DEEP LEARNING-DRIVEN PEST DETECTION AND CLASSIFICATION WITH INSTANT SMS ALERTS FOR PRECISION AGRICULTURE

Report submitted to the SASTRA Deemed to be University
as the requirement for the course

CSE300 - MINI PROJECT

Submitted by

CHILUKURI HEMANTH SAI NAG
(Reg. No.: 226003030, B. Tech CSE)
DEVIREDDY DEEPAK REDDY
(Reg. No.: 226003039, B. Tech CSE)
GOGULA SAI AASISH
(Reg. No.: 226003050, B. Tech CSE)

MAY 2025

Department of Computer Science and Engineering

SRINIVASA RAMANUJAN CENTRE

KUMBAKONAM, TAMIL NADU, INDIA - 612 001

SASTRA

ENGINEERING • MANAGEMENT • LAW • SCIENCES • HUMANITIES • EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

THANJAVUR | KUMBAKONAM | CHENNAI

Department of Computer Science and Engineering

SRINIVASA RAMANUJAN CENTRE

KUMBAKONAM, TAMIL NADU, INDIA – 612 001

Bonafide Certificate

This is to certify that the report titled "**Deep Learning-Driven Pest Detection And Classification With Instant SMS Alerts For Precision Agriculture**" submitted as a requirement for the course, **CSE300: MINI PROJECT** for B.Tech. is a bonafide record of the work done by **Mr. Chilukuri Hemanth Sai Nag (Reg. No. 226003030), Mr. Devireddy Deepak Reddy (Reg. No. 226003039)** and **Mr. Gogula Sai Aasish (Reg. No. 226003050)** during the academic year 2024-25, in the Srinivasa Ramanujan Centre, under my supervision.

Signature of Project Supervisor :

Name with Affiliation : **Mr. Ramesh R/AP-I/CSE/SRC/SASTRA**

Date : 25/4/2025

Mini Project *Viva voce* held on 26/4/25

Examiner 1

Dr. M. Marthaa

[AP II / CSE / SRC]

Examiner 2 26/4/25

[S. SUGANYA / AP-II]

i

Acknowledgments

We would like to thank our Honorable Chancellor, **Prof. R. Sethuraman**, for providing us with the opportunity and the necessary infrastructure for carrying out this project as a part of our curriculum.

We would like to thank our Honorable Vice-Chancellor, **Dr. S. Vaidhyasubramaniam**, and **Dr. S. Swaminathan**, Dean, Planning & Development, for the encouragement and strategic support at every step of our college life.

We extend our sincere thanks to **Dr. R. Chandramouli**, Registrar, SASTRA Deemed to be University for providing the opportunity to pursue this project.

We extend our heartfelt thanks to **Dr. V. Ramaswamy**, Dean, and **Dr. A. Alli Rani**, Associate Dean, Srinivasa Ramanujan Centre, SASTRA Deemed to be University. We are also pleased to express our gratitude to **Dr. V. Kalaichelvi**, Associate Professor, Department of Computer Science and Engineering, Srinivasa Ramanujan Centre, for her encouragement and support during our project work.

Our guide, **Mr. Ramesh R**, AP-I, Department of Computer Science and Engineering, Srinivasa Ramanujan Centre, was the driving force behind this whole idea from the start. His deep insights in the field and invaluable suggestions helped us in making progress throughout our project work. We also thank the project review panel members, **Dr. Martina M** and **Dr. Suganya S**, for their valuable comments and insights which made this project better.

We would like to extend our special thanks to **Mr. M. Jeya Pandian**, Assistant Professor-II & Project Coordinator, Department of Computer Science and Engineering, Srinivasa Ramanujan Centre, for organizing and supporting us in the successful completion of the project.

We gratefully acknowledge all the contributions and encouragement from my family and friends resulting in the successful completion of this project. We thank you all for providing us an opportunity to showcase our skills through project.

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
CPU	Central Processing Unit
CNN	Convolutional Neural Network
FSL	Few-Shot Learning
GAT	Graph Attention Network
GCN	Graph Convolutional Network
GNN	Graph Neural Network
GPU	Graphics Processing Unit
OpenCV	Open-Source Computer Vision Library
R	Recall
SSL	Self-Supervised Learning
ViT	Vision Transformer

Notations

English Symbols

a	First input image (e.g., a new pest image to identify in Siamese Network)
b	Second input image (e.g., a known pest image from the support set)
d	Distance between two feature vectors (embeddings) Euclidean distance between image embeddings
y	Label indicating if the image pair is similar (0) or dissimilar (1)
m	Margin in contrastive loss (typically m=1.0m = 1.0m=1.0)
L	Contrastive loss function

Greek Symbols

$\phi(x)$	Feature extractor function (e.g., CNN encoder or MobileNetV2)
$\phi(a)$	Feature vector of image a
$\phi(b)$	Feature vector of image b
$\sum$	Summation

Abstract

Pest detection in agriculture is a critical task for ensuring crop health and improving yield, especially in data-scarce environments. This work explores the application of Few-Shot Learning (FSL) as a promising alternative to traditional deep learning models such as CNNs, particularly in scenarios with limited labeled data. In contrast to conventional models, FSL demonstrates improved adaptability and generalization with minimal supervision.

To enhance performance across various data conditions, we compare and evaluate multiple advanced models including FSL, Graph Neural Networks (GNNs), Vision Transformers (ViT), and MobileNetV2 enhanced through Self-Supervised Learning **(SSL)**. GNNs are used to model spatial and relational patterns by converting image data into graph structures, while ViT leverages strong global feature representation for large-scale datasets. SSL further aids in learning robust feature representations from unlabeled data, minimizing the need for extensive manual annotation.

The system accepts pest detection input in **three formats**: **live video stream**, **image uploads**, and **video file uploads**, providing flexibility in real-world deployment. Real-time video input is processed using OpenCV, and **Streamlit** is employed to create an intuitive, lightweight, and interactive web-based interface. For mobile and edge deployment, the models are optimized using **TensorFlow Lite** for efficient on-device inference. Additionally, **WhatsApp integration** allows automatic pest detection alerts to be sent directly to users, ensuring timely intervention.

This comprehensive, multi-modal framework aims to deliver a practical, scalable, and user-friendly solution for intelligent pest detection in agricultural fields, especially suited for resource-constrained environments.

Keywords: Few Shot Learning - SSL - GNN - ViT – Pest Detection – Real Time Processing **– Agriculture AI Systems.**

Table of Contents

CHAPTER 1

SUMMARY OF THE BASE PAPER

Title: Pest classification: Explainable few-shot learning vs. convolutional neural networks vs. transfer learning

Journal Name: Scientific African

Publisher: Elsevier

Year: 2025 (Accepted: December 17, 2024; Available online: December 21, 2024)

Indexing: Scopus, SCI, SCIE (as it is published in Scientific African, which is indexed in these databases)

1. Summary

The paper titled "Pest classification: Explainable few-shot learning vs. convolutional neural networks vs. transfer learning" was published in Scientific African by Elsevier B.V. in 2025. It is indexed in Scopus, SCI, and SCIE. The study introduces Explainable Few-Shot Learning (FSL) for agricultural pest classification, offering a novel solution to the limitations posed by traditional methods such as CNNs and transfer learning, which rely heavily on large labeled datasets and lack interpretability.

The research addresses the issue of data scarcity in pest detection by applying FSL models—specifically, Prototypical Networks and Siamese Networks—that can learn from just a few samples per class. To enhance transparency, the study integrates explainability techniques like Grad-CAM, enabling visual interpretation of model decisions. The models were tested across three different datasets: Full Pest Images, Half Pest Images (cropped pests), and Malaysian Pest Images, each presenting varying challenges and degrees of data completeness.

Explainable FSL proved to be highly effective, with the Prototypical Network achieving a peak accuracy of 99.81% with only 5 training samples (shots) in the Full and Malaysian Pest datasets. The Siamese Network also performed well, particularly for one-shot learning tasks, reaching up to 89.12% accuracy. In contrast, CNN and transfer learning models required significantly more training data to achieve similar performance levels. The CNN model, for instance, required up to 270 shots to reach above 99% accuracy. Transfer learning using MobileNetV2, while efficient with pretrained knowledge, showed limitations in generalizing with fewer training samples and plateaued at lower accuracies for partial datasets.

