Title- **Comparative Analysis of Deep Learning Architectures for Image Classification: VGG, MobileNet, and ResNet**

Author-hardik chauhan

This project explores a deep learning pipeline for image classification using three widely used convolutional neural network (CNN) architectures: VGG, MobileNet, and ResNet. The main goal is to compare and evaluate the performance of these models in terms of classification accuracy, computational efficiency and robustness, on a real-world image dataset. The dataset is processed using various data augmentation and preprocessing techniques, including resizing, random rotations, horizontal flipping, blur, etc. These augmentations aim to introduce diversity into the dataset, simulating various real-world conditions to help improve model generalization and reduce overfitting. Each model is also normalized using the standard deviation values and the mean values associated with the architecture being used, ensuring optimal input scaling for MobileNet, ResNet, and VGG. This enables a fair comparison of their capabilities.

The project also investigates the effect of different hyperparameters (like batching, learning rate, etc.) on the accuracy scores of these models. Moreover, the impact of using pre-trained weights versus training the models from scratch is explored, providing insights into how fine-tuning affects accuracy. By leveraging these transformations and configurations during the training phase, the models recognize features and patterns in the images more effectively, while also learning to handle variations in the input data. The results of this study aim to provide a comprehensive understanding of the strengths and weaknesses of each model, aiding researchers in selecting the most suitable architecture for a particular image classification tasks.

Chapter 1

Introduction

Image classification/ recognition is one of the core tasks in the field of computer vision and plays a crucial role in a wide range of applications, from healthcare and autonomous driving to security and entertainment. Over the past decade, deep learning techniques, particularly Convolutional Neural Networks (CNNs), have revolutionized the field of image processing, delivering state-of-the-art performance across various domains. Among the most successful

CNN architectures are VGG, MobileNet, and ResNet, each with unique design philosophies and strengths.

This project aims to explore and compare these three CNN architectures for their performance in the task of image classification. VGG, known for its simplicity and deep architecture, was one of the first models to show that deeper networks could significantly improve image classification results. MobileNet, on the other hand, is computationally efficient, making it suitable for use on devices that do not have access to high computational power and other resource-constrained environments. ResNet deals with the problem of vanishing gradients using its innovative residual connections, enabling construction of even deeper architectures without compromising performance.

The success of the models depends on the quality of the input data and how well it is preprocessed. Data augmentation and transformation techniques help in enhancing the diversity of the dataset, which helps increase the model's ability to generalize. In this project, several transformation techniques, such as resizing, rotation, color jitter, and Gaussian blur, are applied to the images to simulate real-world variations and improve model robustness.

Furthermore, the impact of using pre-trained weights versus training models from scratch is explored. Pre-trained models, which are previously trained on large datasets like ImageNet, often lead to faster convergence and better performance, especially when the available data is limited or similar to the dataset used for pre-training. On the other hand, training models from scratch provides a deeper understanding of the model's learning process and can be beneficial for unique or domain-specific tasks.

By evaluating these architectures on a real-world dataset, this project seeks to give insights on the weaknesses and strengths, helping through selection of the most appropriate model for various image classification tasks. Through rigorous experimentation, the comparison of performance across different models and training strategies aims to give valuable knowledge to the field of computer vision.

Chapter 2

Literature Review

Here, we give an overview of the fundamental concepts, architectures, and frameworks used in this project. The focus is on the deep learning models implemented—VGG16, MobileNetV2, and ResNet18—and the tools used for training and evaluation, including PyTorch and Kaggle's environment. We also highlight the specific code and libraries used.

2.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a type of deep learning models that are highly successful in image processing and classification tasks. CNNs automatically and adaptively learn spatial hierarchies of features from input images, making them more effective than traditional machine learning algorithms that rely on handcrafted features. CNNs consist of multiple layers such as pooling layers, convolutional layers, and fully connected layers. Convolutional layers detect features such as edges, textures, and patterns. Pooling layers reduced the dimensionality of the image and retaining essential information, while fully connected layers help in producing the output of the final image classification result.

CNNs have presented state-of-the-art results in image classification tasks, even medical image analysis. The ability of CNNs to learn the image features directly using the raw pixels of the image has revolutionized computer vision, making them an essential tool for tasks that require automated image understanding.

2.2 Visual Geometry Group

The VGG architecture, introduced by the Visual Geometry Group at the University of Oxford. Visual Geometry Group networks are differentiated by their simplicity and depth, using small 3x3 convolutional filters and max-pooling layers stacked together in a deep network structure. VGG is highly effective in large-scale image identifying tasks, with variants like VGG-16 and VGG-19 being widely used for benchmarking. However, one of the drawbacks of VGG is the large number of parameters it requires, making it computationally expensive and resource-intensive.

2.3 MobileNet

MobileNet is a lightweight Convolutional Neural Networks architecture designed to perform well with limited computational resources. The key feature of MobileNet is the use of depthwise separable convolutions, which reduces the computational cost and the number of parameters required when compared with traditional convolutions. This enables MobileNet to achieve high accuracy while being computationally efficient. MobileNet has become particularly popular for applications in real-time image classification tasks, such as object detection on mobile devices, where computational efficiency is critical.

2.4 ResNet

ResNet (Residual Networks) introduced the concept of residual connections, which allow the network to skip over certain layers. This innovation helps mitigate the problem of vanishing gradients, which occurs when training very deep networks. Residual connections make it possible to train extremely deep networks (hundreds of layers) without losing performance. ResNet with variants like ResNet-18, ResNet-50, ResNet-101, and ResNet-152 has become one of the most widely used architectures for image classification tasks.

2.5 Data Preprocessing and Augmentation techniques

Data augmentation and preprocessing techniques play an essential role in improving the performance of deep learning models, particularly when the dataset is small or lacks variability. Augmentation methods such as resizing, rotation, flipping, blur, etc. help create a more diverse and generalized dataset, allowing the models to better handle real-world variations in images. Normalization is another crucial step in preprocessing, where pixel values are standardized to a common range, usually between 0 and 1, or based on model-specific mean and standard deviation values. This ensures that the model performs more efficiently during training.

2.6 Pre-trained Models and Transfer Learning

Transfer learning leverages pre-trained models, which are previously trained on large-scale dataset (such as ImageNet), to improve performance on new unseen data. Pre-trained models like these have already learned to detect general characteristics like edges, shapes, and textures, which can be applied to other tasks. By fine-tuning these models on a domain-specific dataset, it's possible to achieve faster convergence and better results. This approach is especially useful when the dataset size is small.

Using these models reduces both training time and computational costs, as the model has already learned valuable characteristics. Instead of starting from scratch, we can adapt the model to new tasks with minimal data. This makes transfer learning a highly efficient and effective method for tackling problems in various fields, from healthcare to autonomous driving, where large-scale labeled data may not be easily accessible.

2.7 Code Imports and Libraries

The following Python libraries and modules are used in this project:

PyTorch: The primary deep learning framework used to build, train, and evaluate the models.

torch: For tensor operations and model building.

torchvision.datasets: For loading image datasets.

torchvision.transforms: For preprocessing and augmenting images.

torch.utils.data.sampler: For sampling subsets of the data.

torch.nn: For building neural network layers.

NumPy: Used for handling numerical operations and arrays.

Pandas: Utilized for data manipulation and organization, especially useful when dealing with datasets.

Matplotlib: Used for visualizing the data, training process, and model evaluation results.

Datetime: Used for tracking time during training to assess model performance and time efficiency.

Scikit-learn: For additional evaluation metrics and performance analysis.

precision_score, recall_score, f1_score: To calculate precision, recall, and F1 score for model performance evaluation.

Seaborn: For enhanced data visualization, especially when visualizing confusion matrices or any other data patterns.

These libraries and imports provide the tools needed for deep learning model development, data manipulation, and performance evaluation. The combination of these libraries ensures an efficient and scalable pipeline for image classification tasks.

2.8 Evaluation Metrics

F1-Score: It is the harmonic mean of recall and precision, aiming to achieve a balance between them. It is useful when dealing with imbalanced datasets(where the classes are not represented equally), as it helps assess the model's ability to correctly identify positive cases (recall) and its ability to avoid false positives (precision).

Formula:

Top-1 Accuracy: This metric measures the percentage of times the model's top prediction (the class with the highest predicted probability) matches the true label. It is a standard metric in classification tasks, reflecting how often the model's first guess is correct.

Formula:

Precision: Precision is the inverse ratio of total positive predictions(true positive + false positive) to the true positive predictions. Precision is important when the cost of false positives is high, as it indicates how reliable the positive predictions are.

Formula:

Where: TP = True Positives

FP = False Positives

Recall: Recall is the inverse ratio of total positive predictions(true positive + false negatives) to the true positive predictions. Recall is important when the cost of false negatives is high, as it indicates how well the model captures all the relevant positive cases.

Formula:

Where: TP = True Positives

FN = False Negatives

These metrics are important for evaluating the overall performance of the model in terms of accuracy, robustness, and reliability. They help understand not only how well the model performs but also its ability to handle false positives and false negatives in different situations.

2.9 About the Dataset

The dataset used is the **MepcoTropicLeaf** dataset sourced from Kaggle, which is specifically designed for image classification tasks. Below are the key details about the dataset:

Number of Classes: It contains **50 classes**, each representing a different category or label for the images.

Number of Images: There are a total of **3,777 images** in the dataset, with varying resolutions and image quality. This diversity helps the model learn to generalize across different types of images and conditions.

Image Types: The images in this dataset represent various types of leaves and plants. Each class contains images that represent a specific leaf or plant type, making it a multiclass image classification problem.

Resolution: The images in the dataset have varied resolutions, which presents a challenge for training deep learning models. The preprocessing step includes resizing the images to a consistent size to ensure uniformity across the input data.

Labeling: Images are labeled with their respective class names, ensuring that the model learns to map visual characteristics to the correct category. The labels are used for training the models and for evaluating performance during validation and testing.

This dataset provides a rich source of images for training deep learning models, allowing researchers to evaluate and improve their models' performance in real-world image classification tasks, such as plant identification.

Chapter 3

Models

3.1 VGG16

VGG16 Pre-trained: This line loads the VGG16 model, pre-trained on ImageNet. The pretrained=True argument means the model is initialized with weights that were previously learned, that will help the model perform better, especially when fine-tuned on a smaller dataset like yours.

Freezing Early Layers: This part of the code freezes the convolutional layers (features) of VGG16, which means their weights/ parameters will not be updated during training. Freezing the early layers helps reduce overfitting, as these layers typically learn low-level characteristics such as edges, textures, etc. that are common across many tasks.

Custom Classifier: Layer-by-Layer Explanation:

First Linear Layer (nn.Linear(25088, 4096)):

Takes input of size 25,088 (flattened output from convolutional layers)

Produces output of size 4,096 (first fully-connected layer)

First Activation (nn.ReLU(inplace=True)):

Applies ReLU (Rectified Linear Unit) activation

inplace=True saves memory by modifying the input directly

First Dropout (nn.Dropout(0.6)):

Randomly sets 60% of neurons to zero during training

Helps prevent overfitting

Second Linear Layer (nn.Linear(4096, 4096)):

Another fully-connected layer maintaining the 4,096 dimension

Second Activation (nn.ReLU(inplace=True)):

Another ReLU activation

Second Dropout (nn.Dropout(0.6)):

Another 60% dropout (also reduced from 70%)

Final Linear Layer (nn.Linear(4096, 50)):

Output layer with 50 units (likely for 50-class classification)

3.2 MobileNetV2

MobileNetV2 Pre-trained: This line loads the MobileNetV2 model, again pre-trained on ImageNet. MobileNetV2 is designed for efficiency, making it suitable for devices with limited computational resources while still providing high performance.

Freezing All Layers: Initially, all layers of MobileNetV2 are frozen (i.e., requires_grad=False). This ensures that no weights are updated during the initial training phase. The model will only use the pre-trained features from ImageNet and will not update any parameters yet.

Unfreezing Specific Layers: After freezing all layers, the code selectively unfreezes the last 6 layers of MobileNetV2, specifically the inverted residual blocks, which will allow the model to learn new characteristics which will be specific to the dataset while still retaining the general features learned before training. You can adjust the number of layers to unfreeze depending on how much you want to fine-tune the model.

Modifying the Classifier: The classifier part of MobileNetV2 is customized:

The second linear layer (output layer) is modified to match your dataset's number of classes (targets_size), which is 50 in our case.

The dropout rate is adjusted from 0.5 (default) to 0.3 to control overfitting while ensuring enough information is passed through.

Custom Classifier (cont.): The classifier is further modified:

A new hidden layer with 512 units and ReLU activation is added.

The final layer outputs 50 classes for leaf classification.

3.3 ResNet18

ResNet18 Pre-trained: This loads the ResNet-18 model, which is a smaller variant of ResNet. It is pre-trained on ImageNet and is known for its residual connections, which help in training very deep networks without facing the vanishing gradient problem. ResNet18 is suitable for experiments requiring both efficiency and deep feature extraction.

3.4 Architecture of each model

Chapter 4

Data Preprocessing

Dataset preprocessing is an important step in ML, particularly in image recognition tasks. The goal of preprocessing is to prepare the raw data for input into a model, ensuring that it is in an optimal format for learning. While working with of images, preprocessing typically involves resizing, normalization, augmentation, and sometimes augmentation techniques which are carried in steps.

Resizing is often necessary as models require a consistent input size. Images in a dataset can vary in dimensions, so resizing them to a uniform size which will be 224x224 pixels for VGG16, ResNet, or MobileNet ensures that all images fit the input size expected by the network. Normalization is another important step, it involves changing the pixel values to a standard range, typically between 0 and 1, or subtracting the mean and dividing by the standard deviation (per-channel normalization). This ensures that the model's training process is more stable and efficient, as it prevents some features from dominating due to differences in scale.

Data Augmentation is used to increase the size of the dataset by applying random transformations such as rotations, flips, color adjustments, and cropping. This helps the model recognise better by making it less sensitive to specific image characteristics and by introducing variability in the data. Augmentation also reduces the risk of overfitting by providing the model with a more diverse set of training examples.

Lastly, shuffling the data helps to ensure that the model is exposed to a randomized order of training samples, further improving generalization. Preprocessing is often tailored to the specific needs of the model, such as VGG16, MobileNet, or ResNet, to ensure that the data is in the right form to achieve optimal results during training.

Data Preprocessing Applied in the Code

I have applied various data preprocessing steps, which can be broadly categorized into Data Augmentation, Data Transformation, and other important steps related to dataset splitting and model fine-tuning. Below is a detailed breakdown of the preprocessing techniques applied:

4.1 Data Augmentation

Data augmentation is used to increase the size of the dataset and add variability, which helps preventing overfitting. Augmentation techniques introduce random changes to the images, creating new variations of the original images that the model can learn from. The following augmentation techniques are used in the code:

Random Rotation:

transforms.RandomRotation(30): This transformation randomly rotates the image by an angle within a specified range (±30 degrees). This is especially useful when the dataset contains images captured from different viewpoints.

Random Resized Crop:

transforms.RandomResizedCrop(224, scale=(0.8, 1.0)): This technique involves randomly cropping a image and resizing it to a fixed size. The scale parameter specifies that the cropped region should be between 80% to 100% of the original image's area. This introduces variability by simulating different object sizes and positions in the image, making the model more robust to different spatial configurations.

Random Horizontal Flip:

transforms.RandomHorizontalFlip(): It randomly flips the image horizontally with a 50% chance(default value).

Color Jitter:

transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1): It randomly adjusts the above settings image. By varying these parameters, the model becomes more robust to changes in lighting conditions and color variations, which often occur in real-world scenarios. This type of augmentation is crucial for training models that need to generalize well across different environmental conditions.

Random Gaussian Blur:

transforms.RandomApply([transforms.GaussianBlur(kernel_size=5)], p=0.2): This technique applies a Gaussian blur to the image with a 20% probability. The blur simulates different levels of image quality or focus, allowing the model to become more resilient to blurry images, which may happen in real-world applications like low-quality cameras or motion blur.

4.2 Data Transformation

It involves modifying the dataset in a way that makes it suitable for input to the model. Unlike data augmentation, transformations are deterministic and apply consistent operations to the images.

These transformations ensure that input data is in the correct format for the model, which is critical for achieving high performance during training. Below are the transformation techniques applied in the code:

Resize:

transforms.Resize((224, 224)) Resizing images to a size is essential because neural networks, including models like VGG16, ResNet, and MobileNet, require the input data to have fixed dimensions. In your code, the images are resized to 224x224 pixels or 256 pixels before they are fed into the model. This resizing ensures that all images are compatible with the model's input layer, regardless of their original size.

Center Crop:

transforms.CenterCrop(224): This transformation crops the center of the image after resizing it. Cropping helps focus on the relevant areas of the image (often the center), which is particularly useful when the object of interest is located in the center of the image. This step ensures that the most important part of the image is preserved while eliminating irrelevant areas.

Normalization:

transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]): Normalization involves adjusting the pixel values so that they lie within a specified range. Typically, pixel values are scaled to a range of [0, 1] and then normalized by subtracting the mean and dividing by the standard deviation. The values used in the code are based on those of pre-trained models like VGG16, ResNet, and MobileNet, which were trained on ImageNet. This normalization step ensures that the model receives inputs in a consistent format, which improves the stability and performance of the model during training.

ToTensor:

transforms.ToTensor(): This transformation converts the image from a PIL Image (or NumPy array) into a PyTorch tensor, which is required for input into the model. Additionally, this transformation scales the pixel values to the range [0, 1] by dividing by 255 (as pixel values are originally in the range [0, 255]). This step is necessary to enable the model to process the image correctly.

4.3 Other Preprocessing

These steps don't directly fall under data augmentation or transformation but are still important for the overall training process:

Freezing Layers:

I have frozen the early layers of VGG16, ResNet, and MobileNetV2, meaning that their

weights will not be updated during training, this process is known as transfer learning, where the model uses the characteristics previously learned from a big dataset (such as ImageNet) and fine-tunes the later layers for the current task at hand. Freezing layers reduces the computational cost and training time, as only the later layers need to be trained.

Dataset Splitting:

The dataset is split into validation, training and test sets using a SubsetRandomSampler. The code ensures that the data is shuffled randomly to introduce variability in the dataset. The split is typically 70% for training, 20% for validation, and 10% for testing.

4.4 Normalization

The process of scaling the pixels of an image by subtracting the mean and dividing by the standard deviation of the dataset or, to a standard range which is typically between 0 and 1. This ensures that the model can learn more effectively by treating all features equally, improving convergence during training. It is crucial because it helps prevent issues like vanishing gradients, ensuring a stable learning process.

In my code, normalization is applied to the image data using transforms.Normalize(), where the pixel values are standardized based on the standard deviation values and mean values from the ImageNet dataset. Specifically, for each image, the Red, Green, and Blue (RGB) channels are normalized using the values [0.485, 0.456, 0.406] for the mean and [0.229, 0.224, 0.225] for the standard deviation. This ensures that the input to the model is in the same format as the data used to pre-train models like VGG16, ResNet, and MobileNet, allowing the model to learn effectively.

4.5 Batching

The process of dividing the dataset into subsets (batches) to be processed by the model during training, due to this the model can update its weights more frequently and efficiently, as training on the entire dataset at once may be computationally expensive. Using batches also helps with memory management and faster convergence by averaging the gradients over the batch instead of the entire dataset.

In my code, the batch size is set to 64 (batch_size = 64). This means that during each training iteration, 64 images will be processed simultaneously before updating the model's parameters. The DataLoader is used to load the dataset in batches. Also the images are shuffled randomly to make sure that the model does not learn from the order of the data. The batch size of 64 strikes a balance between efficient training and memory usage, helping the model converge faster while preventing memory overflow.

Chapter 5

Training and Evaluation

5.1 Hyperparameter Tuning:

The process of selecting and adjusting the hyperparameters(learning rate, weight decay, etc.) of a machine learning or a deep learning model to improve its performance on a given task. It includes the parameters that are set before the model training begins and cannot be known from the data. These parameters control the behavior of the learning algorithm and directly influence the model's performance.

Learning Rate:

For MobileNetV2's feature layers it is set to 1e-4(0.0001) and for the classifier layers it is set to 1e-3(0.001). The different learning rates allow the model to update the pre-trained feature layers more conservatively while allowing larger updates to the classifier layers that are being fine-tuned.

For ResNet18, the learning rate is set to 1e-4(0.0001), which is then used to optimize the model's weights.

For VGG16, the learning rate is set to 0.0001 for the Adam optimizer, using a lower learning rate to allow fine-tuning with more subtle adjustments.

Optimizer:

The Adam optimizer is used across all models (MobileNetV2, ResNet18, VGG16). Adam optimizer is a popular optimization algorithm because it adjusts the learning rate for each parameter individually, making it more efficient for training deep neural networks.

Weight Decay (L2 Regularization):

Weight decay is applied to all models as a regularization technique to penalize large weights, helping to reduce overfitting.

For MobileNetV2, it is set to 1e-5(0.00001).

For ResNet18, it is set to 1e-4(0.0001).

For VGG16, it is set to 5e-5(0.00005).

Batch Size:

The batch size is set to 64 for all three models (MobileNetV2, ResNet18, and VGG16). This defines how many images are processed in one iteration through the model. 64 is commonly seen as a good trade-off between memory efficiency and computational performance.

Epochs:

It defines how many times the entire dataset is passed through the model during training.

For MobileNetV2 and ResNet18, it is set to 10.

For VGG16, the number of epochs is set to 20. This is generally done to ensure the model has enough iterations to converge properly.

Freezing Layers (Transfer Learning):

For all pre-trained models (MobileNetV2, ResNet18, and VGG16), some layers are frozen to prevent their weights from being updated during training. Freezing the early layers helps to leverage the features learned on large datasets like ImageNet, while only fine-tuning the final layers for the specific task (leaf classification).

MobileNetV2 freezes all layers initially and then unfreezes the last 6 layers.

ResNet18 and VGG16 freeze the convolutional layers and fine-tune the fully connected layers.

Dropout:

It randomly deactivates a fraction of the neurons during training, which helps to reduce overfitting. It generalizes the model better by preventing it from relying too heavily on specific neurons.

For VGG16, dropout is applied with a rate of 0.6.

For MobileNetV2, dropout is set to 0.3 in the classifier part to prevent overfitting.

Adam Optimizer:

It is a gradient-based optimization algorithm that adapts the learning rate for each parameter individually, allowing the model to update weights more efficiently during training. Adam calculates mean and variance of the gradients, which helps in smoother and faster convergence.

MobileNetV2:

It is set to 0.001 for the classifier layers and 0.0001 for the feature layers.

The feature layers are updated conservatively with a lower learning rate to preserve the pre-trained features. The classifier layers are fine-tuned with a higher learning rate to allow more significant updates during training.

ResNet18:

The Optimizer is used with a learning rate of 1e-4.

The optimizer is applied uniformly across the model, adjusting all weights with the specified learning rate

VGG16:

The optimizer is used with a learning rate of 0.0001.

The optimizer applies smaller learning rate updates to fine-tune the pre-trained weights gradually.

Hidden Layers:

They are layers in a neural network that are not exposed directly to the output or input but are responsible for learning the complex patterns and representations of the input data. These layers transform the input into abstract features that are used to make predictions. The size and number of hidden layers impact the model's ability to learn from the data, with larger hidden layers generally capable of learning more complex features.

MobileNetV2:

The classifier is modified to include a hidden layer with 512 units.

A ReLU activation function is applied after the hidden layer to introduce non-linearity.

Dropout is applied in the classifier to prevent overfitting and improve generalization.

ResNet18:

The custom classifier consists of fully connected layers that transform the learned features into the final output.

The final layer which is the output layer consists of 50 units, i.e. the number of distinct classes of the leaf classification task.

VGG16:

The classifier is modified with two hidden layers, each containing 4096 units.

ReLU activation is applied to both hidden layers to introduce non-linearity.

Dropout= 0.6 is applied to prevent overfitting.

5.2 Training Process

In the code, I have implemented the training process for three pre-trained models: VGG16, ResNet18, and MobileNetV2. The main steps involved include data preprocessing, Hyperparameter tuning, defining loss functions and optimizers, and tracking training

performance. For each model, I applied transfer learning by freezing the convolutional layers and fine-tuning the classifier layers, by using different learning rates for the feature and classifier layers. I also implemented data augmentation and normalization to improve model performance. The models were trained using Adam optimizer with different learning rates and weight decay values, and their performance is captured using as accuracy, precision, recall, and F1-score were calculated. Finally, the models were evaluated on training, validation, and test datasets.

5.2.1. Training Process for VGG16

After loading the model and modifying the classifier in Chapter 3, the next steps for VGG16 involve:

Defining the Loss Function and Optimizer:

CrossEntropyLoss is used as the loss function for multi-class classification.

The Adam optimizer is used with a learning rate of 0.0001 and weight decay of 5e-5 to optimize the model:

Training the Model:

The model is trained over 20 epochs using the Gradient Descent. During each epoch:

The model processes batches of training data, calculates predictions, and computes the loss using criterion.

The optimizer updates the model weights with optimizer_vgg.step() and the gradients are computed using backpropagation (loss.backward()).

After every batch, the optimizer performs a weight update, minimizing the loss, and ensures the model improves after each iteration.

Validation: The model's performance is evaluated after each epoch on the validation set (using validation_loader_vgg). This helps to track how the model generalizes to new unseen data. Accuracy and loss for the validation set are recorded to monitor for overfitting.

Tracking Metrics: The training loss is tracked for each epoch, and the validation loss and validation accuracy are computed at the end of each epoch. This allows for monitoring the model’s progress and detecting issues such as overfitting (where the training loss decreases but validation loss increases).

5.2.2. Training Process for ResNet18

After loading the model and modifying the classifier in Chapter 3, the next steps for ResNet18 involve:

Defining the Loss Function and Optimizer:

CrossEntropyLoss is used for calculating the loss function for classification:

The Adam optimizer is used with a learning rate of 0.0001 to optimize the weights:

Training the Model:

Now the model is trained over 10 epochs using Gradient descent. The process is as follows:

For each epoch, batches from the training data are passed to the model.

The forward pass computes the output and the loss.

Backpropagation, calculates the gradients of the loss function with respect to the model's parameters (loss.backward()).

The optimizer updates the model's weights with optimizer_res.step(), reducing the loss in each step.

After processing each batch, the model's weights are updated, and the training loss is calculated for the batch.

Validation: After each pass through the dataset, the model is evaluated using the validation_loader_res to calculate the validation loss and accuracy. This step helps determine if the model is generalizing well or overfitting.

The validation and training loss are recorded at the end of each epoch to evaluate the model's convergence.

Model Evaluation: At the end of training, accuracy on the test set is evaluated to assess the final model performance.

5.2.3. Training Process for MobileNetV2

After loading the model and modifying the classifier in Chapter 3, the next steps for MobileNetV2 involve:

Defining the Loss Function and Optimizer:

The optimizer is used with different learning rates for the feature and classifier layers. A learning rate of 0.0001 is used for the feature layers, and 1e-3 for the classifier layers:

CrossEntropyLoss is the loss function used for multi-class classification.

Training the Model:

MobileNetV2 is trained over 10 epochs using train_loader_mob. The training process involves:

During each epoch, the model receives batches of data from the training set.

For each batch, the model computes predictions, calculates the loss, and the backpropagation step updates the gradients using loss.backward().

The optimizer (Adam) then changes the model's weights based on the gradients.

After every batch, the optimizer updates the model parameters, and training loss is computed to make sure if the model is learning.

Validation: After each pass through the dataset, the model's performance is evaluated on the validation set (validation_loader_mob), calculating validation loss and accuracy.

Performance Monitoring: The training loss and validation loss are recorded after each epoch, helping to monitor for any issues like overfitting. If the validation accuracy stops improving, hyperparameters such as the learning rate or batch size may be adjusted.

Test Set Evaluation: The model's performance is evaluated on the test set to obtain final accuracy and other metrics.

5.3 Monitoring& Evaluation

Monitoring and evaluation are critical steps in training deep learning models to ensure they are learning effectively and generalizing well to unseen data. Here is an overview of how monitoring and evaluation are typically carried out during and after training:

Training Loss Monitoring:

During training, training loss is computed for each batch and epoch. The loss quantifies how far the model's predictions are from the actual labels, and minimizing this loss is the primary goal of training.

A decreasing training loss tells us that the model is improving its performance by adjusting its weights to reduce the prediction error.

Tracking training loss helps to ensure that the model is learning properly and converging toward an optimal solution.

Validation Loss and Accuracy:

Validation loss and accuracy are computed after each epoch using the validation set. This tells us how well the model is generalizing to new unseen data.

Validation accuracy is particularly useful for monitoring if the model is overfitting. A large gap between training loss and validation loss can indicate overfitting, while a close match suggests the model is generalizing well.

Test Set Evaluation:

The model's performance is evaluated on the test set, which contains data the model has never seen before. This provides a final measure of how well the model will perform in a real-world scenario.

Precision, Top-1 Accuracy, F-1 score, and recall are calculated to assess the model's overall performance across different classes, especially when dealing with imbalanced datasets.

Metrics Calculation:

Recall, Precision and F1-score are some of important metrics for evaluating classification models, particularly in imbalanced datasets:

Precision: The ratio of true positive predictions among all positive predictions made by the model.

F1-score: The harmonic mean of recall and precision, providing a balanced metric when both precision and recall are important.

Recall: The proportion of true positive predictions among all actual positive instances.

Top-1 Accuracy score: It measures the percentage of times the model's top prediction matches the true label. It reflects how often the model correctly identifies the most probable class in multi-class classification tasks

Overfitting and Underfitting:

Overfitting- It occurs when the model performs good on the training data but poorly on the validation and test sets. This suggests the model has not learnt the general patterns in the dataset but learnt it.

Underfitting- It occurs when the model fails to perform well on both training and validation sets, indicating the model is too simple or has not been trained long enough.

Visualization:

Loss curves (plotting training loss and validation loss over epochs) and accuracy curves are helpful to visually monitor the model's learning progress. A smooth, decreasing training loss curve alongside an increasing validation accuracy curve indicates good model training.

If both loss curves stabilize and do not improve further, it may indicate that the model has converged, and training can be stopped.

By carefully tracking training loss, validation performance, and test metrics, we can ensure the model is not underfitting or overfitting and is ready for deployment. Regular evaluation helps make adjustments to the training process, such as tweaking hyperparameters or improving the data preprocessing pipeline to achieve better model performance.

Chapter 6

Results

6.1 Results: These metrics help in understanding the strengths and weaknesses of each model and provide a clear comparison of their performance.

1. Accuracy Scores

Top-1 Accuracy:

Top-1 accuracy measures the percentage of times the correct class was the model's top prediction. This metric is crucial in classification tasks where the correct class is expected to be the most probable prediction.

Precision:

Precision is calculated as the ratio of true positive predictions to the total number of positive predictions. It is used to evaluate the accuracy of the model in identifying positive instances.

Recall:

Recall is the ratio of true positive predictions to the total number of actual positive instances in the dataset. This metric is critical when minimizing false negatives is important.

2. Loss Metrics

Training Loss:

Training loss is calculated at each epoch to monitor how well the model is fitting the training data. A decreasing training loss indicates that the model is learning and improving its performance on the training data.

Test Loss:

Test loss is evaluated on the unseen test set, and it measures how well the model generalizes to data it has not seen before. The difference between training loss and test loss helps in assessing overfitting.

3. Computational Efficiency

Max GPU Usage:

The maximum GPU usage reflects the peak usage of GPU resources during model training. It indicates the highest demand placed on GPU resources at any point during the training process.

4. Inference Time

Inference Time:

Inference time measures the time taken by the model to make predictions on a batch of data. This metric is important for assessing the model's deployment efficiency and is crucial in real-time applications where speed is important.

6.2 Model's Evaluation

The following table summarizes the performance metrics for the three models: MobileNetV2, ResNet18, and VGG16.

6.3 Interpretation of Results:

Accuracy: ResNet18 achieves the highest accuracy across training, validation, and test sets, followed by MobileNetV2. VGG16, although it performs well on the training set, shows a drop in test accuracy, indicating some overfitting.

Precision, Recall, and F1-Score: ResNet18 has the highest precision, recall, and F1-score, demonstrating a balanced performance in detecting both positive and negative classes. MobileNetV2 also performs well, and VGG16 performs decently but slightly lags behind in these metrics.

Loss Metrics: ResNet18 has the lowest training and validation losses, indicating that it learns effectively. VGG16 shows a higher validation loss compared to training loss, suggesting possible overfitting.

Inference Time and Computational Efficiency: VGG16 is the most efficient in terms of inference time per batch, but it uses the most GPU memory, which might be a concern for resource-limited environments. MobileNetV2 strikes a good balance between memory usage and inference time, making it efficient for real-time applications. ResNet18, while highly accurate, uses more GPU resources.

Based on the results, I would recommend ResNet18 as the best model for the leaf classification task. ResNet18 achieves the highest test accuracy (95.15%) and validation accuracy (93.95%), indicating that it generalizes well to unseen data. Additionally, it performs exceptionally well in terms of precision, recall, and F1-score, showing a balanced ability to classify leaf images correctly while minimizing false positives and false negatives. The low training loss (0.0254) and validation loss (0.2232) further indicate that the model is learning effectively and not overfitting.

While MobileNetV2 offers the best computational efficiency with lower GPU memory usage (66.49 MB) and fast inference time (0.0130 seconds per batch), its test accuracy (92.06%) and validation accuracy (91.68%) are slightly lower than those of ResNet18. Therefore, while MobileNetV2 is a good option for resource-constrained environments, ResNet18 offers the best overall performance when accuracy is prioritized. ResNet18 provides the best trade-off between accuracy and computational efficiency, making it the optimal choice for the task.