

Summer Internship Report On

Transfer Learning for Improved Image Classification

In partial fulfillment for the award of the degree of:

MASTER OF SCIENCE
(Data Science)



Submitted to:

Amity Institute of Integrative Sciences and Health,
Amity University, Haryana

Submitted by:

Navya Tyagi

Enrollment Number:

A525117724001

Under the Supervision of:

Dr. Andrew Lynn

Professor,

School of Computational and Integrative Sciences,
Jawaharlal Nehru University, New Delhi - 110067, India



जवाहरलाल नेहरू विश्वविद्यालय
JAWAHARLAL NEHRU UNIVERSITY

संगणकीय एवं समेकित विज्ञान संस्थान
School of Computational and Integrative Sciences
नई दिल्ली-110067, भारत / New Delhi-110067, India

Andrew M. Lynn
Professor

Tel.: +91-011-2670417
Email: andrew@jnu.ac.in

DATE: 15th July 2025

TO WHOMSOEVER IT MAY CONCERN

This is to certify that the dissertation report entitled "Transfer Learning for Improved Image Classification" submitted by Ms. Navya Tyagi, an Msc student in Data Science, is an original piece of work carried out by her. She has successfully completed her project under my guidance from 3rd June 2025 to 15th July 2025 at my laboratory at the School of Computational and Integrative Sciences, Jawaharlal Nehru University.

Ms. Navya has shown sincere interest in her work. She is hardworking, dedicated, and has demonstrated an exceptional ability to grasp new concepts quickly and apply them efficiently. Besides the topic in the dissertation, she also worked on developing and testing protocols in bioinformatics including systems administration, molecular docking, molecular dynamics simulations and implementing a basic neural network. We wish her all the best for her future endeavors.

Andrew M. Lynn, Ph. D.
Professor
School of Computational and Integrative Sciences
Jawaharlal Nehru University
New Delhi - 110067

Declaration

Amity Institute of Integrative Sciences and Health
Amity University, Haryana

Candidate's Declaration

I hereby certify that the work presented in this thesis “**Transfer Learning for Improved Image Classification**” by “**Navya Tyagi**” in partial fulfilment of requirements of the degree of **Master in Data Science** submitted to the *Amity Institute of Integrative Sciences and Health, Amity University Haryana*, is an original work record of my own carried out during a period of *June 2025* to *July 2025*, under the supervision of **Prof. Andrew Lynn**. The matter submitted in this thesis has not been submitted by me in any other University / Institute for the award of any Degree / Diploma.

Navya Tyagi

Enrollment Number: A525117724001

Amity University, Haryana





1% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.




Filtered from the Report

- Bibliography
- Quoted Text
- Cited Text
- Small Matches (less than 14 words)

Match Groups

-  **1 Not Cited or Quoted 1%**
Matches with neither in-text citation nor quotation marks
-  **0 Missing Quotations 0%**
Matches that are still very similar to source material
-  **0 Missing Citation 0%**
Matches that have quotation marks, but no in-text citation
-  **0 Cited and Quoted 0%**
Matches with in-text citation present, but no quotation marks

Top Sources

- 1%  Internet sources
- 0%  Publications
- 0%  Submitted works (Student Papers)

Integrity Flags

0 Integrity Flags for Review

No suspicious text manipulations found.

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

Acknowledgments

I would like to express my deepest gratitude to **Professor Andrew Lynn** at JNU for the unparalleled opportunity to work as an intern in his lab. His mentorship, patience, and profound knowledge created a truly rewarding and enjoyable learning experience.

My sincere thanks also go to the PhD students in the lab—especially Shailendra Singh Niboriya, Abhinay Yadav and Namrata Joshia—for their continuous support, valuable technical assistance, and willingness to share their expertise. Their encouragement, along with the camaraderie of my fellow interns, made the lab a vibrant and supportive place to learn and grow.

Finally, a truly genuine thank you to my family and friends. Your unwavering belief, constant encouragement, and occasional, much-needed distractions were my anchor throughout this demanding internship. Knowing I had your support made all the difference.

Table of Contents

1	Introduction	1
1.1	Introduction of the project	1
2	Model Architectures and Data Preparation	3
2.1	The Fashion-MNIST Dataset	3
2.2	Convolutional Neural Networks (CNNs)	3
2.3	Foundational Deep CNN Architectures	4
2.3.1	VGG16 and VGG19	4
2.3.2	ResNet50	4
2.3.3	DenseNet121	5
2.3.4	EfficientNetB0	5
3	Methodology	6
3.1	Overview of the Deep Learning Pipeline	6
3.1.1	Data Loading and Preprocessing	7
3.1.1.1	Data Loading:	7
3.1.1.2	Preprocessing Steps	7
3.1.2	Model Building and Transfer Learning	8
3.1.2.1	Architecture Selection	8
3.1.2.2	Transfer Learning Process	8
3.1.3	Training and Cross-Validation	8
3.1.3.1	Cross-Validation	8
3.1.3.2	Final Model Training	8
3.2	Evaluation and Prediction	8

3.2.0.1	Model Evaluation	8
3.2.0.2	Output Generation	9
3.2.0.3	Pipeline Orchestration	9
3.3	My Contributions and Extensions	9
3.3.1	Debugging and Modernization	9
3.3.2	Integration of EfficientNet	9
3.3.3	Generalization Experiments: Fashion-MNIST	10
4	Results	11
4.1	Codebase	11
4.2	Pipeline Execution	31
4.2.1	Linux execution	31
4.2.1.1	General Command Structure	31
4.2.2	Colab Notebook	32
4.3	Results	34
4.3.1	Accuracy	35
4.3.2	Architectural Efficiency	35
4.3.3	The Performance-Efficiency Trade-Off	35
5	Conclusion	36
5.1	Summary and discussion	36
5.2	Future prospects	37
5.2.1	Final validation on the original domain	37
5.2.2	Extended EfficientNet Scaling and Optimization	37
5.2.3	Integration of Advanced Training Strategies	37
	References (Bibliography)	39

List of Figures

3.1	Complete Modular Workflow of the Transfer Learning Pipeline. This diagram illustrates the five stages of the re-engineered deep learning pipeline.	7
4.1	Implementation of pipeline in Linux system-1	32
4.2	Implementation of pipeline in Linux system-2	32

List of Tables

4.1	Summarizes the key performance metrics for each of the five models . . .	34
-----	--	----

CHAPTER 1

Introduction

1.1 Introduction of the project

The COVID-19 pandemic emphasized the urgent need for fast, reliable, and scalable diagnostic tools in healthcare, particularly for interpreting medical images such as chest X-rays (CXRs). The interpretation of medical images, traditionally based on human expertise, is time-consuming, subjective, and susceptible to different observations. The rise of deep learning offers a revolutionary approach, with its capacity for automated feature extraction and robust pattern recognition, leading to development of diagnostic models that can assist or even outperform human experts in certain contexts [1]. Among the various approaches in deep learning, transfer learning emerges as a powerful strategy, especially in medical imaging, where creating large, annotated datasets is expensive and challenging.

Transfer learning uses models that are already trained using huge, diverse datasets (ImageNet) and such models are later modified for new, related tasks using smaller, domain-specific data. This methodology accelerates the model development, enhances generalization and prevents overfitting in data-constrained environments [1] [17]. In clinical settings, transfer learning directly addresses critical concerns such as data privacy, ethical standards, and the reusability of models across various clinical settings.

This thesis details the comprehensive implementation, debugging, and extension of a deep learning pipeline for image classification. Our initial focus was on COVID-19 detection from CXRs, followed by generalization to non-medical datasets. The work is built upon the original framework detailed in Chapter 2 of Tarun's PhD thesis [19], a pipeline

designed for modularity, automation, and adherence to standards such as the Checklist for AI in Medical Imaging (CLAIM) [3]. The original pipeline classified lung infections (SARS-CoV-2, pneumonia, normal) using the models like VGG16, VGG19, ResNet50 and DenseNet121 on a dataset of 9,990 CXR images. My contributions included extensive debugging to ensure compatibility with modern deep learning frameworks, integration of the state-of-the-art EfficientNet model, and rigorous testing on the Fashion-MNIST dataset to evaluate domain generalization.

This project aimed to demonstrate the pipeline's technical strength, flexibility, and scalability, highlighting its relevance to broader challenges in image classification, whether medical or non-medical. The following chapters provide an overview of the image classification and transfer learning; pipeline's architecture and workflow; my methodology and contributions and results; and a final conclusion.

Model Architectures and Data Preparation

2.1 The Fashion-MNIST Dataset

We selected the Fashion-MNIST dataset [5] to serve as a challenging, yet accessible, benchmark. The original MNIST dataset, comprising handwritten digits, is no longer sufficient to reveal performance differences among state-of-the-art modern architectures. In contrast, FASHION-MNIST introduces visual complexity with low-resolution images of various clothing, making it a practical test for generalization.

The dataset can be found at <https://www.kaggle.com/datasets/zalando-research/fashionmnist>. It is made up of 70,000 grayscale images, each one (28×28 pixels). Out of those, 60,000 go into the training set, and the remaining 10,000 are for testing. The data is distributed across 10 distinct clothing classes (e.g., T-shirt/top, Trouser, Coat). Its balance of low resolution and increased visual nuance makes it an ideal environment for comparing the efficiency and feature extraction capabilities of the five selected CNN models.

2.2 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are one of the most important deep learning networks used in image classification due to their ability to automatically and hierarchically learn spatial features [11, 6]. CNNs are employed in various fields such as computer vision [10], speech processing [8, 9], face recognition [16, 7], etc.

The CNN architecture consists of a number of layers:

1. **Convolutional Layers:** These layers apply learned filters (or kernels) across the input image to extract local features, such as edges, textures, and corners [23]. Each filter produces a feature map, and the depth of the network allows for the learning of increasingly abstract features in subsequent layers.
2. **Pooling Layers:** Typically inserted between convolutional layers, pooling (often Max Pooling) downsamples or subsamples the feature maps, reducing the dimensions and the overall number of parameters. This important step reduces the computational load, enhancing the network's efficiency and generalization ability.
3. **Fully Connected Layers:** Located near the output, these traditional neural network layers take the high-level feature maps derived from the earlier stages and use them to perform the final classification into the defined output classes.

2.3 Foundational Deep CNN Architectures

The models chosen for this study represent the foundational deep CNNs that established key standards in image recognition.

2.3.1 VGG16 and VGG19

The VGG or Visual Geometry Group models are characterized by their architectural uniformity and simplicity [12]. Both the VGG16 and VGG19 networks achieve great depth by relying exclusively on small 3×3 convolutional filters stacked in consistent sequences. Their primary difference is depth: VGG19 incorporates three additional convolutional layers compared to VGG16. This structure allows VGG19 to capture a greater complexity of features, though it comes with a corresponding increase in trainable parameters and computational demand. These models provide a valuable baseline due to their straightforward, deep design.

2.3.2 ResNet50

As we increase the number of layers in CNN-based architecture to reduce the error rate, we encounter a common problem known as the vanishing gradient problem, which causes the gradient to become zero or too large. The ResNet or Residual Network models fundamentally solved this problem. ResNet50 introduced the core concept: the Residual Block [13]. This block uses shortcut or skip connections to bypass one or more layers, enabling the network to learn underlying mappings. By forcing the layers to learn residual functions rather than the full layers, this architecture ensures that performance

can continue to improve with depth, making the stable training of the 50-layer network highly effective and without facing the vanishing gradient problem.

2.3.3 DenseNet121

DenseNet or Densely Connected Convolutional Networks streamlines the flow of information and gradients through a novel approach to connectivity. It addresses challenges of previous architecture such as feature reuse, parameter efficiency, and vanishing gradients. [14]. In DenseNet121, every layer is directly connected to all prior layers in its block, resulting in reuse of common features unlike traditional CNN where each layer is connected to its following layer. This dense connectivity structure improves feature propagation throughout the network, alleviates the vanishing gradient problem, and reduces the total number of parameters required by minimizing the redundancy of learned features.

2.3.4 EfficientNetB0

EfficientNetB0 is the baseline model of EfficientNet family, founded on the Compound Scaling methodology [15]. Traditional scaling methods arbitrarily adjust only one dimension (depth, width, or resolution). Compound Scaling, conversely, uniformly scales all three dimensions using a fixed set of scaling coefficients derived through a neural architecture search. This approach, combined with the use of mobile-size baseline components and depthwise separable convolutions, allows EfficientNetB0 to achieve classification accuracy comparable to much larger state-of-the-art models but with vastly fewer parameters and significantly reduced computational cost. Its inclusion provides a crucial comparison against the parameter-heavy foundational architectures.

3.1 Overview of the Deep Learning Pipeline

Our research framework focuses on a modular and automated deep learning pipeline, which was rigorously extended and modernized to facilitate this comparative study. Designed to ensure maximum transparency, reproducibility, and traceability, the pipeline organizes the entire experimental process into five integrated stages:

1. Data Loading and Preprocessing
2. Model Building and Transfer Learning
3. Training and Cross-Validation
4. Evaluation and Prediction
5. Pipeline Orchestration

Each module is enclosed in independent Python scripts, facilitating flexibility, extensibility, and ease of maintenance.

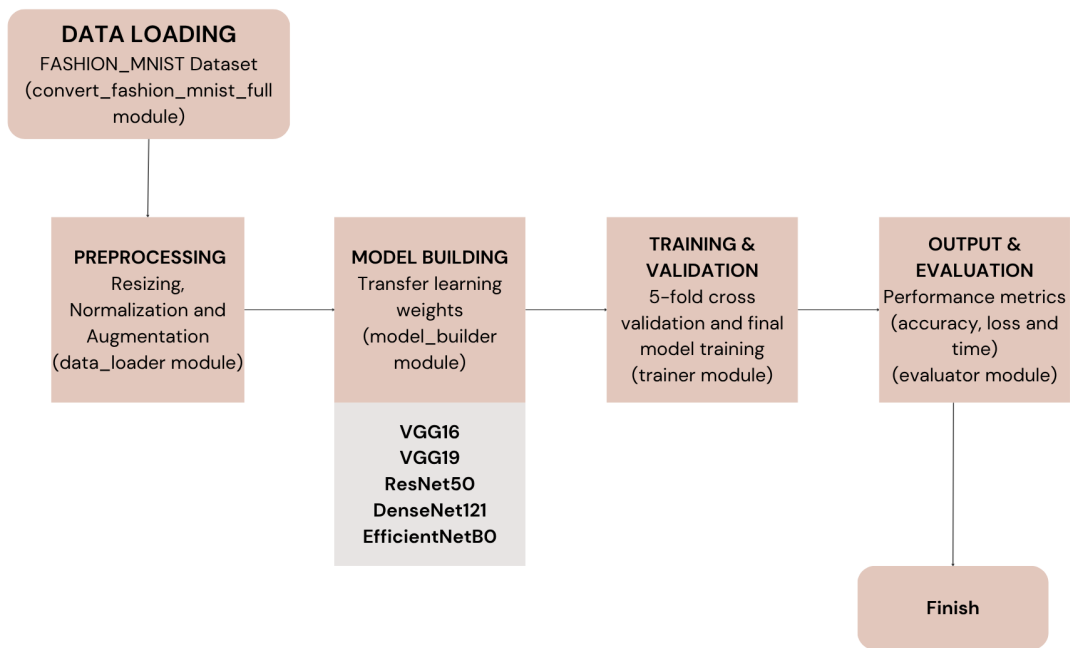


Figure 3.1: Complete Modular Workflow of the Transfer Learning Pipeline. This diagram illustrates the five stages of the re-engineered deep learning pipeline.

3.1.1 Data Loading and Preprocessing

3.1.1.1 Data Loading:

The `data_loader` module manages the preparation of the fashion-MNIST dataset. We input all 70,000 28 x 28 grayscale images and prepare them for multi-class (10-class) classification task.

3.1.1.2 Preprocessing Steps

1. **Resizing:** We resize images (e.g., to 224x224 pixels for VGG16) to match the fixed dimensions required by the selected CNN (VGG, ResNet, etc.).
2. **Normalization:** We scale pixel intensities from the original $[0, 255]$ to a $[0, 1]$ range, typically by dividing by 255.0
3. **Tensor Conversion:** We convert images into numerical arrays suitable for input to deep learning models.
4. **Label Encoding:** We encode the 10 target classes using `LabelBinarizer` for the multi-class problem.

5. **Data Augmentation:** To address overfitting and model generalization problems, we apply real-time transformations such as rotation, translation, shearing, zooming, and horizontal flipping during training [17].

3.1.2 Model Building and Transfer Learning

3.1.2.1 Architecture Selection

The `model_builder` module enables the selection and implementation of pre-trained CNN architectures.

3.1.2.2 Transfer Learning Process

1. **Feature Extraction:** The base convolutional layers of the pre-trained model (trained on ImageNet) are retained to extract generic visual features.
2. **Custom Classification Head:** The original ImageNet classification layers are replaced with new, task-specific layers—typically including `GlobalAveragePooling2D`, Dropout layer (e.g., 0.5 rate) for regularization, and Dense layers (e.g., 512 and 64 units) with ReLU activations. The final output layer employs softmax activation for multi-class classification.
3. **Layer Freezing and Fine-Tuning:** We initially freeze the pre-trained layers during training to preserve learned features, with only the custom head being trained. Selective unfreezing and fine-tuning may be performed later to optimize performance [21] [1].