The architecture of the Prototypical Network is based on computing Euclidean distances between learned class prototypes and query images in the embedding space. The Siamese Network, composed of twin networks, measures similarity between image pairs using contrastive loss. CNNs follow a standard deep learning architecture, and MobileNetV2-based transfer learning repurposes features from pretrained models. Grad-CAM was applied across

all architectures to identify the most influential visual features in the classification process, enhancing interpretability and user trust.

The research also details a robust methodology including preprocessing (resizing to 28×28 pixels, normalization, and rotation), data augmentation (using Keras to generate 10x additional images), and a meta-learning framework for FSL tasks (N-way, K-shot classification). The training and testing processes were conducted using Kaggle Notebooks with Python 3.7 on a machine equipped with Intel i5 CPU and GTX 1050 GPU.

This work demonstrates that Explainable FSL models not only outperform traditional CNN and transfer learning models in data-scarce conditions but also provide interpretable predictions that are vital for real-world deployment in agriculture. It fills an important gap in the literature by being the first to combine FSL and explainability for pest detection, and evaluates models using unique datasets including incomplete images. The findings contribute to the development of accurate, efficient, and transparent pest detection systems, making them especially useful for resource-constrained settings and supporting global food security goals. This research aligns with SDG Target 2.4 and the African Union's Agenda 2063 by promoting sustainable, resilient agricultural practices. The authors suggest exploring other meta-learning models like MAML and Relation Networks in future work to further advance this field

1.1 Introduction

Agricultural productivity faces persistent threats from pests, which can severely damage crops and diminish yields if not detected and addressed in a timely manner. Traditional pest identification methods, largely reliant on manual inspection by experts, are labor-intensive, error-prone, and impractical for large-scale deployment, especially in rural regions with limited access to skilled personnel. As global food demands rise, there is a growing need for intelligent, automated pest detection systems that are accurate, efficient, and adaptable to real-world agricultural environments.

Recent advancements in computer vision have enabled automated pest detection through deep learning models; however, these models typically demand large volumes of annotated training data—a resource that is often scarce in agricultural contexts. To address this, **Few-Shot Learning (FSL)** is explored as a viable alternative, enabling models to generalize from just a few labeled samples and mimicking the human ability to recognize new patterns with minimal supervision. This makes FSL particularly suitable for pest identification in low-resource and highly diverse environments.

To compare and enhance model performance across different data scales, this research investigates multiple learning strategies, including **Convolutional Neural Networks (CNNs), Vision Transformers (ViT), Graph Neural Networks (GNNs)**, and **MobileNetV2 combined with Self-Supervised Learning (SSL)**. ViT models offer robust representation learning for large datasets by capturing long-range dependencies, while GNNs are utilized to model spatial relationships by converting images into graph structures, improving the ability to distinguish visually similar pests. SSL techniques allow the model to learn effective features from unlabelled data, reducing reliance on costly manual annotation and enhancing generalization.

The system supports **three input modalities**: **live video stream**, **image uploads**, and **video file uploads**, providing flexibility for diverse field applications. **OpenCV** is used to capture and process real-time visual data, enabling continuous and responsive pest monitoring. The model is optimized using **TensorFlow Lite**, ensuring low-latency, high-efficiency inference on mobile and edge devices. To improve usability and accessibility, a **Streamlit-based user interface** is developed, offering an interactive and intuitive platform for farmers and agricultural workers. Furthermore, the system is integrated with **WhatsApp messaging**, enabling instant pest detection alerts and facilitating timely interventions.

This work presents a comprehensive and scalable pest detection framework that combines state-of-the-art machine learning techniques with practical deployment tools. Designed with real-world constraints in mind, the system supports intelligent, real-time pest management in resource-limited settings, contributing to smarter, more sustainable agricultural practices.

1.2 Problem Statement

Pest detection in agriculture is critical for crop protection but is limited by the scarcity of annotated data and the need for real-time analysis. Traditional deep learning models require large datasets and computational resources, making them unsuitable for field deployment. Farmers in rural or resource-constrained areas lack access to such advanced tools. There is a need for a lightweight, adaptable, and efficient pest detection system that works with minimal labeled data.

1.3 Objective

This work aims to develop a pest detection framework using Few-Shot Learning, enabling accurate identification with limited labeled samples. Vision Transformers and Graph Neural Networks are utilized to enhance representation and spatial understanding of pests. Self-Supervised Learning techniques are incorporated to reduce dependency on manual annotation. The final system is optimized for real-time use on edge devices, with a user-friendly interface and automated alert system.

1.4 Literature Survey

Pest detection in agriculture has seen significant advancements due to the emergence of computer vision and machine learning techniques. Traditional pest identification methods, which rely heavily on manual observation, are time-consuming, inconsistent, and often inaccurate, particularly when pests exhibit visually similar features. Consequently, researchers have explored various artificial intelligence (AI) methods to automate and enhance the accuracy of pest classification.

Early approaches to pest detection utilized traditional machine learning algorithms such as Support Vector Machines (SVM), k-Nearest Neighbors (k-NN), and Naïve Bayes (NB). These

models required manual feature extraction and demonstrated limited scalability and generalization capability, especially with complex agricultural images or under variable lighting and environmental conditions.

Deep learning models, particularly Convolutional Neural Networks (CNNs), have revolutionized image-based pest detection. Liu et al. (2019) proposed PestNet, a CNN-based framework enhanced with Channel-Spatial Attention (CSA), Region Proposal Networks (RPN), and Position-Sensitive Score Maps (PSSM) to improve detection performance. Similarly, Malathi and Gopinath (2021) applied a ResNet-50-based transfer learning model for paddy pest recognition, while Pattnaik et al. (2020) used DenseNet169 for tomato pest classification. These models demonstrated strong performance but required large amounts of labeled training data, which is often unavailable in real-world agricultural settings.

To address the data scarcity problem, recent research has turned to Few-Shot Learning (FSL), which aims to learn robust classifiers from a limited number of labeled examples. FSL has gained traction in various fields such as scene classification (Alajaji et al., 2020), text classification (Muthukumar, 2021), and hyperspectral image classification (Li et al., 2021). In agriculture, FSL has been applied to plant disease detection (Argüeso et al., 2020), fruit ripeness classification (Ng et al., 2022), and leaf identification. Li and Yang (2020) introduced few-shot cotton pest recognition, while Nuthalapati and Tunga (2021) explored transformer-based embeddings for pest classification tasks. However, these studies largely overlooked the need for model explainability, a crucial aspect when deploying AI systems in sensitive domains like agriculture.

Explainable AI (XAI) techniques such as Grad-CAM (Gradient-weighted Class Activation Mapping) have emerged to bridge this gap by visually identifying which image regions influenced the model's prediction. Despite their effectiveness, most prior pest detection studies did not integrate explainability into their models, resulting in black-box solutions that lacked interpretability.

The current paper by Nitiyaa Ragu and Jason Teo makes a notable contribution by integrating explainability into few-shot learning models. It utilizes Prototypical Networks and Siamese Networks, both of which are metric-based FSL methods. These models are capable of learning class representations from a few examples and generalizing to unseen categories. The Prototypical Network identifies class prototypes in the embedding space, while the Siamese Network computes similarity between image pairs. The use of Grad-CAM alongside these models enables the generation of heatmaps that explain model predictions, enhancing trust and transparency.

In addition to FSL methods, the paper benchmarks the performance of traditional CNNs and transfer learning models such as MobileNetV2. While these models perform well with large datasets, they are less suitable for data-scarce environments. The authors demonstrate that Explainable FSL models not only match or exceed the accuracy of CNNs and transfer learning models but also offer the added benefit of interpretability.

In summary, the literature reveals a growing interest in applying FSL to pest detection, especially in contexts with limited annotated data. The inclusion of explainability techniques marks a significant step forward, providing actionable insights into model behavior and fostering user confidence. This evolution in pest detection approaches underscores the transition from black-box to transparent, data-efficient AI solutions for smart agriculture.

