Abbas Mustafa 20i2404

Advantages and limitations of using pre-trained models (such as EfficientNets, ViTs, and custom models) for image classification vary depending on the context and requirements of your specific task. Here's an overview:

Advantages of Pre-trained Models (EfficientNets and ViTs):

1. Higher Performance: Pre-trained models are trained on large and diverse datasets (e.g., ImageNet), which enables them to capture a wide range of features. This often results in better performance compared to models trained from scratch, especially when you have limited labeled data for your specific task.

2. Transfer Learning: You can leverage knowledge learned from one task (e.g., object recognition on ImageNet) and apply it to your task with fine-tuning. This can significantly reduce the amount of training data required and accelerate convergence.

3. State-of-the-Art: Models like EfficientNets and ViTs are often state-of-the-art in terms of accuracy on various computer vision tasks, making them a strong choice for many applications.

4. Community Support: Popular pre-trained models have extensive community support, which means you can find resources, tutorials, and pre-trained weights readily available.

Limitations of Pre-trained Models (EfficientNets and ViTs):

1. Resource Intensive: Pre-trained models are typically larger and require more computational resources (memory and processing power) for training and inference compared to smaller custom models.

2. Fine-tuning Complexity: Fine-tuning pre-trained models might require some effort in adapting the model to your specific task, including modifying the final fully connected layers, adjusting hyperparameters, and choosing the right learning rate.

3. Overfitting: Fine-tuning a large pre-trained model on a small dataset can lead to overfitting, so you may need to use techniques like data augmentation, regularization, or dropout to mitigate this issue.

Advantages of Custom Models:

Abbas Mustafa 20i2404

1. Flexibility: Custom models give you full control over the architecture, enabling you to design a model tailored to the specific characteristics of your dataset and task.

2. Efficiency: Smaller custom models can be more computationally efficient, making them suitable for deployment on resource-constrained devices or in real-time applications.

3. Interpretability: Custom models are often more interpretable than very deep pre-trained models, which may be essential for some applications.

Limitations of Custom Models:

1. Data Requirements: Training a custom model from scratch typically requires a larger amount of labelled data, which may not be available for all tasks.

2. Expertise: Designing an effective custom model often requires a good understanding of deep learning principles, architecture design, and hyperparameter tuning. It can be time-consuming and requires expertise.

3. Performance: Custom models may not achieve the same level of performance as state-of-the-art pre-trained models, especially for tasks where pre-trained models have excelled.

In summary, the choice between using pre-trained models like EfficientNets and ViTs or custom models depends on factors such as available data, computational resources, task requirements, and the level of expertise in model design and training. Pre-trained models can provide a powerful starting point and save time in many cases, but custom models offer flexibility and control when tailored solutions are needed. It's often beneficial to experiment with both approaches and choose the one that best suits your specific use case.

Abbas Mustafa 20i2404

Custom Model Design:

Data Preprocessing:

- Data is loaded and pre-processed using common transformations (e.g., resizing, normalization) as specified in the `transforms.Compose` functions for the training and test datasets.

Model Architecture:

- The custom Model architecture consists of two convolutional layers (`conv1` and `conv2`) with ReLU activation functions and max-pooling layers (`pool`).

- Two fully connected layers (`fc1` and `fc2`) follow the convolutional layers, with ReLU activation applied to `fc1`.

- The model architecture is designed to learn hierarchical features from input images.

Loss Function:

- The loss function used in the training process is the cross-entropy loss, appropriate for multi-class classification.

Optimizer:

- Stochastic Gradient Descent (SGD) with momentum is used as the optimizer to update the model's weights during training.

Training Process:

Data Splitting:

- The dataset is split into training and test sets. Typically, a validation set would be useful for hyperparameter tuning, but it's not explicitly shown in the code.

Model Initialization:

- The weights of the custom Model are initialized randomly.

Training Loop:

- The model is trained over a specified number of epochs (in this case, 10).

- For each epoch, the training dataset is iterated through mini-batches.

- The model is set to training mode (`model.train()`), and the optimizer's gradients are zeroed with `optimizer.zero_grad()`.

- Forward pass: Images are passed through the model to compute predictions.

- Loss computation: The cross-entropy loss is computed between model predictions and ground-truth labels.

- Backpropagation: Gradients are calculated with `loss.backward()`, and the optimizer updates the model's weights with `optimizer.step()`.

- Training loss is recorded for each epoch.

Results Analysis:

Evaluation:

- After training, the model is evaluated on a test dataset.

- Accuracy on the test dataset is calculated to measure the model's performance.

Possible Improvements:

1. Validation Set: Consider splitting the dataset into training, validation, and test sets. This allows you to monitor overfitting and tune hyperparameters effectively.

2. Data Augmentation: Experiment with more data augmentation techniques during preprocessing to improve model generalization. Augmentation can include rotation, shear, and colour jitter.

3. Regularization: Apply regularization techniques like dropout or weight decay to prevent overfitting, especially if you observe training accuracy significantly exceeding test accuracy.

4. Learning Rate Schedule: Implement a learning rate schedule that adjusts the learning rate during training. This can help stabilize training and potentially lead to better convergence.

5. Model Architecture: Experiment with different custom model architectures, including deeper or wider networks, or explore more advanced architectures such as residual connections or attention mechanisms (e.g., Transformer-based models like ViTs).

6. Ensemble Methods: Consider using ensemble methods to combine predictions from multiple models, which often leads to improved accuracy.

7. Class Imbalance: If your dataset has class imbalance, address it with techniques like class weighting or oversampling/under sampling strategies.

8. Hyperparameter Tuning: Fine-tune hyperparameters such as the learning rate, batch size, and the number of training epochs to find the optimal settings for your specific problem.

9. Pretrained Initialization: If you have access to a similar pretrained model, initialize your custom model with those weights before training. Fine-tuning pretrained weights can lead to faster convergence and better results.

10. Data Collection: Consider collecting more labelled data if possible, as a larger dataset can often lead to improved model performance.

11. Model Interpretability: Implement methods for model interpretability (e.g., Grad-CAM) to gain insights into which parts of the image contribute to the model's predictions.

Improvements can be task-specific, so it's essential to experiment and iterate to find the best combination of techniques and strategies for your specific image classification problem.