3.1.3 Training and Cross-Validation

3.1.3.1 Cross-Validation

A 5-fold cross-validation strategy is applied on the training dataset, improving model robustness and reducing the risk of overfitting.

3.1.3.2 Final Model Training

Once cross-validation is done, we retrain the model on all 60,000 training images to get the final, deployable model.

3.2 Evaluation and Prediction

3.2.0.1 Model Evaluation

The `evaluator` module loads the trained model and performs inference on unseen, unlabeled test datasets (10,000 images).

3.2.0.2 Output Generation

Predictions (class labels and confidence scores) are saved in CSV files. Evaluation metrics such as accuracy, loss, and confusion matrices are generated where ground truth labels are available, enabling rigorous assessment of model performance.

3.2.0.3 Pipeline Orchestration

The `main.py` script coordinates the entire pipeline, parsing command-line arguments for data paths, architecture selection, and hyperparameter configuration. This modular implementation supports reproducibility and user customization.

3.3 My Contributions and Extensions

3.3.1 Debugging and Modernization

The original pipeline repository, developed in Python 3.6 and TensorFlow 2.2, required substantial updates to ensure compatibility with current deep learning frameworks. My debugging process included:

1. Code Analysis: Systematic identification and resolution of deprecated functions, syntax incompatibilities, and API changes in TensorFlow/Keras.
2. Dependency Management: Verification and updating of the Conda environment file to ensure all required packages were correctly specified and installed.
3. Reproducibility Enhancements: Documentation and scripting improvements to facilitate seamless setup and execution for future users and collaborators.

3.3.2 Integration of EfficientNet

Recognizing the superior efficiency and accuracy of EfficientNet architectures, I extended the `model_builder` module to support EfficientNet model(B0). This involved:

1. Weight Loading: Ensuring correct loading of ImageNet pre-trained weights and compatibility with custom classification heads.
2. Fine-Tuning Support: Fine-tuning the training workflow to manage selective freezing/unfreezing of EfficientNet layers.

3.3.3 Generalization Experiments: Fashion-MNIST

To test the pipeline’s versatility beyond medical imaging, I implemented the pipeline on the Fashion-MNIST dataset. This dataset presents a distinct challenge due to its non-natural, low-resolution images and balanced class distribution [17]. The Fashion-MNIST experiments assessed and ensured pipeline robustness, model generalization and established performance benchmarks.

CHAPTER 4

Results

4.1 Codebase

This section documents the complete source code and execution steps for the deep learning pipeline. The entire code can be accessed on github version control system: <https://github.com/Navya003/RadImageNet.git>

4.1: env.yml: This file specifies the package dependencies to recreate the execution system on Linux environments.

```
1 name: ImageAnalysisEnv
2 channels:
3   - conda-forge
4   - defaults
5 dependencies:
6   - python=3.9
7   - tensorflow-gpu=2.15.0
8   - py-opencv
9   - scikit-image
10  - scikit-learn
11  - numpy
12  - pandas
13  - matplotlib
14  - seaborn
```

```
15 - jupyter
16 - pip
```

4.2: convert_fashion_mnist_full.py: This file handles the task of downloading and saving the images to train and test sub-directories.

```
1 # import necessary libraries
2 import tensorflow as tf
3 import numpy as np
4 import os
5 import cv2 as cv
6
7 def
8     convert_fashion_mnist_to_dirs(output_base_dir="fashion_mnist_full_dataset"):
9         """
10         Downloads the Fashion-MNIST dataset and organizes it into
11         'train' and 'test' directories, with subdirectories for each class.
12         Each image is saved as a PNG file.
13         """
14         print(f"Starting Fashion-MNIST conversion to: {output_base_dir}")
15
16         # Loading Fashion-MNIST dataset
17         (X_train, y_train), (X_test, y_test) =
18             tf.keras.datasets.fashion_mnist.load_data()
19
20         # Defining class names for Fashion-MNIST
21         class_names = [
22             'T-shirt', 'Trouser', 'Pullover', 'Dress', 'Coat',
23             'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot'
24         ]
25
26         # Creating base output directories
27         train_dir = os.path.join(output_base_dir, 'train')
28         test_dir = os.path.join(output_base_dir, 'test')
29         os.makedirs(train_dir, exist_ok=True)
30         os.makedirs(test_dir, exist_ok=True)
31
32         print("Processing training data...")
```

```
31
32     # Process training data
33     for i, (image, label) in enumerate(zip(X_train, y_train)):
34         class_name = class_names[label]
35         class_output_dir = os.path.join(train_dir, class_name)
36         # Ensure class directory exists
37         os.makedirs(class_output_dir, exist_ok=True)
38
39         # Save image directly into the class directory
40         # Fashion-MNIST images are grayscale (28x28). Resized to 128x128 and
41         # convert to 3 channels for consistency.
42         image_resized = cv.resize(image, (128, 128))
43         image_bgr = cv.cvtColor(image_resized, cv.COLOR_GRAY2BGR) # Convert
44         # to 3 channels (BGR for OpenCV)
45
46         img_filename = os.path.join(class_output_dir,
47                                     f"train_img_{i:05d}.png")
48         cv.imwrite(img_filename, image_bgr)
49
50         if (i + 1) % 10000 == 0:
51             print(f" Saved {i + 1} training images.")
52
53     print(f"Finished saving {len(X_train)} training images.")
54
55     print("Processing test data...")
56     # Process test data
57     for i, (image, label) in enumerate(zip(X_test, y_test)):
58         class_name = class_names[label]
59         class_output_dir = os.path.join(test_dir, class_name)
60         os.makedirs(class_output_dir, exist_ok=True) # Ensure class
61         # directory exists
62
63         # Save image directly into the class directory
64         image_resized = cv.resize(image, (128, 128)) # MODIFIED: Resized to
65         # 128x128
66         image_bgr = cv.cvtColor(image_resized, cv.COLOR_GRAY2BGR) # Convert
67         # to 3 channels (BGR for OpenCV)
68
69         img_filename = os.path.join(class_output_dir,
```

```
        f"test_img_{i:05d}.png")
64     cv.imwrite(img_filename, image_bgr) # Save the image
65
66     if (i + 1) % 2000 == 0:
67         print(f" Saved {i + 1} test images.")
68
69     print(f"Finished saving {len(X_test)} test images.")
70     print(f"Fashion-MNIST conversion complete. Data saved to
        '{output_base_dir}'")
71
72 if __name__ == "__main__":
73     convert_fashion_mnist_to_dirs()
```

4.3: data_loader.py: This file manages the pre-processing of the images.

```
1 import os
2 import cv2 as cv
3 import numpy as np
4 from sklearn.preprocessing import LabelBinarizer
5 import tensorflow as tf # Import TensorFlow
6
7 class DataLoader:
8     def __init__(self, directory):
9         self.directory = directory
10
11     def load_train_data(self):
12         """
13         Load training data paths and labels, returning them as lists.
14         The actual image loading and preprocessing will be done later in the
15         tf.data pipeline.
16         """
17         image_paths = []
18         labels = []
19
20         categories_list = os.listdir(self.directory)
21         print(f"DataLoader (train): Found categories in directory:
            {categories_list}")
22         for category in categories_list:
23             path = os.path.join(self.directory, category)
```

```
23         if os.path.isdir(path):
24             print(f"DataLoader (train): Processing category: {category}
                at {path}")
25             for image_name in os.listdir(path):
26                 img_path = os.path.join(path, image_name)
27                 if os.path.isfile(img_path):
28                     # We only collect paths and labels here.
29                     # Preprocessing (cv2.imread, resize, normalize) will
                    be done in the tf.data map function.
30                     image_paths.append(img_path)
31                     labels.append(category)
32
33     lb = LabelBinarizer()
34     encoded_labels = lb.fit_transform(labels)
35
36     # Ensure all 10 classes are present in the LabelBinarizer's classes_
37     # This is a critical check for the 9-class issue.
38     if len(lb.classes_) != 10:
39         print(f"WARNING: LabelBinarizer detected {len(lb.classes_)}
                unique classes instead of 10 for training data.")
40         print(f"Detected classes: {lb.classes_}")
41         # You might want to raise an error or handle this more robustly
                if it's a common issue.
42
43     print(f"DataLoader (train): Total images found: {len(image_paths)}")
44     print(f"DataLoader (train): Total labels found: {len(labels)}")
45     print(f"DataLoader (train): Number of unique labels after encoding:
                {encoded_labels.shape[1]}")
46
47     # Return paths, encoded labels, and category names.
48     # These will be used to create a tf.data.Dataset in the trainer.
49     return image_paths, encoded_labels, lb.classes_.tolist()
50
51 def load_test_data(self):
52     """
53     Load test data paths and labels, returning them as lists.
54     The actual image loading and preprocessing will be done later in the
        tf.data pipeline.
55     """
```

```
56     image_paths = []
57     labels = []
58
59     categories_list = os.listdir(self.directory)
60     print(f"DataLoader (test): Found categories: {categories_list} in
        {self.directory}")
61     for category in categories_list:
62         path = os.path.join(self.directory, category)
63         if os.path.isdir(path):
64             print(f"DataLoader (test): Processing category: {category} at
                {path}")
65             for image_name in os.listdir(path):
66                 img_path = os.path.join(path, image_name)
67                 if os.path.isfile(img_path):
68                     image_paths.append(img_path)
69                     labels.append(category)
70
71     lb = LabelBinarizer()
72     encoded_labels = lb.fit_transform(labels)
73
74     if len(lb.classes_) != 10:
75         print(f"WARNING: LabelBinarizer detected {len(lb.classes_)}
                unique classes instead of 10 for test data.")
76         print(f"Detected classes: {lb.classes_}")
77
78     print(f"DataLoader (test): Total images found: {len(image_paths)}")
79     print(f"DataLoader (test): Total labels found: {len(labels)}")
80     print(f"DataLoader (test): Number of unique labels after encoding:
        {encoded_labels.shape[1]}")
81
82     # Return paths, encoded labels, and category names.
83     # These will be used to create a tf.data.Dataset in the evaluator.
84     return image_paths, encoded_labels, lb.classes_.tolist()
85
86     pass # This class now only has __init__ and load_data methods that
        return paths/labels.
```

4.4: `model_builder.py`: This file handles the initiation of the various CNN

models and manages the transfer learning task.

```

1 # importing necessary libraries
2 from tensorflow.keras.applications import VGG16, VGG19, ResNet50,
    DenseNet121, EfficientNetB0 # ADDED EfficientNetB0
3 from tensorflow.keras.layers import Dense, Dropout, GlobalAveragePooling2D
4 from tensorflow.keras.models import Model
5 from tensorflow.keras.optimizers import Adam
6 import tensorflow as tf
7
8 class ModelBuilder:
9     def __init__(self, base_model_name, input_shape, num_classes, lr,
10         epochs):
11         self.base_model_name = base_model_name
12         self.input_shape = input_shape # This will now be (128, 128, 3) from
13             pipeline.py
14         self.lr = lr
15         self.epochs = epochs
16         self.num_classes = num_classes
17
18     def build_model(self, num_classes):
19         base_model = None
20         preprocess_input_fn = None # Initialize preprocess_input_fn
21
22         if self.base_model_name == "VGG16":
23             base_model = VGG16(include_top=False, weights='imagenet',
24                 input_shape=self.input_shape)
25             preprocess_input_fn =
26                 tf.keras.applications.vgg16.preprocess_input # Get specific
27                     preprocess_input
28         elif self.base_model_name == "VGG19":
29             base_model = VGG19(include_top=False, weights='imagenet',
30                 input_shape=self.input_shape)
31             preprocess_input_fn = tf.keras.applications.vgg19.preprocess_input
32         elif self.base_model_name == "ResNet50":
33             base_model = ResNet50(include_top=False, weights='imagenet',
34                 input_shape=self.input_shape)
35             preprocess_input_fn =
36                 tf.keras.applications.resnet50.preprocess_input
37         elif self.base_model_name == "DenseNet121":

```

```

30         base_model = DenseNet121(include_top=False, weights='imagenet',
31                                   input_shape=self.input_shape)
32         preprocess_input_fn =
33             tf.keras.applications.densenet.preprocess_input
34     elif self.base_model_name == "EfficientNetB0": # ADDED
35         EfficientNetB0 case
36         # EfficientNet models typically expect 224x224 for B0, but can be
37         # scaled.
38         # The default input_shape for EfficientNetB0 is (224, 224, 3)
39         base_model = EfficientNetB0(include_top=False,
40                                     weights='imagenet', input_shape=self.input_shape)
41         preprocess_input_fn =
42             tf.keras.applications.efficientnet.preprocess_input
43     else:
44         raise ValueError(f"Unsupported base model:
45                             {self.base_model_name}")
46
47     if preprocess_input_fn is None:
48         # Fallback for models without specific preprocess_input, or
49         # generic one
50         print(f"Warning: No specific preprocess_input function found for
51               {self.base_model_name}. Using generic /255.0 normalization.")
52         preprocess_input_fn = lambda X: X / 255.0
53
54     # Building the custom top layers
55     x = base_model.output
56     X = GlobalAveragePooling2D()(X)
57     X = Dense(512, activation="relu")(X)
58     X = Dropout(0.5)(X)
59     X = Dense(64, activation="relu")(X)
60     X = Dropout(0.5)(X)
61     # Output layer
62     output = Dense(num_classes, activation="softmax")(X)
63     # Instantiate the final model
64     model = Model(inputs=base_model.input, outputs=output)
65
66     # Freezing the base model layers
67     for layer in base_model.layers:
68         layer.trainable = False

```

```
60
61     # Compile the model
62     opt = Adam(learning_rate=self.lr, decay=self.lr/self.epochs)
63     model.compile(optimizer=opt, loss='categorical_crossentropy',
64                  metrics=['accuracy'])
65     return model, preprocess_input_fn
```

4.5: trainer.py: This script implements model training and cross-validation.

```
1 # importing necessary libraries
2 from tensorflow.keras.callbacks import ModelCheckpoint, CSVLogger,
   EarlyStopping, ReduceLROnPlateau, TensorBoard
3 from sklearn.model_selection import KFold
4 import pandas as pd
5 import os
6 import numpy as np
7 import time
8 import tensorflow as tf
9
10 class ModelTrainer:
11     def __init__(self, model, image_paths_train, y_train, batch_size,
12                 epochs, output_dir, preprocess_input_fn):
13         self.model = model
14         self.image_paths_train = image_paths_train # Store image paths
15         self.y_train = y_train # Store one-hot encoded labels
16         self.batch_size = batch_size
17         self.epochs = epochs
18         self.output_dir = output_dir
19         self.preprocess_input_fn = preprocess_input_fn
20         self.acc_per_fold = []
21         self.loss_per_fold = []
22
23         self.models_dir = os.path.join(self.output_dir, 'models')
24         self.tensorboard_dir = os.path.join(self.output_dir,
25                                             'tensorboard_logs')
26         self.csv_logs_dir = os.path.join(self.output_dir, 'csv_logs')
27         self.create_subdirectories()
```

```

27     def create_subdirectories(self):
28         os.makedirs(self.models_dir, exist_ok=True)
29         os.makedirs(self.tensorboard_dir, exist_ok=True)
30         os.makedirs(self.csv_logs_dir, exist_ok=True)
31
32     # Helper function to load, preprocess, and augment images for
33     # tf.data.Dataset
34     def _parse_image_function(self, filepath, label):
35         # Load the raw image bytes
36         imge = tf.io.read_file(filepath)
37         # Decode to tensor
38         imge = tf.image.decode_image(imge, channels=3,
39                                     expand_animations=False) # Ensure 3 channels
40         # Resize
41         imge = tf.image.resize(imge, (128, 128)) # Changed to 128x128
42         # Apply model-specific preprocessing
43         imge = self.preprocess_input_fn(imge)
44         return imge, label
45
46     # Helper function for data augmentation
47     def _augment_image(self, image, label):
48         image = tf.image.random_flip_left_right(image)
49         return image, label
50
51     def get_callbacks(self, fold, use_validation_callbacks=True):
52         callbacks_list = []
53
54         csv_logger = CSVLogger(os.path.join(self.csv_logs_dir,
55                                             f'training_log_{fold}.csv'))
56         tensor_board =
57             TensorBoard(log_dir=os.path.join(self.tensorboard_dir,
58                                             f'tensorboard_logs_{fold}'))
59
60         callbacks_list.append(csv_logger)
61         callbacks_list.append(tensor_board)
62
63         if use_validation_callbacks:
64             # MODIFIED: Changed filepath extension to .weights.h5
65             mc_path = os.path.join(self.models_dir,

```

```

        f'model_{fold}.weights.h5')
61     model_checkpoint = ModelCheckpoint(filepath=mc_path,
        monitor='val_loss', mode='min', save_best_only=True,
        verbose=1, save_weights_only=True)
62     # added EarlyStopping and ReduceLRonPlateau for better training
        control
63     early_stop = EarlyStopping(monitor='val_loss', mode='min',
        patience=10, restore_best_weights=True, verbose=1)
64     reduce_lr = ReduceLRonPlateau(monitor='val_loss', factor=0.1,
        patience=5, verbose=1)
65     # adding to callbacks list
66     callbacks_list.append(model_checkpoint)
67     callbacks_list.append(early_stop)
68     callbacks_list.append(reduce_lr)
69     return callbacks_list
70
71     def cross_validate(self, n_folds=3):
72         acc_per_fold, loss_per_fold, val_acc_per_fold, val_loss_per_fold,
        time_per_fold = [], [], [], [], []
73         kf = KFold(n_splits=n_folds, shuffle=True, random_state=0)
74         total_time = 0
75
76         # Convert image_paths_train and y_train to TensorFlow Tensors for
        tf.data
77         image_paths_tensor = tf.constant(self.image_paths_train)
78         y_train_tensor = tf.constant(self.y_train, dtype=tf.float32) #
        Ensure correct dtype for labels
79
80         for fold_no, (tr_idx, val_idx) in
        enumerate(kf.split(self.image_paths_train, self.y_train), 1): #
        Use image_paths_train for splitting
81             print(f'Training fold {fold_no}...')
82
83             start_time = time.time()
84
85             # Create train and validation datasets using tf.data.Dataset
86             train_filepaths = tf.gather(image_paths_tensor, tr_idx)
87             train_labels = tf.gather(y_train_tensor, tr_idx)
88             val_filepaths = tf.gather(image_paths_tensor, val_idx)

```

```

89     val_labels = tf.gather(y_train_tensor, val_idx)
90
91     # Build dataset for training
92     training_dataset =
93         tf.data.Dataset.from_tensor_slices((train_filepaths,
94                                             train_labels))
95     training_dataset =
96         training_dataset.map(self._parse_image_function,
97                             num_parallel_calls=tf.data.AUTOTUNE)
98     training_dataset = training_dataset.map(self._augment_image,
99                                             num_parallel_calls=tf.data.AUTOTUNE) # Apply augmentation
100    training_dataset =
101        training_dataset.shuffle(buffer_size=1000).batch(self.batch_size).prefetch(
102
103    # Build dataset for validation
104    valn_dataset = tf.data.Dataset.from_tensor_slices((val_filepaths,
105                                                       val_labels))
106    valn_dataset = valn_dataset.map(self._parse_image_function,
107                                    num_parallel_calls=tf.data.AUTOTUNE)
108    valn_dataset =
109        valn_dataset.batch(self.batch_size).prefetch(tf.data.AUTOTUNE)
110    # No augmentation for validation
111
112    history = self.model.fit(training_dataset,
113                             validation_data=valn_dataset,
114                             epochs=self.epochs,
115                             verbose=1,
116                             callbacks=self.get_callbacks(fold_no,
117                                                         use_validation_callbacks=True))
118
119    acc_per_fold.append(history.history['accuracy'][-1] * 100)
120    loss_per_fold.append(history.history['loss'][-1])
121    val_acc_per_fold.append(history.history['val_accuracy'][-1]*100)
122    val_loss_per_fold.append(history.history['val_loss'][-1])
123
124    end_time = time.time()
125    fold_time = end_time - start_time
126    total_time += fold_time
127    time_per_fold.append(fold_time)

```

```
116
117     print(f"Time taken for fold {fold_no}: {fold_time:.3f} seconds")
118     print(f"Total training time for {n_folds} folds: {total_time:.3f}
119           seconds")
120
121     self.save_cv_results(acc_per_fold, loss_per_fold, val_acc_per_fold,
122                          val_loss_per_fold, time_per_fold, total_time)
123     return np.mean(acc_per_fold), np.mean(loss_per_fold), time_per_fold
124
125 def save_cv_results(self, acc_per_fold, loss_per_fold, val_acc_per_fold,
126                    val_loss_per_fold, time_per_fold, total_time):
127     results_df = pd.DataFrame({
128         'Fold': [i + 1 for i in range(len(acc_per_fold))],
129         'Training Accuracy': acc_per_fold,
130         'Training Loss': loss_per_fold,
131         'Validation Accuracy': val_acc_per_fold,
132         'Validation Loss': val_loss_per_fold,
133         'Mean_acc': np.mean(acc_per_fold),
134         'Mean_loss': np.mean(loss_per_fold),
135         'Time (seconds)': time_per_fold
136     })
137     results_df.to_csv(os.path.join(self.output_dir, 'cv_results.csv'),
138                      index=False)
139
140     with open(os.path.join(self.output_dir, 'training_time.txt'), 'w')
141           as f:
142         f.write(f'Total training time: {total_time:.3f} seconds\n')
143
144 def train_on_entire_dataset(self):
145     print("Training the model on the entire dataset...")
146
147     callbacks = self.get_callbacks(fold='entire_dataset',
148                                    use_validation_callbacks=False)
149
150     # Build tf.data.Dataset for the entire training set
151     image_paths_tensor = tf.constant(self.image_paths_train)
152     y_train_tensor = tf.constant(self.y_train, dtype=tf.float32)
153
154     train_dataset_full =
```

```

        tf.data.Dataset.from_tensor_slices((image_paths_tensor,
            y_train_tensor))
149 train_dataset_full =
        train_dataset_full.map(self._parse_image_function,
            num_parallel_calls=tf.data.AUTOTUNE)
150 train_dataset_full = train_dataset_full.map(self._augment_image,
            num_parallel_calls=tf.data.AUTOTUNE) # Apply augmentation
151 train_dataset_full =
        train_dataset_full.shuffle(buffer_size=1000).batch(self.batch_size).prefetch(
152
153 start_time = time.time()
154
155 history = self.model.fit(train_dataset_full,
156                           epochs=self.epochs,
157                           verbose=1, callbacks=callbacks)
158
159 end_time = time.time()
160 total_time = end_time - start_time
161
162 final_acc = history.history['accuracy'][-1] * 100
163 final_loss = history.history['loss'][-1]
164
165 print(f"Time taken to train on the entire dataset: {total_time:.3f}
        seconds")
166 print(f"Final training accuracy on the entire dataset:
        {final_acc:.3f}%")
167 print(f"Final training loss on the entire dataset: {final_loss:.3f}")
168
169 # MODIFIED: Changed final model path to .weights.h5
170 model_path = os.path.join(self.models_dir, 'final_model.weights.h5')
171 self.model.save_weights(model_path)
172 print(f"Final model weights saved at: {model_path}")
173
174 self.save_entire_dataset_results(final_acc, final_loss, total_time)
175
176 def save_entire_dataset_results(self, final_acc, final_loss, total_time):
177     results_df = pd.DataFrame({"Final Accuracy (%)": [final_acc], "Final
        Loss": [final_loss], "Training Time (seconds)": [total_time]})
178

```

```

179         results_df.to_csv(os.path.join(self.output_dir,
180                                     'entire_dataset_results.csv'), index=False)
181     with open(os.path.join(self.output_dir, 'entire_training_time.txt'),
182               'w') as f:
183         f.write(f"Final Accuracy: {final_acc:.3f}%\n")
184         f.write(f"Final Loss: {final_loss:.3f}\n")
185         f.write(f"Total training time: {total_time:.3f} seconds\n")

```

4.6: evaluator.py: This script calculates and reports key performance indicators.

```

1  # import necessary libraries
2  import os
3  import numpy as np
4  import pandas as pd
5  from sklearn.metrics import classification_report, confusion_matrix
6  import matplotlib.pyplot as plt
7  import seaborn as sns
8  import tensorflow as tf
9  from tensorflow.keras.models import load_model
10
11 class ModelEvaluator:
12     def __init__(self, output_dir, image_paths_test, y_test, class_names,
13                 x_test_original, preprocess_input_fn, model_builder):
14         self.output_dir = output_dir
15         self.image_paths_test = image_paths_test
16         self.y_test = y_test
17         self.class_names = class_names
18         self.x_test_original = x_test_original
19         self.preprocess_input_fn = preprocess_input_fn
20         self.model_builder = model_builder
21         self.evaln_results_dir = os.path.join(self.output_dir,
22                                               'evaluation_results')
23         os.makedirs(self.evaln_results_dir, exist_ok=True)
24
25     def _parse_image_function_eval(self, filepath, label):
26         img = tf.io.read_file(filepath)
27         img = tf.image.decode_image(img, channels=3, expand_animations=False)

```

```

26     img = tf.image.resize(img, (128, 128)) # MODIFIED: Changed to 128x128
27     img = self.preprocess_input_fn(img)
28     return img, label
29
30 def evaluate(self):
31     # Build the model architecture (without training it) to load weights
32     # Set input shape for evaluation model
33     input_shape = (128, 128, 3)
34     num_classes = self.y_test.shape[1]
35
36     # Build an untrained model with the correct architecture to load
37     # weights into
38     # We pass dummy lr and epochs, as they are not used for just
39     # building the model architecture for evaluation
40     model, _ = self.model_builder.build_model(num_classes)
41
42     # Load the best weights saved during training
43     model_path = os.path.join(self.output_dir, 'models',
44                               'final_model.weights.h5')
45     if not os.path.exists(model_path):
46         print(f"Error: Final model weights not found at {model_path}.
47               Cannot evaluate.")
48         return
49
50     model.load_weights(model_path)
51     print(f"Loaded model weights from: {model_path}")
52
53     # Create tf.data.Dataset for test data
54     test_filepaths = tf.constant(self.image_paths_test)
55     test_labels = tf.constant(self.y_test, dtype=tf.float32)
56
57     testing_dataset =
58         tf.data.Dataset.from_tensor_slices((test_filepaths, test_labels))
59     testing_dataset =
60         testing_dataset.map(self._parse_image_function_eval,
61                             num_parallel_calls=tf.data.AUTOTUNE)
62     testing_dataset =
63         testing_dataset.batch(32).prefetch(tf.data.AUTOTUNE) # Using a
64         default batch size for evaluation

```

```
56
57     print("Predicting on test data...")
58     predictions = model.predict(testing_dataset)
59     y_pred = np.argmax(predictions, axis=1)
60     y_true = np.argmax(self.y_test, axis=1)
61
62     # Generating classification report
63     report = classification_report(y_true, y_pred,
64                                   target_names=self.class_names)
65     print("\nClassification Report:")
66     print(report)
67
68     report_path = os.path.join(self.eval_results_dir,
69                                'classification_report.txt')
69     with open(report_path, 'w') as f:
70         f.write(report)
71     print(f"Classification report saved to {report_path}")
72
73     # Generating confusion matrix
74     confm = confusion_matrix(y_true, y_pred)
75     print("\nConfusion Matrix:")
76     print(confm)
77
78     plt.figure(figsize=(10, 8))
79     sns.heatmap(confm, annot=True, fmt='d', cmap='Blues',
80                xticklabels=self.class_names, yticklabels=self.class_names)
81     plt.xlabel('Predicted Label')
82     plt.ylabel('True Label')
83     plt.title('Confusion Matrix')
84     cm_path = os.path.join(self.eval_results_dir, 'confusion_matrix.png')
85     plt.savefig(cm_path)
86     print(f"Confusion matrix saved to {cm_path}")
87     plt.close()
88
89     # Saving raw predictions
90     predictions_df = pd.DataFrame(predictions, columns=[f'prob_{name}'
91                                                    for name in self.class_names])
92     predictions_df['true_label'] = [self.class_names[i] for i in y_true]
93     predictions_df['predicted_label'] = [self.class_names[i] for i in
```

```
        y_pred]
91 predictions_df.to_csv(os.path.join(self.eval_results_dir,
        'predictions.csv'), index=False)
92 print(f"Predictions saved to {os.path.join(self.eval_results_dir,
        'predictions.csv')}")
```

4.7: pipeline.py: This file defines and manages the overall workflow sequence by defining a step-by-step execution path.

```
1 from data_loader import DataLoader
2 from model_builder import ModelBuilder
3 from trainer import ModelTrainer
4 from evaluator import ModelEvaluator
5 import os
6 from tensorflow.keras.models import load_model
7
8 class Pipeline:
9     def __init__(self, model_name, train_dir, test_dir, output_dir, lr,
10         batch_size, epochs):
11         self.model_name = model_name
12         self.train_dir = train_dir
13         self.test_dir = test_dir
14         self.output_dir = output_dir
15         self.lr = lr
16         self.batch_size = batch_size
17         self.epochs = epochs
18         os.makedirs(self.output_dir, exist_ok=True)
19
20     def run(self):
21         # Load training data paths and labels
22         print("Loading training data paths and labels...")
23         data_load_train = DataLoader(self.train_dir)
24         # MODIFIED: x_train will now be image_paths_train
25         image_paths_train, y_train, class_names =
26             data_load_train.load_train_data()
27         print(f'Training data: {len(image_paths_train)} images,
28             {y_train.shape[1]} classes')
```

```

27     # Build model and get its specific preprocessing function
28     print("Building Model ....")
29     num_classes = y_train.shape[1]
30     model_builder = ModelBuilder(self.model_name, (128, 128, 3),
        num_classes, self.lr, self.epochs) # MODIFIED: input_shape to
        (128, 128, 3)
31     model, preprocess_input_fn = model_builder.build_model(num_classes)
32
33     # Train and cross validate model
34     print("Training and cross-validating model...")
35     # MODIFIED: Pass image_paths_train instead of x_train to ModelTrainer
36     model_trainer = ModelTrainer(model, image_paths_train, y_train,
        self.batch_size, self.epochs, self.output_dir,
        preprocess_input_fn)
37     mean_accuracy, mean_loss, time_per_fold =
        model_trainer.cross_validate()
38     model_trainer.train_on_entire_dataset()
39     print(f"Cross-validation results - Mean Accuracy:
        {mean_accuracy:.3}%, Mean Loss: {mean_loss:.3f}")
40
41     # Load test data paths and labels
42     print("Loading test data paths and labels....")
43     data_load_test = DataLoader(self.test_dir)
44     # MODIFIED: x_test will now be image_paths_test
45     image_paths_test, y_test, test_categories =
        data_load_test.load_test_data()
46     print(f'Test data: {len(image_paths_test)} images, {y_test.shape[1]}
        classes')
47
48     # Evaluate model on test data
49     print("Evaluating model on unseen test data...")
50     # MODIFIED: Pass image_paths_test instead of x_test to ModelEvaluator
51     model_evaluator = ModelEvaluator(self.output_dir, image_paths_test,
        y_test, test_categories, image_paths_test, preprocess_input_fn,
        model_builder)
52     model_evaluator.evaluate()

```

4.8: main.py: This is the application entry point and handles the command

line arguments parsing.

```
1 import argparse
2 from pipeline import Pipeline
3 import tensorflow as tf
4
5 print(tf.__version__)
6
7 if __name__ == "__main__":
8     parser = argparse.ArgumentParser(description = 'Run Image Classification
9         Pipeline')
10
11     # MODIFIED: Added 'EfficientNetB0' to choices
12     parser.add_argument('--model_name', type=str, required=True,
13         choices=['VGG16', 'VGG19', 'DenseNet121', 'ResNet50',
14         'EfficientNetB0'], help="Pre-trained model to use")
15     parser.add_argument('--train_dir', type=str, required=True, help="Path
16         to the training data directory")
17     parser.add_argument('--test_dir', type=str, required=True, help="Path to
18         the test data directory")
19     parser.add_argument('--output_dir', type=str, required=True, help="Path
20         to the output directory for saving models and results")
21     parser.add_argument('--lr', type=float, help="Learning rate")
22     parser.add_argument('--batch_size', type=int, help="Batch size for
23         training")
24     parser.add_argument('--epochs', type=int, help="Number of epochs for
25         training")
26
27     args = parser.parse_args()
28
29     pipeline = Pipeline(
30         model_name=args.model_name,
31         train_dir=args.train_dir,
32         test_dir=args.test_dir,
33         output_dir=args.output_dir,
34         lr=args.lr,
35         batch_size=args.batch_size,
36         epochs=args.epochs
37     )
```

```
31 pipeline.run()
```

4.2 Pipeline Execution

The pipeline can be executed through two ways, a dedicated linux machine with conda dependency management and an interactive cloud environment like Google Colab (or jupyter) notebook.

4.2.1 Linux execution

The conda dependencies can be installed using `env.yml` file to set up the environment. Then data is organized into test and train sub-directories which we are doing here using `convert_fashion_mnist_full.py` file. Finally, the pipeline can be executed using the `main.py` script with the appropriate arguments.

4.2.1.1 General Command Structure

```
python main.py \  
--model_name [MODEL_NAME] \  
--train_dir [PATH_TO_TRAINING_DATA] \  
--test_dir [PATH_TO_TEST_DATA] \  
--output_dir [PATH_TO_SAVE_RESULTS] \  
--lr [LEARNING_RATE] \  
--batch_size [BATCH_SIZE] \  
--epochs [NUM_EPOCHS]
```

```

container@jarvis:~/navya/RadiImageNet
(container@jarvis:~/navya/RadiImageNet)$ nvidia-smi
Fri Jul 11 11:41:38 2025

+-----+
| NVIDIA-SMI 525.85.05 | Driver Version: 525.85.05 | CUDA Version: 12.0 |
+-----+-----+
| GPU Name | Persistence-M | Bus-Id | Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap | Memory-Usage | GPU-Util  Compute M. |
|-----+-----+-----+-----+-----+
| 0  N/A  34C    P8      8W / 151W | 100MiB / 8192MiB |      0%      Default |
+-----+-----+-----+-----+-----+

Processes:
+-----+
| GPU | GI | CI | PID | Type | Process name | GPU Memory |
|-----+-----+
| 0  N/A | N/A | 3115 | G | /usr/libexec/Xorg | 75MiB |
| 0  N/A | N/A | 3243 | G | /usr/bin/gnome-shell | 22MiB |
+-----+

(container@jarvis:~/navya/RadiImageNet)$ ls
data_loader.py  convert_fashion_mnist_full.py  convert_fashion_mnist.py  env.yml  evaluator.py  main.py  model_builder.py  pipeline.py  __pycache__  README.md  trainer.py
(container@jarvis:~/navya/RadiImageNet)$ python convert_fashion_mnist_full.py
2025-07-11 11:41:53.931093: E external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:9261] Unable to register cuDNN factory: Attempting to register factory for plugin cuDNN when one has already been reg
istered
2025-07-11 11:41:53.931134: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:687] Unable to register cuFFT factory: Attempting to register factory for plugin cuFFT when one has already been reg
istered
2025-07-11 11:41:53.932062: E external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1515] Unable to register cuBLAS factory: Attempting to register factory for plugin cuBLAS when one has already been reg
istered
2025-07-11 11:41:53.937180: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: SSE4.1 SSE4.2 AVX AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
Starting Fashion-MNIST conversion to: fashion_mnist_full_dataset
Processing training data...
Saved 10000 training images.
Saved 20000 training images.
Saved 30000 training images.
Saved 40000 training images.
Saved 50000 training images.
Saved 60000 training images.
Finished saving 60000 training images.
Processing test data...
Saved 2000 test images.
Saved 4000 test images.
Saved 6000 test images.
Saved 8000 test images.
Saved 10000 test images.
Finished saving 10000 test images.
Fashion-MNIST conversion complete. Data saved to 'fashion_mnist_full_dataset'
(container@jarvis:~/navya/RadiImageNet)$ python main.py --model_name EfficientNetB0 --train_dir ./navya/RadiImageNet/fashion_mnist_full_dataset/train --test_dir ./navya/RadiImageNet/fashion_mnist
_full_dataset/test --output_dir ./eff_output_small --lr 0.0001 --batch_size 8 --epochs 5
2025-07-11 11:42:51.978495: E external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:9261] Unable to register cuDNN factory: Attempting to register factory for plugin cuDNN when one has already been reg
istered

```

Figure 4.1: Implementation of pipeline in Linux system-1

```

(container@jarvis:~/navya/RadiImageNet)$ python main.py
DataLoader (test): Processing category: Ankle boot at /home/container/navya/RadiImageNet/fashion_mnist_full_dataset/test/ankle boot
DataLoader (test): Processing category: Pullover at /home/container/navya/RadiImageNet/fashion_mnist_full_dataset/test/Pullover
DataLoader (test): Processing category: Trouser at /home/container/navya/RadiImageNet/fashion_mnist_full_dataset/test/Trouser
DataLoader (test): Processing category: Shirt at /home/container/navya/RadiImageNet/fashion_mnist_full_dataset/test/Shirt
DataLoader (test): Processing category: Coat at /home/container/navya/RadiImageNet/fashion_mnist_full_dataset/test/Coat
DataLoader (test): Processing category: Sandal at /home/container/navya/RadiImageNet/fashion_mnist_full_dataset/test/Sandal
DataLoader (test): Processing category: Sneaker at /home/container/navya/RadiImageNet/fashion_mnist_full_dataset/test/Sneaker
DataLoader (test): Processing category: Dress at /home/container/navya/RadiImageNet/fashion_mnist_full_dataset/test/Dress
DataLoader (test): Processing category: Bag at /home/container/navya/RadiImageNet/fashion_mnist_full_dataset/test/Bag
DataLoader (test): Processing category: T-shirt at /home/container/navya/RadiImageNet/fashion_mnist_full_dataset/test/T-shirt
DataLoader (test): Total images found: 10000
DataLoader (test): Total labels found: 10000
DataLoader (test): Number of unique labels after encoding: 10
Test data: 10000 images, 10 classes
Evaluating model on unseen test data...
Loaded model weights from: Eff_output_small/models/final_model.weights.h5
Predicting on test data...
313/313 [=====] - 10s 27ms/step

Classification Report:
precision    recall  f1-score   support

Ankle boot   1.00      0.67      0.80      1000
Bag           0.92      1.00      0.96      1000
Coat          0.32      0.99      0.48      1000
Dress         0.99      0.14      0.24      1000
Pullover      1.00      0.81      0.90      1000
Sandal        0.74      1.00      0.85      1000
Shirt         0.26      0.37      0.30      1000
Sneaker       0.80      0.78      0.79      1000
T-shirt       0.99      0.17      0.29      1000
Trouser       0.92      0.97      0.95      1000

accuracy          0.61      10000
macro avg         0.79      0.61      0.57      10000
weighted avg      0.79      0.61      0.57      10000

Classification report saved to Eff_output_small/evaluation_results/classification_report.txt

Confusion Matrix:
[[667  0  0  0 138  0 195  0  0]
 [  0 990  3  0  0  1  0  0  0]
 [  0  5 995  0  0  0  0  0  0]
 [  0 12 517 136  0 1250  0 177]
 [  0 10 859  10 10 121  0  0  0]
 [  0  3  0  0 997  0  0  0  0]
 [  0 25 687  0  0  0 366  0 11]
 [  1  3  0  0 212  0 784  0  0]
 [  0 28 125  0  0 677  0 167  3]
 [  0  1 21  1  0  0  2  0 974]]

Confusion matrix saved to Eff_output_small/evaluation_results/confusion_matrix.png
Predictions saved to Eff_output_small/evaluation_results/predictions.csv
(container@jarvis:~/navya/RadiImageNet)$ python main.py --model_name EfficientNetB0 --train_dir ./navya/RadiImageNet/fashion_mnist_full_dataset/train --test_dir ./navya/RadiImageNet/fashion_mnist
_full_dataset/test --output_dir ./eff_output_small --lr 0.0001 --batch_size 8 --epochs 5

```

Figure 4.2: Implementation of pipeline in Linux system-2

4.2.2 Colab Notebook

This is an user-friendly method where user can execute the pipeline sequentially utilizing the GPU services of Google's colab notebook. The notebook can be found at: https://colab.research.google.com/drive/1QNk_tiu7IVKGbG1z6ebcBT3VUTg5Jhg5?usp=sharing

4.9: RadImageNetCode.ipynb: This script calculates and reports key performance indicators.

```
1 # %cd /content/
2 !git clone https://github.com/Navya003/RadImageNet.git
3
4 # %cd /content/RadImageNet/
5
6 !pip install scikit-learn numpy pandas opencv-python matplotlib openpyxl
7
8 !python convert_fashion_mnist_full.py
9
10 !python main.py \
11     --model_name VGG16 \
12     --train_dir /content/RadImageNet/fashion_mnist_full_dataset/train \
13     --test_dir /content/RadImageNet/fashion_mnist_full_dataset/test \
14     --output_dir /content/RadImageNet/fashion_Eff_output \
15     --lr 0.0001 \
16     --batch_size 16 \
17     --epochs 5
18
19 !python main.py \
20     --model_name VGG19 \
21     --train_dir /content/RadImageNet/fashion_mnist_full_dataset/train \
22     --test_dir /content/RadImageNet/fashion_mnist_full_dataset/test \
23     --output_dir /content/RadImageNet/fashion_Eff_output \
24     --lr 0.0001 \
25     --batch_size 16 \
26     --epochs 5
27
28 !python main.py \
29     --model_name ResNet50 \
30     --train_dir /content/RadImageNet/fashion_mnist_full_dataset/train \
31     --test_dir /content/RadImageNet/fashion_mnist_full_dataset/test \
32     --output_dir /content/RadImageNet/fashion_Eff_output \
33     --lr 0.0001 \
34     --batch_size 16 \
35     --epochs 5
36
37 !python main.py \
```

```

38  --model_name DenseNet121 \
39  --train_dir /content/RadImageNet/fashion_mnist_full_dataset/train \
40  --test_dir /content/RadImageNet/fashion_mnist_full_dataset/test \
41  --output_dir /content/RadImageNet/fashion_Eff_output \
42  --lr 0.0001 \
43  --batch_size 16 \
44  --epochs 5
45
46 !python main.py \
47  --model_name EfficientNetB0 \
48  --train_dir /content/RadImageNet/fashion_mnist_full_dataset/train \
49  --test_dir /content/RadImageNet/fashion_mnist_full_dataset/test \
50  --output_dir /content/RadImageNet/fashion_Eff_output \
51  --lr 0.0001 \
52  --batch_size 16 \
53  --epochs 5

```

4.3 Results

We trained and evaluated the five CNN architectures using transfer learning on the fashion-MNIST dataset. For each one, we set the learning rate at 0.0001 and used a batch size of 16, running them for five epochs. The models' performance was assessed based on computational efficiency (training time), training convergence, and the mean accuracy derived from 5-fold cross-validation (CV).

Model Architec- ture	Training Time (s)	Final Train Ac- curacy (%)	Final Train Loss	CV Mean Accuracy (%)	CV Mean Loss
VGG16	611.52	98.173	0.070	96.4	0.143
VGG19	768.43	97.960	0.084	96.3	0.152
ResNet50	429.78	98.697	0.059	97.7	0.095
DenseNet121	500.70	98.832	0.048	97.8	0.088
EfficientNetB0	319.07	98.162	0.067	97.0	0.107

Table 4.1: Summarizes the key performance metrics for each of the five models

4.3.1 Accuracy

The results show the superiority of residual and densenet connectivity architectures. DenseNet121 achieved the highest mean cross-validation accuracy at 97.8%, closely followed by ResNet50 at 97.7%. This success confirms the architectural advantage of feature reuse and improved gradient flow provided by the dense connections, enabling these models to better capture subtle distinctions within the clothing images.

On the other hand, the VGG models (VGG16 and VGG19) showed the lowest CV mean accuracy (96.4% and 96.3%, respectively) and the highest CV losses. This is expected, as their simple design often struggles with the degradation problem and feature redundancy compared to modern architectures.

4.3.2 Architectural Efficiency

Computational efficiency appeared as the most distinguishing factor among the models.

- EfficientNetB0 surpassed all other models in terms of speed, completing the full model training in 319.07 seconds. This speed advantage is significant, making it 1.35 times faster than ResNet50 and 2.4 times faster than VGG19, validating its compound scaling approach for efficient resource usage.
- The VGG19 model was the least efficient, requiring 768.43 seconds due to its deep and parameter-heavy structure.

4.3.3 The Performance-Efficiency Trade-Off

The comparison highlights a critical trade-off:

1. VGG models provided a low-performance baseline at a high computational cost.
2. DenseNet121 and ResNet50 provided the peak accuracy but required moderate to high training times.
3. EfficientNetB0 delivered the best balance of speed and performance. It achieved a high CV accuracy of 97.0%, falling only 0.8% behind the DenseNet peak, yet it reduced training time by over 36% compared to DenseNet.

This finding is important for practical application: while DenseNet121 may offer a slight edge, EfficientNetB0 is the optimal choice for real-time applications or environments with limited computational resources where efficiency is prioritized.

5.1 Summary and discussion

This thesis successfully debugged and extended a modular deep learning pipeline, cementing it as a benchmark framework for transfer learning-based image classification. Through its automated architecture, the pipeline validated its robustness and generalizability by performing effectively on the non-medical benchmark (fashion-MNIST).

We chose to perform the comparative architectural analysis on the fashion-MNIST dataset instead of the original COVID-19 medical dataset unbiased architectural benchmarking. Medical imaging datasets, such as the COVID-19 CXR data, often introduce complexities like ethical constraints, and limited data availability. By shifting the primary comparison to MNIST, we gained access to a globally recognized, large-scale, and publicly accessible benchmark. This move allowed us to isolate and measure the differences in efficiency and generalization across the five competing architectures (VGG, ResNet, DenseNet, EfficientNet) without the confounding variables of the clinical domain. The high CV accuracies obtained (up to 97.8%) validate that the modified pipeline, including the EfficientNet integration, functions optimally.

This comparative analysis gave two important observations:

1. Architectural Validation: The integration of the EfficientNet model successfully confirmed its superior efficiency (319.07 s), confirming the advantage of compound scaling. This speed gain makes modern architectures necessary for deployment in resource-constrained environments.

2. Performance Baseline: We confirmed a robust performance hierarchy, confirming that DenseNet121 and ResNet50 deliver maximum classification quality (up to 97.8% CV accuracy), while the VGG family served as a high-cost, low-accuracy baseline.

These results highlight the importance of transfer learning in achieving high performance with limited domain data, aligning directly with contemporary research on effective, responsible AI development [1, 20, 21]. By establishing a reproducible, extensible, and standards-adherent foundation, this work advances the evolution of responsible and effective AI in image classification and beyond.

5.2 Future prospects

Building upon the robust framework and clear performance hierarchy established in this study, we identified three critical points for future research that can further advance the understanding and application of this deep learning pipeline.

5.2.1 Final validation on the original domain

The immediate next step requires re-testing the fully updated pipeline on the original COVID-19 detection task in lung imaging data. We can access the performance and efficiency of the EfficientNet integration and general code modernization (debugging) within the original high-impact medical domain. This step will serve as the final validation of the pipeline's practical utility.

5.2.2 Extended EfficientNet Scaling and Optimization

While EfficientNetB0 delivered exceptional efficiency, its potential for peak accuracy requires further investigation. We recommend expanding the analysis to include scaled-up EfficientNet models (e.g., B3, B5, and B7) and find the most efficient model variant for deployment.

5.2.3 Integration of Advanced Training Strategies

To address the limitations inherent in static dataset training, future work should use advanced techniques that improve performance and usability:

- Model Interpretability Frameworks: Integrate techniques like Grad-CAM [22] to visualize and understand the specific features used by the models for classification, increasing the trust and accountability of the pipeline's decisions.

-
- **Active Learning (AL) Strategies:** Implement AL to intelligently select the most important features (informative data samples) for labeling, potentially maximizing model performance while significantly reducing the manual labeling effort and overall data requirement for future domain applications.
 - **Advanced Domain Adaptation:** Explore unsupervised domain adaptation techniques to further reduce the gap between source (ImageNet) and target (FASHION-MNIST) domains,. minimizing the need for extensive fine-tuning

Bibliography

- [1] Plested, Jo, and Tom Gedeon. "Deep transfer learning for image classification: a survey." *arXiv preprint arXiv:2205.09904* (2022).
- [2] Ball, Joshua. "Few-shot learning for image classification of common flora." *arXiv preprint arXiv:2105.03056* (2021).
- [3] Mongan, John, Linda Moy, and Charles E. Kahn Jr. "Checklist for artificial intelligence in medical imaging (CLAIM): a guide for authors and reviewers." *Radiology: Artificial Intelligence* 2.2 (2020): e200029.
- [4] Chai, J., Zeng, H., Li, A. and Ngai, E.W., 2021. Deep learning in computer vision: A critical review of emerging techniques and application scenarios. *Machine Learning with Applications*, 6, p.100134
- [5] Xiao, Han, Kashif Rasul, and Roland Vollgraf. "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms." *arXiv preprint arXiv:1708.07747* (2017).
- [6] Alzubaidi, Laith, et al. "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions." *Journal of big Data* 8.1 (2021): 53.
- [7] Wang, Di, et al. "Face recognition system based on CNN." 2020 International conference on computer information and big data applications (CIBDA). IEEE, 2020.
- [8] Mustaqeem, N., and Soonil Kwon. "A CNN-assisted enhanced audio signal processing for speech emotion recognition." *Sensors* 20.1 (2019): 183.
- [9] Hema, C., and Fausto Pedro Garcia Marquez. "Emotional speech recognition using cnn and deep learning techniques." *Applied Acoustics* 211 (2023): 109492.
- [10] Zhang, Kanghui, et al. "Computer vision detection of foreign objects in coal processing using attention CNN." *Engineering Applications of Artificial Intelligence* 102 (2021): 104242.

- [11] Chua, Leon O. "CNN: A vision of complexity." *International Journal of Bifurcation and Chaos* 7.10 (1997): 2219-2425.
- [12] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556* (2014).
- [13] He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.
- [14] Huang, Gao, et al. "Densely connected convolutional networks." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017.
- [15] Tan, Mingxing, and Quoc Le. "Efficientnet: Rethinking model scaling for convolutional neural networks." *International conference on machine learning*. PMLR, 2019.
- [16] Sharma, Sakshi, Karthikeyan Shanmugasundaram, and Sathees Kumar Ramasamy. "FAREC—CNN based efficient face recognition technique using Dlib." *2016 international conference on advanced communication control and computing technologies (ICACCCT)*. IEEE, 2016.
- [17] Ball, Joshua. "Few-shot learning for image classification of common flora." *arXiv preprint arXiv:2105.03056* (2021).
- [18] Ferrando, Javier, et al. "Improving accuracy and speeding up document image classification through parallel systems." *International Conference on Computational Science*. Cham: Springer International Publishing, 2020.
- [19] Jain, T., Chauhan, N. K., Niboriya, S. S., Prakash, A., Lynn, A. M. Rapid deployment of pretrained models to develop diagnostic screens from radiological images: Application to detecting covid-19 from chest x-ray images.
- [20] Ilani, Mohsen Asghari, and Yaser Mike Banad. "Brain Tumor Detection Using Transfer Learning-Based CNN Architectures in IoT Healthcare Industries." *2024 International Conference on AI x Data and Knowledge Engineering (AIxDKE)*. IEEE, 2024.
- [21] Ferrando, Javier, et al. "Improving accuracy and speeding up document image classification through parallel systems." *International Conference on Computational Science*. Cham: Springer International Publishing, 2020.
- [22] Selvaraju, Ramprasaath R., et al. "Grad-cam: Visual explanations from deep networks via gradient-based localization." *Proceedings of the IEEE international conference on computer vision*. 2017.
- [23] Bansal, Aryan Ramesh Pillai Riya Anjali. "Neural Networks and Deep Learning." (2025).