1.5 Architecture Diagram

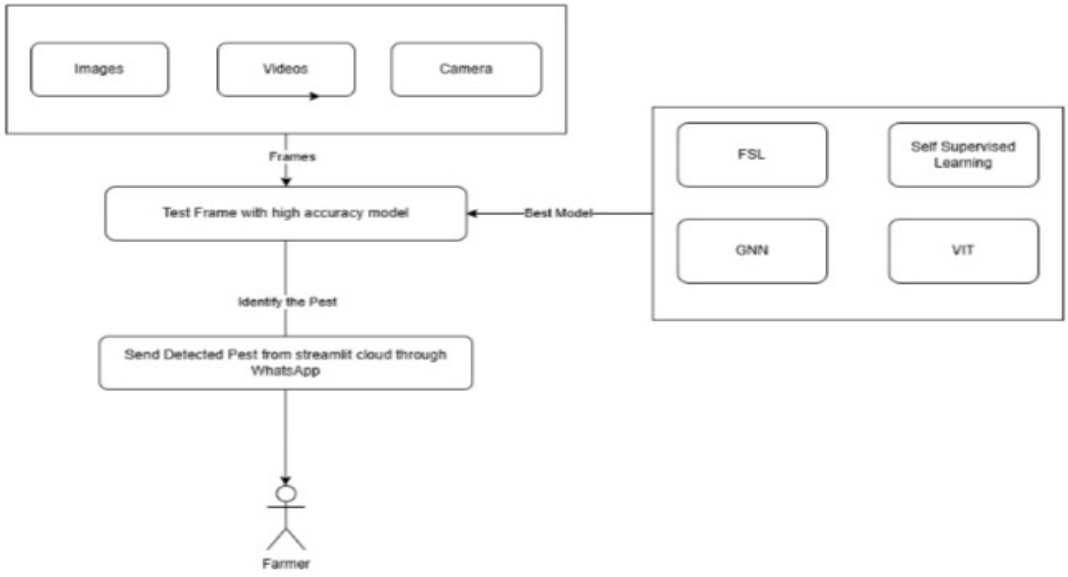


Fig 1.5 Architecture Diagram

1.6 Hardware and Software Requirements

This study uses Windows 11 Home Single Language as the operating system, Jupyter Notebook as the coding environment, and Python 3.12 64bit for developing and executing machine learning models. The hardware requirements include an Intel Core i5–1204P, 16GB RAM and 512GB ROM.

1.7 Modules and Descriptions

1.7.1 Data Description

Classes	aphids	armyworm	beetle	bollworm	grasshop-per	mites	mosquito	sawfly	Stem borer
									

Fig. 1. Pest images from three datasets: Full Pest Images. The datasets include various pest classes such as aphids, armyworms, bollworms, grasshoppers, mites, and others. Adapted from : https://www.kaggle.com/simranvolunesia/pest-dataset;

1.7.2 Data Acquisition

The primary dataset used in this project is sourced from Kaggle (dataset) and includes images of nine major agricultural pests: aphids, armyworm, beetle, bollworm, grasshopper, mites, mosquito, sawfly, and stem borer. These full-view images form the foundation for classification and explainability tasks. A secondary dataset from the same source contains half-cropped images of these pests from different angles, aiding in Few-Shot Learning (FSL) and explainability evaluation. Additionally, a third dataset comprising Malaysian rice pest images includes nine classes with 300 training images per class (2,700 total), adding diversity in shape, colour, and orientation, which is essential for training robust and interpretable models.

1.7.3 Data Pre-Processing

Preprocessing ensures the input data is clean, consistent, and suited for learning high-level visual features. Raw colour images are used to preserve natural patterns, vital for interpretability in models like Vision Transformers (ViT), GNNs, FSL, and MobileNetV2. All images are resized to 224×224 pixels for compatibility and computational efficiency. Pixel values are normalized to [0, 1] using a factor of 1/255 to stabilize and accelerate training.

The dataset is split as follows: **Training Set (70%)**: for model learning **Validation Set (10%)**: for hyperparameter tuning **Testing Set (20%)**: for final model evaluation

1.7.4 Data Augmentation

To enhance model generalization and address the limited variability in pest image datasets, a series of data augmentation techniques are applied to the training and validation sets. Data augmentation artificially increases the diversity of the dataset by applying controlled transformations to the existing images, simulating different real-world conditions and perspectives.

The augmentation pipeline includes the following transformations:

Rotation: Random rotations up to 30° to simulate pest appearances at different orientations. Width and Height Shifts: Horizontal and vertical shifts up to 10% and 20%, respectively, to mimic displacement in field-captured images. Shearing: Applied at 20° to emulate camera

angle variations. Zooming: Random zoom up to 80% to introduce scale diversity. Horizontal and Vertical Flipping: Reflects the insect's orientation from multiple perspectives. ZCA Whitening: Applied with epsilon = 1e-06 to decorrelate image pixels, improving contrast and model sensitivity to edges and textures. Fill Mode: Missing pixels from transformations are filled using the nearest neighbour strategy.

1.7.5 Few Shot Learning

Few-Shot Learning (FSL) is a paradigm where models are trained to recognize new categories with only 1–5 labelled samples per class—mirroring human-like generalization [21] [22] [23]. Unlike conventional deep learning that requires thousands of labelled samples, FSL is especially effective in agriculture where acquiring annotated pest images is labour-intensive and costly [24] [25].

How FSL Works

Few-Shot Learning typically involves meta-learning, or "learning to learn." Instead of training on fixed classes, the model is trained on many small tasks (called episodes) that simulate few-shot conditions. Each task has: A support set: a few labelled examples from each class (e.g., 1–5 images per class). A query set: unlabelled examples the model must classify using the support set. This way, the model learns how to compare and distinguish between classes based on a few examples, rather than memorizing fixed patterns.

This setup enables the model to learn class-wise similarities and adapt to new categories [21] [22].

Types of FSL Approaches:

Common approaches in FSL include:

Metric-Based Learning (e.g., Siamese Networks, Prototypical Networks): Learn a similarity function to compare query images with the support set.
Optimization-Based Learning (e.g., MAML - Model-Agnostic Meta-Learning): Learn model parameters that can adapt quickly to new tasks.
Memory-Augmented Networks: Use external memory to store and recall knowledge from previous tasks.

This approach has shown success in various bio-agricultural tasks [22] [25] [30].

FSL - Siamese Network

A Siamese Network is a type of neural network architecture used to learn similarity between two inputs—especially useful in Few-Shot Learning tasks. Instead of directly classifying an image, the network learns how similar or different two images are.

A Siamese Network consists of **two identical subnetworks (shared weights)** that process two input images in parallel. Each subnetwork extracts **feature vectors** from the input images.

a : First image (e.g., a new pest image to identify)

b : Second image (e.g., a known pest image from the support set)

$\phi(x)$: Feature extractor network (e.g., MobileNetV2 or a CNN encoder)

$\phi a = \phi(a)$, $\phi b = \phi(b)$

These outputs ϕa and ϕb are **embedding vectors** representing the images.

Distance Calculation:

The network computes a **distance metric** (usually Euclidean distance or cosine similarity) between the two embeddings:

Euclidean Distance: $d = \| \phi a - \phi b \|^2 = \sqrt{i}\sum(\phi a, i - \phi b, i)^2$

The Siamese network is trained using a **contrastive loss** to bring similar images closer and push dissimilar ones apart.

When y=0: The loss becomes $0.5d^{-2}$, encouraging similar images to have **closer embeddings** (small distance). m is the margin (e.g., m=1.0m = 1.0m=1.0)

When y=1: The loss becomes $0.5\max(0,m-d)^2$, pushing dissimilar images **apart** by at least margin m.

$d = \| \phi a - \phi b \|^2 = \sqrt{i}\sum(\phi a, i - \phi b, i)^2$ (Euclidean distance)

The **contrastive loss** is:

$L = (1-y).(0.5d^2) + y.(0.5\max(0,m-d)^2)$

Where:

m = margin (minimum distance between embeddings of different classes)

This encourages:

Similar pairs (same class): small distance

Dissimilar pairs (different classes): distance greater than margin

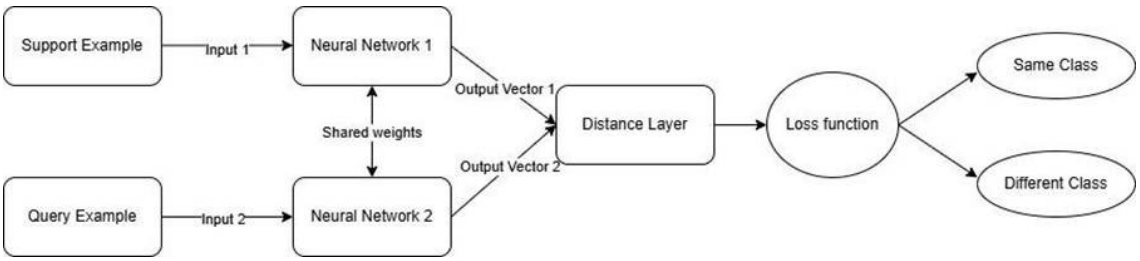


Fig 1.7.4 Siamese Network Architecture Diagram

1.7.6 ViT - Visual Transformer

The Vision Transformer (ViT) is a deep learning model designed for image recognition tasks, which leverages transformer architecture, initially developed for natural language processing. Unlike traditional CNNs, ViT divides an image into fixed size patches, linearly embeds them, and processes them as sequences, like how transformers process text. This approach allows ViT to capture long-range dependencies and achieve state-of-the-art performance in image classification tasks, especially when trained on large datasets.

Vision Transformers (ViT) leverage transformer architecture for image classification by converting images into sequences of patch embeddings—enabling global context modelling [11] [12] [14].

Patch Embeddings: Instead of processing pixels directly, ViT splits the image into non-overlapping patches, typically of size 16x16. Each patch is flattened into a 1D vector and linearly embedded into a higher-dimensional space. These patch embeddings are treated as tokens (like words in NLP tasks) for the transformer.

Transformer Architecture: ViT uses a standard transformer architecture, with multi-head self-attention layers and feed-forward neural networks. The transformer layer learns relationships between different patches, allowing the model to capture global context, which traditional convolutional layers may miss.

Positional Encoding: Since transformers are not inherently spatial, ViT adds positional encodings to the patch embeddings to retain information about the spatial location of patches in the image.

Self-Attention: The key feature of the transformer model is self-attention, which helps the model focus on important regions in the image by assigning different weights to different patches based on their relevance. This enables ViT to better understand long-range dependencies.

Training Data Requirement: One of the challenges of ViT is that it requires a large amount of training data to perform well, as transformers typically excel when trained on large-scale datasets (such as ImageNet or JFT-300M).

Advantages: ViT has been shown to outperform traditional CNNs on large datasets, particularly in terms of scalability. The transformer model is more flexible and can easily integrate with other deep learning architectures.

Applications: ViT is primarily used for image classification, but its transformer-based nature makes it suitable for a wide range of computer vision tasks like object detection, segmentation, and even video analysis.

ViT requires large datasets to reach optimal performance but can be fine-tuned for pest classification using transfer learning [19] [20].

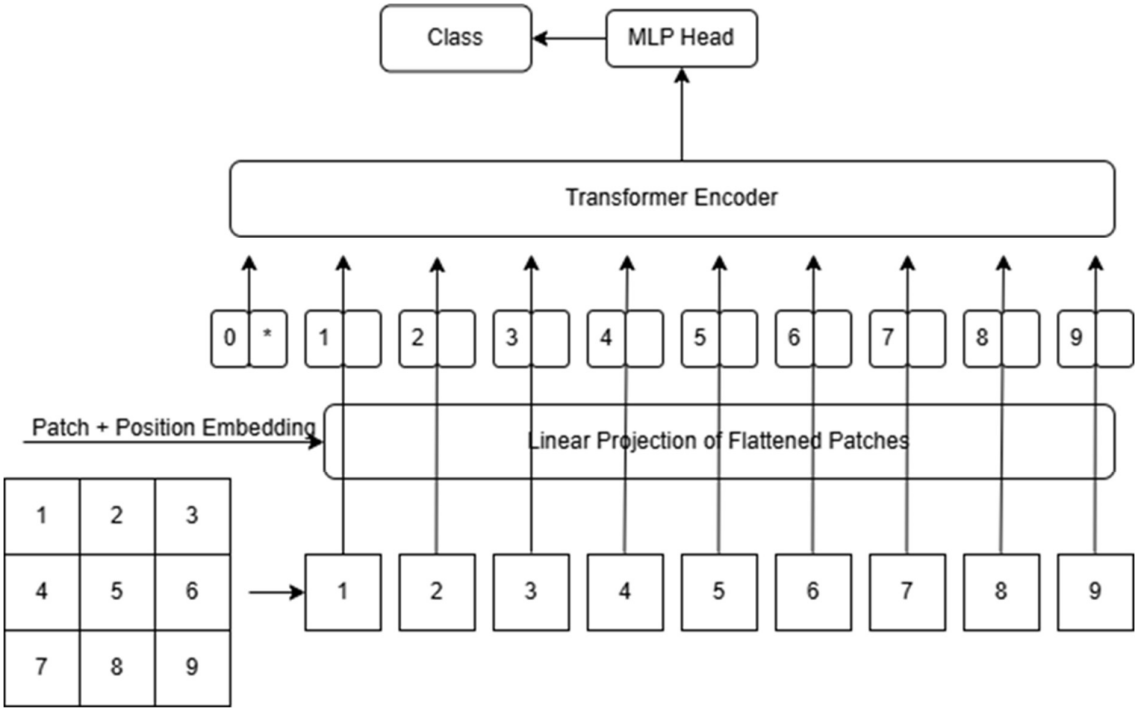


Fig 1.7.5 Visual Transformer Architecture Diagram

1.7.7 GNN- Graphical Neural Network

Graph Neural Networks (GNNs) are a class of deep learning models designed to work directly with graph-structured data. They are particularly useful for problems where the data can be represented as a graph, with nodes representing entities and edges representing relationships between them. Here's a quick breakdown:

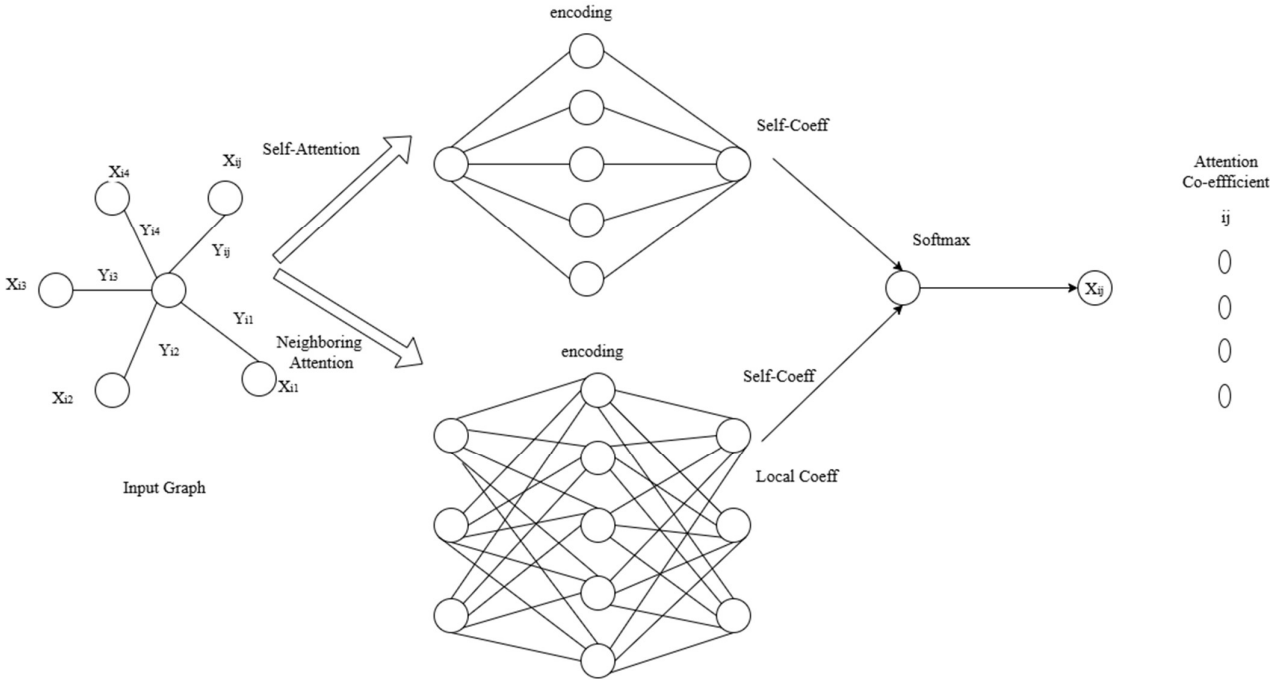


Fig 1.7.6 Graphical Neural Network Architecture Diagram

Graph Representation: A graph is made up of nodes (vertices) and edges (connections). Each node can have features, and each edge can represent a relationship between two nodes, with its own attributes.
Message Passing: In GNNs, information (or "messages") is passed between nodes in the graph. Each node aggregates information from its neighbours through the edges. This process is repeated for several layers, allowing the model to learn from local and global structures in the graph.
Aggregation Functions: The core operation in GNNs is the aggregation step, where a node aggregates the features of its neighbouring nodes. Common aggregation functions include sum, mean, or max. After aggregation, the node's feature is updated using this information and its own feature.
Node, Edge, and Graph-Level Tasks: GNNs can be used for:
Node classification: Classifying individual nodes in the graph.
Edge classification: Classifying the relationships between nodes.
Graph classification: Classifying the entire graph as a whole.
Link prediction: Predicting missing edges or future connections.

Types of GNNs:

GCN (Graph Convolutional Networks): One of the most well-known GNNs, where each node updates its feature by aggregating the features of its neighbours.

GAT (Graph Attention Networks): A variation of GNNs that uses attention mechanisms to weight the importance of neighbouring nodes when aggregating information.

Graphs AGE: This method uses a sampling approach to aggregate information from a subset of neighbours, making it scalable to large graphs.

Applications:

Social Networks: Identifying communities, detecting fake news, recommending friends, etc.
Knowledge Graphs: Used in search engines, semantic analysis, and question answering systems.
Drug Discovery: Modelling molecules as graphs to predict properties and interactions.
Traffic Prediction: Modelling road networks for traffic flow and accident prediction.

Challenges:

Scalability: GNNs can struggle with very large graphs, requiring efficient sampling or partitioning methods.

Over-smoothing: In deeper GNNs, node features may become too similar after many layers of aggregation, causing the model to lose useful information.

GNN Mechanics

Message Passing: Nodes share and aggregate neighbour information, **Aggregation Functions**: Mean, sum, or max operations used.

Applications:

Node classification: Identify regions within pests.

Graph classification: Identify entire pest species.

Edge prediction: Understand pest parts or environment
Popular variants include GCN, GAT, and Graphs AGE [6] [5]. GNNs have been proven effective in domains requiring relational reasoning such as drug discovery, traffic prediction, and agriculture [5] [6].

1.7.8 Self-Supervised Learning with MobileNetV2:

Self-supervised learning is a paradigm where the model learns to predict certain features or patterns from the data without explicit labelled data (no ground truth labels). Instead, the model generates labels from the input data itself.

MobileNetV2 can be adapted for self-supervised learning by using the following strategies:

Pretext Tasks: In self-supervised learning, pretext tasks are often used to help the model learn useful representations from data. These tasks can be designed in a way that encourages the model to learn meaningful features that can be transferred to downstream tasks (like classification, segmentation, etc.). Some common pretext tasks that could work with MobileNetV2 include:

Contrastive Learning: This approach trains the model to distinguish between similar and dissimilar image pairs. For example, **Simclar** or **MoCo** (Momentum Contrast) methods are popular contrastive learning frameworks where MobileNetV2 can be used as the backbone.

Predictive Tasks: These tasks involve predicting parts of an image or the future context of a sequence of images. For example, **rotation prediction**, where the model learns to predict how an image has been rotated.

Colorization: The model learns to predict the missing colour channels from grayscale images.

Inpainting or Context Prediction: Where parts of an image are masked, and the model learns

to predict the missing parts.

Fine-Tuning: After training the MobileNetV2 model on a self-supervised task, the learned features can be fine-tuned on a labelled dataset for downstream tasks, like classification, detection, or segmentation. This process is often referred to as **transfer learning**. The feature representations learned through self-supervision are expected to be useful in a variety of tasks.

Backbone for Self-Supervised Frameworks: MobileNetV2 can be used as the backbone in self-supervised learning frameworks such as:

BYOL (Bootstrap Your Own Latent): A self-supervised learning method where the model learns representations without negative samples, typically using a Siamese network architecture. MobileNetV2 can serve as the encoder in such systems.

MoCo: Momentum Contrast is another self-supervised method, and MobileNetV2 can be used as the backbone network for feature extraction.

Advantages of Using MobileNetV2 for Self-Supervised Learning:

Efficiency: MobileNetV2 is designed for efficiency and speed, making it ideal for deployment on resource-constrained devices. Using it for self-supervised learning allows it to learn meaningful representations without large amounts of labelled data, which is especially useful for edge devices or in domains where labelled data is scarce.

Transferability: The learned features from self-supervised tasks using MobileNetV2 can be easily transferred to other tasks, leading to better performance in downstream supervised tasks.

Scalability: MobileNetV2 can be scaled up or down depending on the computational resources available, making it suitable for both small and large datasets.

Example Pipeline:

Step 1: Pretrain MobileNetV2 on a self-supervised task like contrastive learning (e.g., Simclar) using unlabelled data.

Step 2: Extract Features from the pre-trained MobileNetV2 model. These features will be learned representations that capture useful information from the images.

Step 3: Fine-tune the MobileNetV2 model with a supervised task (e.g., image classification) using a smaller labelled dataset, leveraging the learned features from the pretext task.

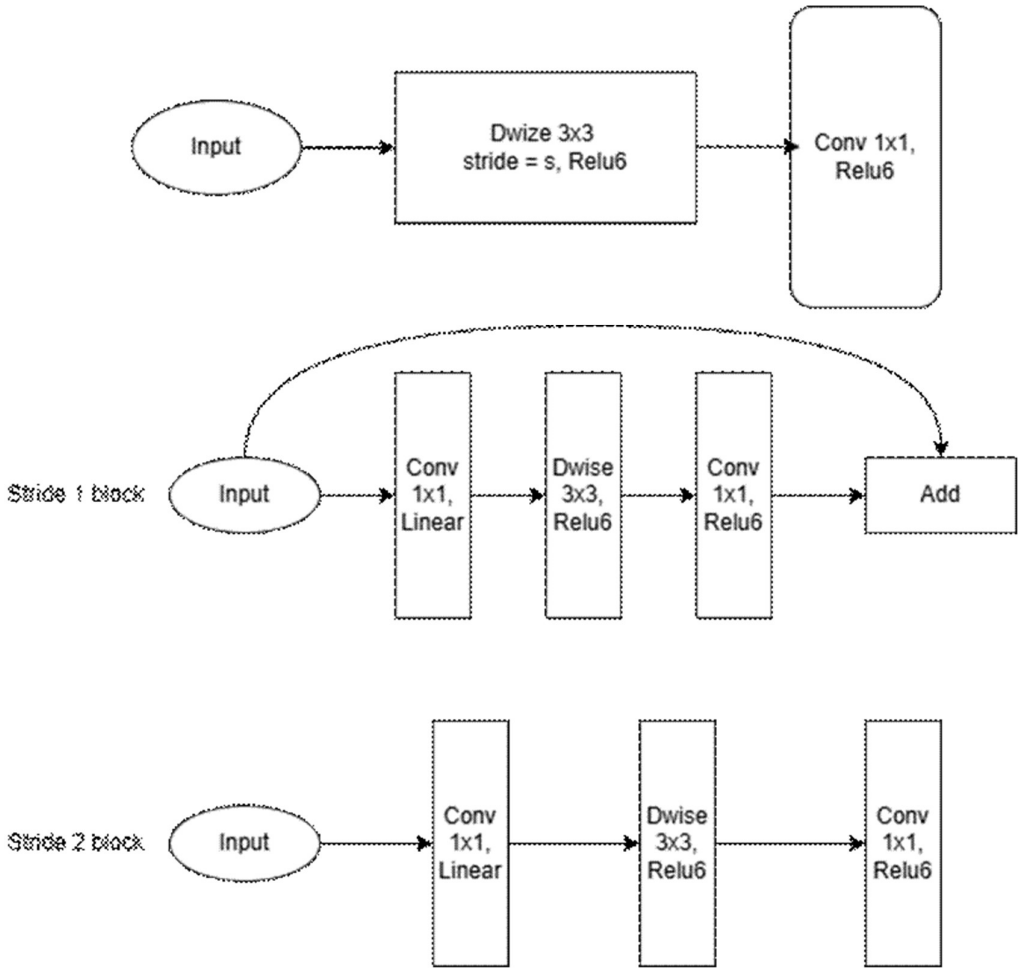


Fig 1.7.7 Self-Supervised Learning (mobilenetV2) Architecture Diagram

1.7.9 CNN (Convolutional Neural Network)

A **Convolutional Neural Network (CNN)** is a type of deep learning model specifically designed to process structured arrays of data such as images. It is especially powerful in tasks like **image classification, object detection, and facial recognition**.

Input Layer : Takes in image data (usually in the form of a 3D array: height × width × channels).

Convolutional Layer : Applies a set of filters (kernels) to extract important features like edges, corners, or textures. Produces **feature maps** that highlight the presence of those features.

Activation Function (ReLU) : Introduces non-linearity into the model. ReLU (Rectified Linear Unit) is most commonly used: *ReLU(x)=max (0,x)\text{ReLU}(x) = \max(0, x)*ReLU(x)=max(0,x)

Pooling Layer : Reduces spatial dimensions (width and height) of the feature maps. Common methods: **Max Pooling** (takes the max value) and **Average Pooling**.

Fully Connected (Dense) Layers : After several convolution and pooling layers, the output is flattened and fed into one or more fully connected layers to make the final classification.

Output Layer : Produces the final result, like class probabilities (e.g., dog, cat, car).

Why CNNs Are Good for Images

- **Parameter sharing**: Reduces the number of parameters drastically.
- **Local connectivity**: Focuses on local patterns like textures or edges.
- **Translation invariance**: Can recognize features even when the position changes slightly in the image.

Common CNN Architectures

- **LeNet-5**: Early CNN for handwritten digit recognition.
- **AlexNet**: Won the ImageNet challenge in 2012; deepened CNN usage.
- **VGGNet**: Used very small filters but deeper networks.
- **ResNet**: Introduced **skip connections** to train very deep networks without vanishing gradients.

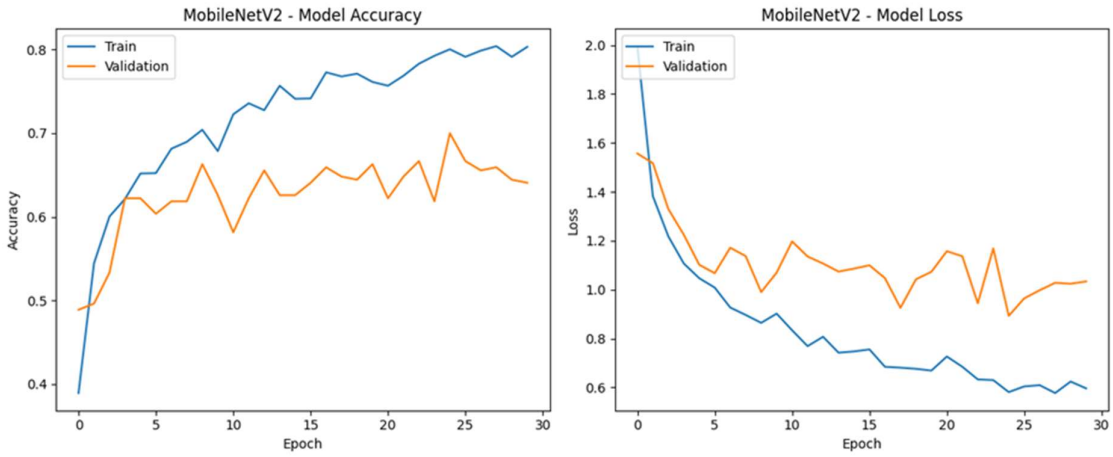
1.7.9 Results and discussion

Result of 4 different models.

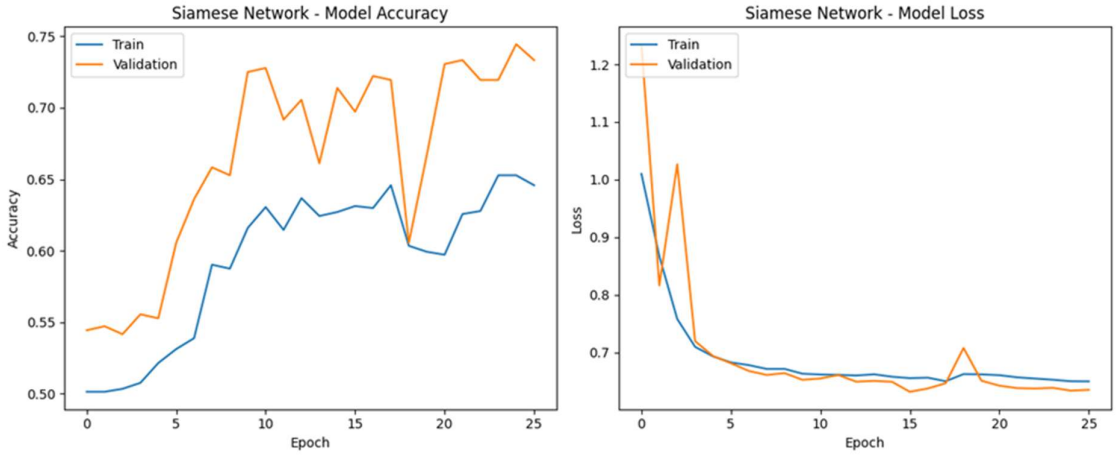
S.no	Model	Epochs	Training Time	Accuracy
1.	MobileNetV2	100	1253.58s	89.78%
2.	Siamese FSL	100	1026.8s	57.66%
3.	GNN	100	12m 41.s	33.33%
4.	Vision Transformer	100	1382.53s	54.44%

Table 1.7.9.1 Accuracy of 4 different models

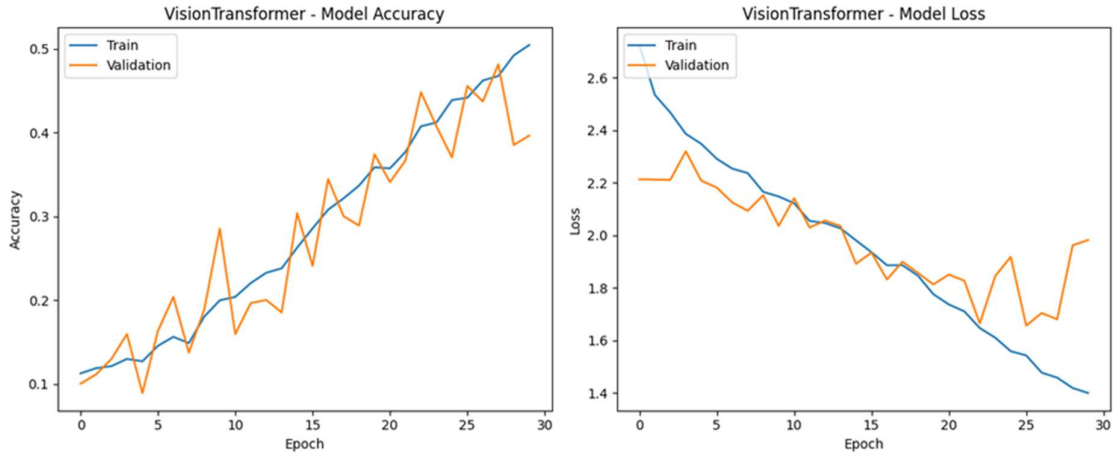
MobilenetV2 Training History



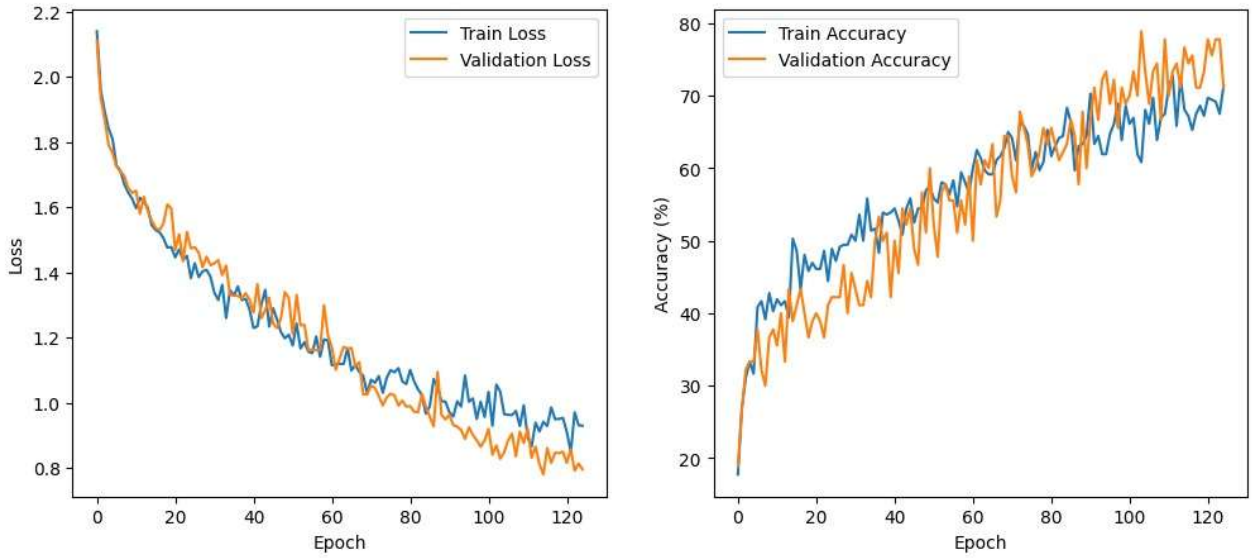
Siamese Network Training History



Visual Transformer Training History



Graphical Neural Network Training History



CHAPTER 2

MERITS AND DEMERITS OF THE BASE PAPER

Here are the merits and demerits of the paper titled "Pest classification: Explainable few-shot learning vs. convolutional neural networks vs. transfer learning":

2.1 Merits

1. Novelty and Relevance

- First to integrate Explainable Few-Shot Learning (FSL) with Grad-CAM for pest classification, addressing both accuracy and interpretability.
- Tackles the real-world challenge of pest detection in data-scarce agricultural settings, which is highly relevant to smart farming.

2. Comprehensive Methodology

- Compares four different approaches: Prototypical Networks, Siamese Networks (both FSL), CNN, and Transfer Learning.
- Evaluated on three datasets: Full Pest Images, Half Pest Images, and Malaysian Pest Images — including both complete and partial pest images, adding robustness.

3. High Accuracy with Low Data

Prototypical Network achieved up to 99.81% accuracy with just 5 shots, outperforming traditional methods like CNN and Transfer Learning, which required significantly more data.

4. Explainability

Integrates Grad-CAM to visualize model decisions, improving transparency and trust, which is essential in real-world agricultural applications.

5. Data Augmentation and Preprocessing

Uses thoughtful augmentation and preprocessing (e.g., rotation, resizing), improving model generalizability and performance under varied conditions.

6. Practical Impact

Aligned with SDG goals and Agenda 2063, emphasizing the societal and environmental relevance of the research.

2.2 Demerits

1. Dataset Limitations

Relies on relatively small and domain-specific datasets (e.g., from Kaggle and Malaysian pest images). These may not generalize well to global pest varieties or unseen species.

2. Lack of Broader Model Variety

Focused only on Prototypical and Siamese Networks for FSL. Other promising FSL models like MAML, Matching Networks, or Relation Networks were mentioned but not explored.

3. Limited Analysis of Grad-CAM Outputs

While Grad-CAM is used, the paper lacks quantitative analysis or human validation of the heatmaps for verifying the correctness of the explanations.

4. Performance Trade-offs Unclear

The paper emphasizes high accuracy, but model training times, resource requirements, or efficiency trade-offs (especially when adding explainability) aren't deeply discussed.

5. Siamese Network Underperformance

While included as part of the study, Siamese Networks had comparatively lower performance, especially with half images highlighting its limited applicability.

6. Scalability and Real-World Deployment

There’s no discussion on deployment feasibility, such as on mobile devices, drones, or IoT systems, which are common in smart farming setups.

CHAPTER 3

SOURCE CODE

Creating Best Model

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import cv2
import os
import tensorflow as tf
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, Flatten, MaxPooling2D, BatchNormalization,
Dropout, Input, Lambda, Conv2D, GlobalAveragePooling2D
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications import MobileNetV2
import tensorflow.keras.backend as K
import random
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
import torch.optim as optim
from torch_geometric.nn import GCNConv, GATConv, global_mean_pool
from skimage.segmentation import slic
from skimage.util import img_as_float
from torch_geometric.data import Data, DataLoader
import time
import warnings
warnings.filterwarnings('ignore')

# Global Variables
train_dir = 'Dataset/pest/train'
```

```
test_dir = 'Dataset/pest/test'

# Define the necessary global variables if not already defined
IMG_SIZE = (224, 224)
BATCH_SIZE = 32
train_dir = 'Dataset/pest/train'
test_dir = 'Dataset/pest/test'

# Image dimensions
IMG_SIZE = (224, 224)
BATCH_SIZE = 32

# Common data loading for TensorFlow models
def load_tf_data(shuffle=True):
    train_datagen = tf.keras.preprocessing.image.ImageDataGenerator(
        zca_epsilon=1e-06,
        rotation_range=30,
        width_shift_range=0.1,
        height_shift_range=0.2,
        shear_range=20,
        zoom_range=0.8,
        fill_mode="nearest",
        horizontal_flip=True,
        vertical_flip=True,
        validation_split=0.1,
        rescale=1./255
    )
    test_datagen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255)
    training = train_datagen.flow_from_directory(
        train_dir,
        batch_size=BATCH_SIZE,
        target_size=IMG_SIZE,
        subset="training",
        shuffle=shuffle
```

```
    )
    validing = train_datagen.flow_from_directory(
        train_dir,
        batch_size=BATCH_SIZE,
        target_size=IMG_SIZE,
        subset='validation',
        shuffle=shuffle
    )
    testing = test_datagen.flow_from_directory(
        test_dir,
        batch_size=BATCH_SIZE,
        target_size=IMG_SIZE,
        shuffle=shuffle
    )
    num_classes = len(training.class_indices)
    class_labels = list(training.class_indices.keys())
    return training, validing, testing, num_classes, class_labels

# Utility function to plot and save training history
def plot_training_history(history, model_name):
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
    plt.title(f'{model_name} - Model Accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(['Train', 'Validation'], loc='upper left')
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title(f'{model_name} - Model Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
```

```
    plt.legend(['Train', 'Validation'], loc='upper left')
    plt.tight_layout()
    plt.savefig(f'{model_name}_training_history.png')
    plt.close()

# Utility function to evaluate and visualize predictions
def visualize_prediction(model, img_path, class_labels, model_name, is_tf_model=True):
    if not os.path.exists(img_path):
        print(f"Warning: Test image path {img_path} not found. Skipping prediction.")
        return
    # Load and display the test image
    test_img = cv2.imread(img_path)
    test_img = cv2.resize(test_img, IMG_SIZE)
    plt.figure(figsize=(6, 6))
    plt.imshow(cv2.cvtColor(test_img, cv2.COLOR_BGR2RGB))
    plt.title("Test Image")
    plt.axis('off')
    plt.savefig(f'{model_name}_test_image.png')
    plt.close()
    if is_tf_model:
        # For TensorFlow models
        img_array = image.img_to_array(image.load_img(img_path, target_size=IMG_SIZE))
        img_array = np.expand_dims(img_array, axis=0) / 255.0
        prediction = model.predict(img_array)
        predicted_class = np.argmax(prediction[0])
        predicted_label = class_labels[predicted_class]
        # Plot prediction probabilities
        plt.figure(figsize=(10, 5))
        plt.bar(class_labels, prediction[0])
        plt.xlabel("Class")
        plt.ylabel("Probability")
        plt.title(f"{model_name} - Prediction: {predicted_label}")
        plt.xticks(rotation=45)
        plt.tight_layout()
```

```
        plt.savefig(f'{model_name}_prediction.png')
        plt.close()
        print(f"{model_name} Predicted Class: {predicted_label}")
    else:
        # For PyTorch models (handled differently by each model's predict function)
        print(f"{model_name} prediction visualization is handled by the model's own function")

# ----- MODEL 1: MobileNetV2 Transfer Learning -----
def build_mobilenetv2_model(num_classes):
    base_model = MobileNetV2(
        weights='imagenet',
        include_top=False,
        input_shape=(IMG_SIZE[0], IMG_SIZE[1], 3)
    )

    # Freeze the base model layers
    for layer in base_model.layers:
        layer.trainable = False

    model = Sequential([
        base_model,
        GlobalAveragePooling2D(),
        Dense(512, activation='relu'),
        BatchNormalization(),
        Dropout(0.5),
        Dense(128, activation='relu'),
        BatchNormalization(),
        Dropout(0.3),
        Dense(num_classes, activation='softmax')
    ])

    model.compile(
        optimizer=Adam(0.001),
        loss='categorical_crossentropy',
```

```
        metrics=['accuracy']
    )

    return model

def train_mobilenetv2(training, validing, testing, num_classes):
    print("\n----- Training MobileNetV2 Transfer Learning Model -----")
    model = build_mobilenetv2_model(num_classes)

    callbacks = [
        EarlyStopping(patience=100, restore_best_weights=True, verbose=1),
        ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=250, verbose=1)
    ]

    start_time = time.time()
    history = model.fit(
        training,
        validation_data=validing,
        epochs=500,
        callbacks=callbacks,
        verbose=1
    )
    training_time = time.time() - start_time

    # Evaluate on test data
    test_loss, test_acc = model.evaluate(testing, verbose=0)
    print(f"MobileNetV2 Test Accuracy: {test_acc * 100:.2f}%")
    print(f"Training time: {training_time:.2f} seconds")

    # Save model and plot training history
    model.save("mobilenetv2_model.h5")
    plot_training_history(history, "MobileNetV2")

    return model, test_acc, training_time
```

```
# ----- MODEL 2: Siamese Network -----
# Euclidean distance function
def euclidean_distance(vectors):
    x, y = vectors
    sum_square = K.sum(K.square(x - y), axis=1, keepdims=True)
    return K.sqrt(K.maximum(sum_square, K.epsilon()))

def build_siamese_base():
    base_model = Sequential([
        Conv2D(32, (3, 3), activation='relu', input_shape=(IMG_SIZE[0], IMG_SIZE[1], 3)),
        MaxPooling2D(2, 2),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D(2, 2),
        Conv2D(128, (3, 3), activation='relu'),
        MaxPooling2D(2, 2),
        Flatten(),
        Dense(128, activation='relu'),
        BatchNormalization(),
        Dropout(0.3)
    ])
    return base_model

def build_siamese_model():
    base_network = build_siamese_base()

    input_a = Input(shape=(IMG_SIZE[0], IMG_SIZE[1], 3))
    input_b = Input(shape=(IMG_SIZE[0], IMG_SIZE[1], 3))

    processed_a = base_network(input_a)
    processed_b = base_network(input_b)

    distance = Lambda(euclidean_distance)([processed_a, processed_b])
    output = Dense(1, activation='sigmoid')(distance)
```

```
    siamese_model = Model(inputs=[input_a, input_b], outputs=output)
    siamese_model.compile(optimizer=Adam(0.001), loss='binary_crossentropy',
metrics=['accuracy'])
    return siamese_model

def create_pairs(images_by_class, num_pairs_per_class=100):
    pairs = []
    labels = []

    # Same class pairs (label=1)
    for class_images in images_by_class.values():
        if len(class_images) < 2:
            continue

        for _ in range(num_pairs_per_class):
            # Randomly select two images from the same class
            idx1, idx2 = random.sample(range(len(class_images)), 2)
            pairs.append([class_images[idx1], class_images[idx2]])
            labels.append(1)  # 1 indicates same class

    # Different class pairs (label=0)
    classes = list(images_by_class.keys())
    for _ in range(num_pairs_per_class * len(classes)):
        # Select two different classes
        class1, class2 = random.sample(classes, 2)

        # Select random images from these classes
        img1 = random.choice(images_by_class[class1])
        img2 = random.choice(images_by_class[class2])

        pairs.append([img1, img2])
        labels.append(0)  # 0 indicates different classes
```

```
    return np.array(pairs), np.array(labels)

def load_images_from_directory(directory):
    images_by_class = {}
    class_indices = {}

    # Get all subdirectories (classes)
    classes = [d for d in os.listdir(directory) if os.path.isdir(os.path.join(directory, d))]

    for i, class_name in enumerate(classes):
        class_indices[class_name] = i
        class_dir = os.path.join(directory, class_name)
        images_by_class[class_name] = []

        # Get all image files (limit to 50 per class to manage memory)
        image_files = [f for f in os.listdir(class_dir) if f.endswith(('.jpg', '.jpeg', '.png'))][:50]

        for img_file in image_files:
            img_path = os.path.join(class_dir, img_file)
            img = cv2.imread(img_path)
            if img is None:
                continue

            img = cv2.resize(img, IMG_SIZE)
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            img = img / 255.0  # Normalize to [0, 1]
            images_by_class[class_name].append(img)

    return images_by_class, class_indices

def prepare_pairs_for_model(pairs):
    left_input = []
    right_input = []
```

```
    for pair in pairs:
        left_input.append(pair[0])
        right_input.append(pair[1])

    return [np.array(left_input), np.array(right_input)]

def classify_with_siamese(siamese_model, reference_images_by_class, test_image,
threshold=0.5):
    results = {}

    # For each class, compare the test image with all reference images
    for class_name, ref_images in reference_images_by_class.items():
        similarities = []

        for ref_img in ref_images:
            # Prepare the pair
            pair = prepare_pairs_for_model([[test_image, ref_img]])

            # Get similarity prediction (1 = same class, 0 = different class)
            similarity = siamese_model.predict(pair, verbose=0)[0][0]
            similarities.append(similarity)

        # Average similarity for this class
        avg_similarity = np.mean(similarities)
        results[class_name] = avg_similarity

    # Find the class with highest similarity
    predicted_class = max(results, key=results.get)
    confidence = results[predicted_class]

    # Only classify if confidence is above threshold
    if confidence > threshold:
        return predicted_class, results
    else:
```

```
        return "Unknown", results

def train_siamese_network():
    print("\n----- Training Siamese Network Model -----")
    # Load images for training
    train_images_by_class, class_indices = load_images_from_directory(train_dir)
    print(f"Found {len(class_indices)} classes")
    # Create pairs for training
    print("Creating image pairs for training...")
    pairs, labels = create_pairs(train_images_by_class)
    # Split into train and validation
    train_pairs, val_pairs, train_labels, val_labels = train_test_split(
        pairs, labels, test_size=0.2, random_state=42, shuffle=True
    )
    # Prepare data for model
    train_pair_data = prepare_pairs_for_model(train_pairs)
    val_pair_data = prepare_pairs_for_model(val_pairs)
    # Build and train model
    siamese_model = build_siamese_model()
    # Define callbacks
    early_stopping = EarlyStopping(monitor='val_loss', patience=100,
restore_best_weights=True, verbose=1)
    reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=250, min_lr=1e-
6, verbose=1)
    # Train the model
    start_time = time.time()
    history = siamese_model.fit(
        train_pair_data, train_labels,
        validation_data=(val_pair_data, val_labels),
        epochs=500,
        batch_size=BATCH_SIZE,
        callbacks=[early_stopping, reduce_lr],
        verbose=1
    )
```

```
training_time = time.time() - start_time
# Save model
siamese_model.save("siamese_model.h5")
# Plot training history
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Siamese Network - