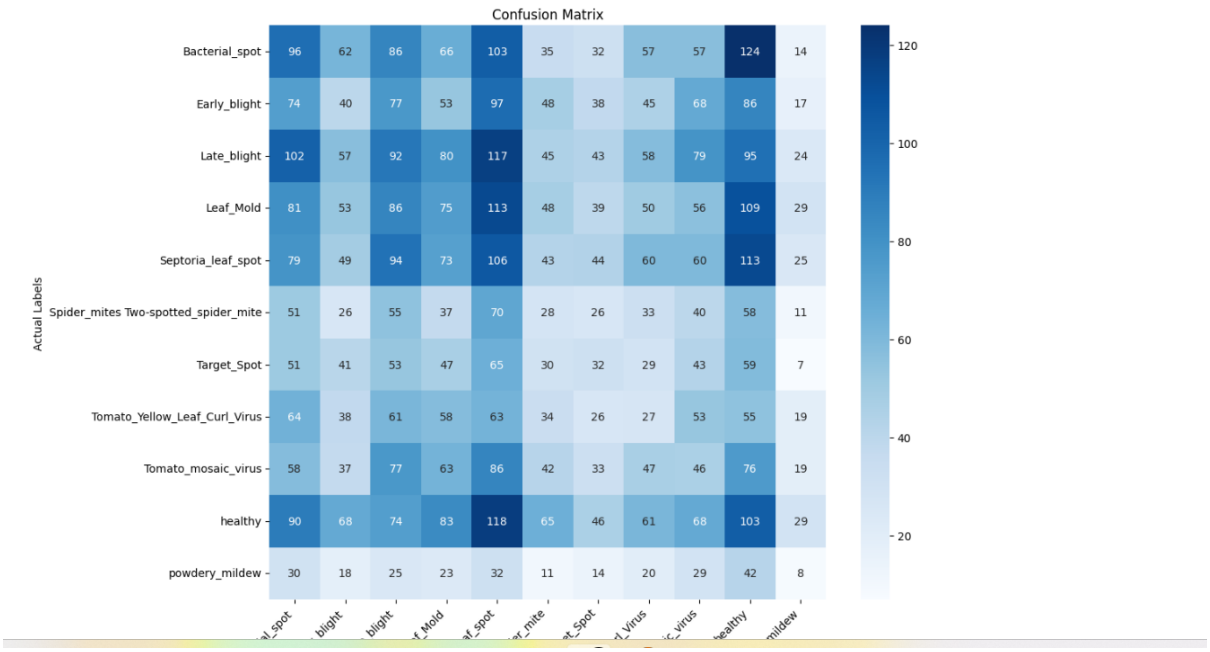
Results

The output is a confusion matrix, a powerful tool for evaluating a classifier. The key insights gained are:

- **True Positives (Diagonal):** The values along the main diagonal show the number of correct predictions for each class.
- **False Positives (Off-diagonal Columns):** Values in a column (excluding the diagonal) reveal how other classes are incorrectly predicted *as* that class.
- **False Negatives (Off-diagonal Rows):** Values in a row (excluding the diagonal) show how that class is incorrectly predicted *as* other classes.

This visualization provides a much deeper understanding of classification errors than a simple accuracy score, highlighting which specific classes the model is confusing.

Snapshots



Experiment 2: Layer Visualization of DenseNet

Aim

The objective is to visualize and understand the internal feature representations that the trained DenseNet model learns at its various layers[9].

Python Libraries Needed

- **TensorFlow/Keras or PyTorch:** To access the intermediate layers of the trained model.
- **NumPy:** To handle the feature map arrays.

- **Matplotlib:** To plot and display the feature map images.
- **OpenCV or PIL:** To load and preprocess the input image.

Algorithm/Flowchart

1. **Load Trained Model:** Load the DenseNet model that has been trained on the tomato leaf dataset.
2. **Select Input Image:** Choose a sample tomato leaf image to feed into the network[10].
3. **Extract Feature Maps:** Pass the image through the network and capture the outputs (feature maps) from various selected convolutional layers[11].
4. **Visualize:** Plot the extracted feature maps for each selected layer to observe the patterns the model is detecting.

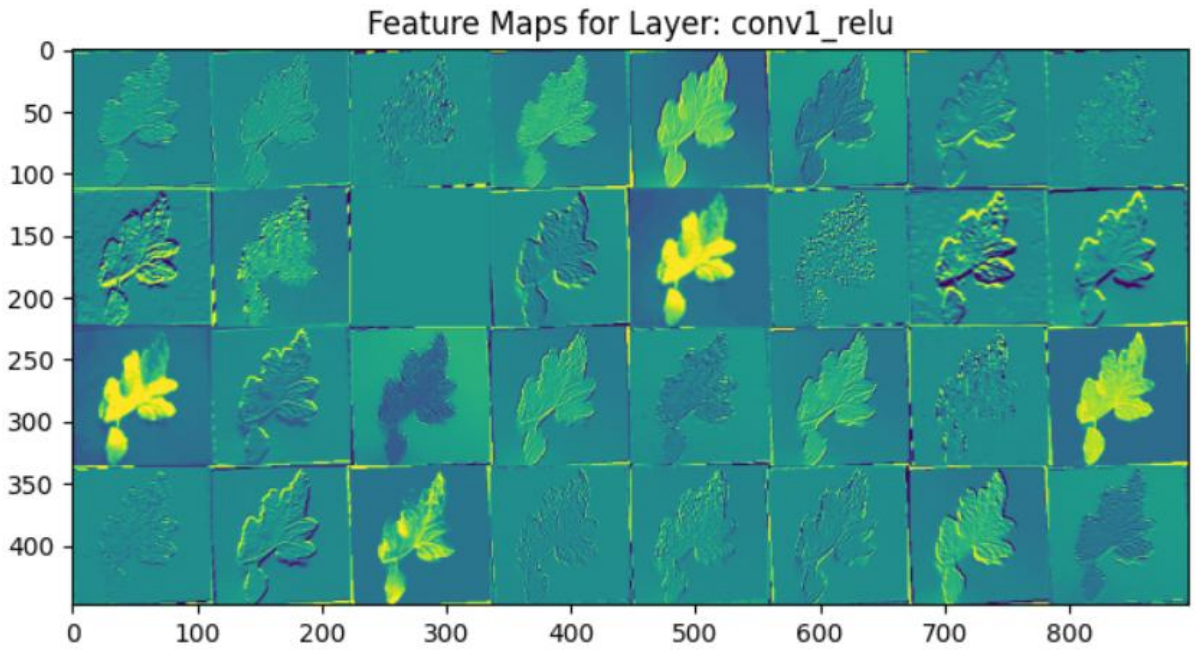
Results

The visualization confirmed that the model learns a hierarchy of features:

- **Early Layers:** Detect primitive patterns such as edges, colors, and simple textures on the leaves[12].
- **Mid Layers:** Combine basic features to identify more complex structures like leaf veins, the texture of spots, and the overall leaf shape[13].
- **Deeper Layers:** Capture highly abstract, class-specific features that identify unique markers of diseases, such as the concentric rings of Early Blight[14].

This confirms the model is learning relevant features for disease classification[15].

Snapshots



Experiment 3: Data Augmentation Techniques

Aim

The goal is to artificially increase the size and diversity of the training dataset to improve the model's ability to generalize to new images and to reduce overfitting[16].

Python Libraries Needed

- **TensorFlow/Keras (ImageDataGenerator) or PyTorch (transforms):** Core libraries for applying on-the-fly image augmentation.
- **NumPy:** For array manipulation if manual augmentation is performed.
- **Matplotlib:** To visualize the effects of the augmentation.

Algorithm/Flowchart

1. **Define Augmentation Pipeline:** Specify a series of random transformations to be applied to the training images.
2. **Implemented Techniques:**
    - **Shift:** Move the image horizontally or vertically[17].
    - **Flip:** Flip the image horizontally[18].
    - **Rotation:** Rotate the image by a random angle (e.g., up to 30 degrees)[19].
    - **Brightness:** Randomly alter the image brightness[20].

- **Zoom:** Randomly zoom into the image[21].

3. **Apply to Dataset:** Integrate the augmentation pipeline into the data loading process so that the model sees slightly different versions of the images in each training epoch.

Results

Data augmentation successfully created a more diverse training set[22]. This process forces the model to learn the fundamental, invariant features of each disease rather than memorizing the specific orientation, lighting, or position of leaves in the training images. This is crucial for achieving high accuracy on real-world data[23].

Experiment 4:Image classification using DenseNet

Aim

The aim is to evaluate the performance of an unmodified, pre-trained DenseNet121 model on the tomato leaf disease dataset without any fine-tuning. This is a "zero-shot" task to see how the model performs out-of-the-box[24].

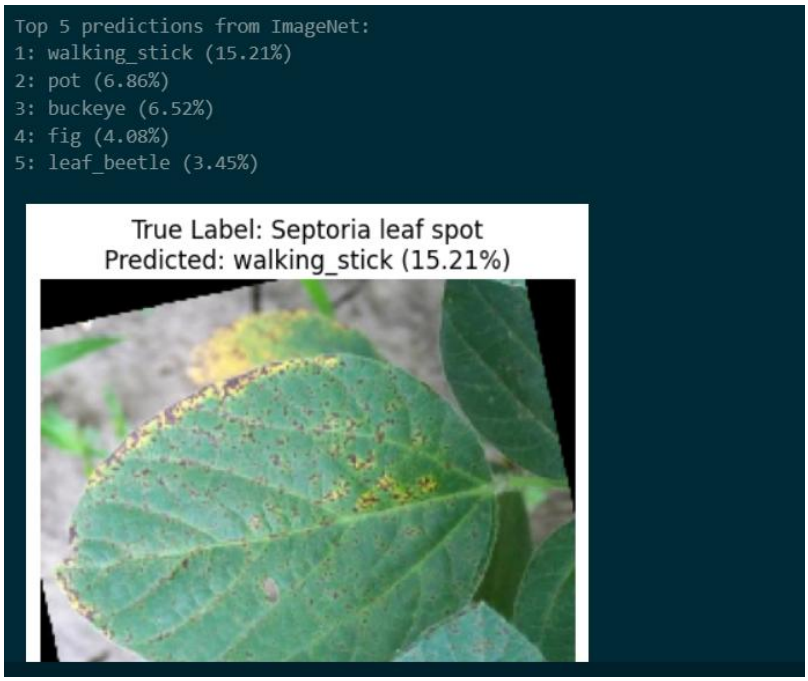
Python Libraries Needed

- **TensorFlow/Keras or PyTorch:** To load the pre-trained DenseNet121 model with its ImageNet weights.
- **PIL or OpenCV:** To load and prepare the test images.

Algorithm/Flowchart

1. **Load Pre-trained Model:** Load a standard DenseNet121 model with its original ImageNet weights and its final 1000-class classifier. No layers are modified[25].
2. **Prepare Test Data:** Load the 500 images from the testing set[26].
3. **Perform Inference:** Pass each test image directly into the unmodified model to get a prediction[27].
4. **Analyze Results:** Examine the model's output predictions.

Snapshots



Results

The model's predictions correspond to the 1000 ImageNet classes (e.g., 'fungus,' 'leaf,' 'dog') and not the 10 specific tomato disease classes[28]. As the model's output vocabulary does not match the task's required classes, it is fundamentally unable to perform the classification correctly[29]. This result highlights the necessity of adapting a pre-trained model using techniques like transfer learning or feature extraction for specialized tasks[30].

Experiment 5: Pre-trained CNN for Transfer Learning[DenseNet]

Aim

The objective is to leverage the knowledge from a pre-trained DenseNet model (trained on ImageNet) and

fine-tune it for the specific tomato leaf disease classification task.

Python Libraries Needed

- **TensorFlow/Keras or PyTorch:** To load the pre-trained model, freeze layers, and add a new custom classification head.
- **NumPy:** For numerical operations on image data.

- **Scikit-learn:** To generate the classification report and confusion matrix.
- **Matplotlib/Seaborn:** To visualize the confusion matrix.

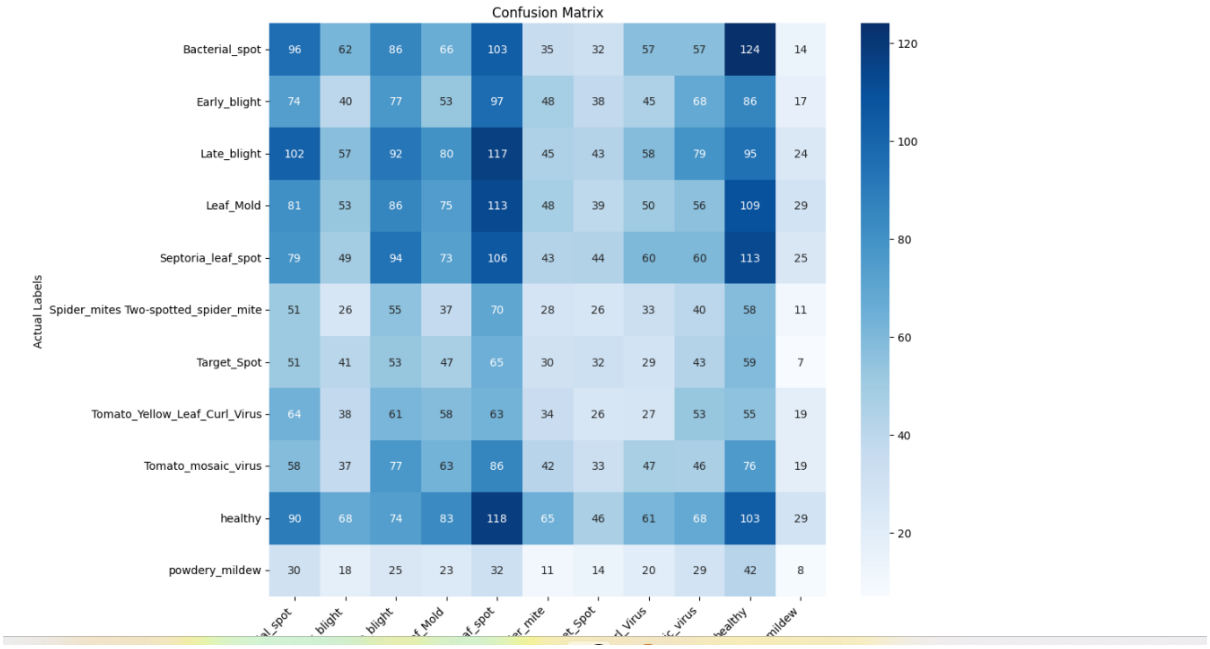
Algorithm/Flowchart

1. **Load Pre-trained Model:** A DenseNet121 model is loaded with its pre-trained ImageNet weights, but its final classification layer is removed.
2. **Freeze Base Layers:** The weights of the initial convolutional layers are "frozen". This preserves the powerful, general-purpose feature knowledge (like edges, textures, and shapes) they already have.
3. **Add Custom Classifier:** A new set of fully-connected layers is added on top of the frozen base, ending in a 10-class Softmax output layer for the tomato diseases.
4. **Train (Fine-Tuning):** The model is trained on the tomato leaf dataset, but only the weights of the new custom classifier layers are updated. This efficiently adapts the model to the new task.
5. **Evaluate:** The model's performance is measured using the test dataset.

Snapshots

Classification Report:

	precision	recall	f1-score	support
Bacterial_spot	0.12	0.13	0.13	732
Early_blight	0.08	0.06	0.07	643
Late_blight	0.12	0.12	0.12	792
Leaf_Mold	0.11	0.10	0.11	739
Septoria_leaf_spot	0.11	0.14	0.12	746
Spider_mites Two-spotted_spider_mite	0.07	0.06	0.06	435
Target_Spot	0.09	0.07	0.08	457
Tomato_Yellow_Leaf_Curl_Virus	0.06	0.05	0.05	498
Tomato_mosaic_virus	0.08	0.08	0.08	584
healthy	0.11	0.13	0.12	805
powdery_mildew	0.04	0.03	0.04	252
accuracy			0.10	6683
macro avg	0.09	0.09	0.09	6683
weighted avg	0.10	0.10	0.10	6683



Results

The model was evaluated on the test set, and its performance was captured in a classification report and a confusion matrix. According to the provided report, the

overall accuracy was 0.10. The precision, recall, and F1-scores for individual classes were also low, indicating that this specific fine-tuning attempt did not perform well

Experiment 6: Pre-trained CNN for Feature Extraction

Aim

The aim is to use the pre-trained DenseNet model

solely as a feature extractor. These extracted features are then fed into a separate, traditional machine learning classifier, such as a Support Vector Machine (SVM), to perform the classification.

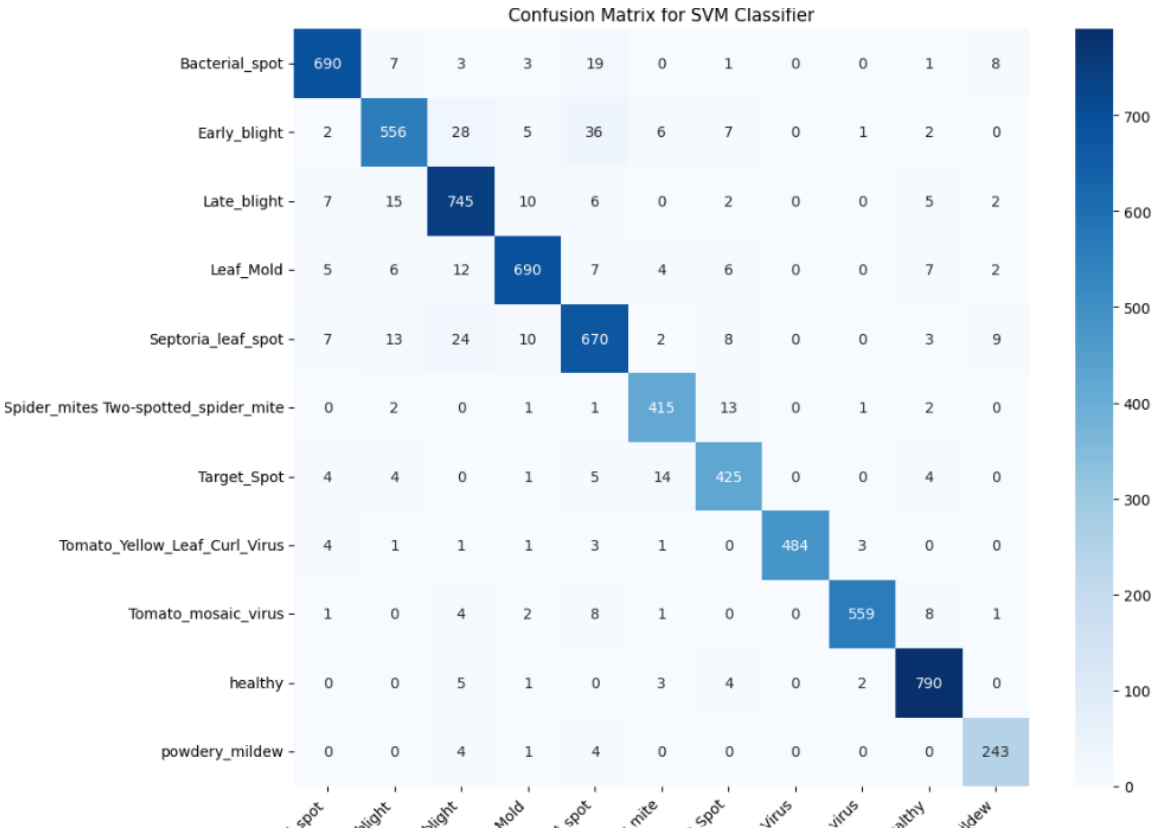
Python Libraries Needed

- **TensorFlow/Keras or PyTorch:** To load the pre-trained DenseNet model and make predictions to extract features.
- **NumPy:** To store the extracted feature vectors.
- **Scikit-learn:** To implement, train, and evaluate the Support Vector Machine (SVM) classifier.
- **Matplotlib/Seaborn:** To visualize the SVM classifier's confusion matrix.

Algorithm/Flowchart

1. **Load Feature Extractor:** DenseNet121 is loaded without its top layer, and all its weights are frozen.
2. **Extract Features:** Each image from the training and testing sets is passed through the frozen DenseNet. The output from the final pooling layer is saved as a fixed-length feature vector (an "embedding") for that image.
3. **Train External Classifier:** The extracted feature vectors from the training set are used to train a separate SVM classifier. The SVM learns to distinguish between classes based on these high-level features instead of raw pixels.
4. **Evaluation:** The trained SVM is then evaluated using the feature vectors extracted from the testing set.

Snapshots



Results

This method proved to be

highly effective. The feature extraction approach is much faster than fine-tuning because the deep network only performs a single forward pass per image. As shown in the confusion matrix and the computed metrics below, the SVM classifier achieved excellent performance, with an

overall accuracy of 0.94.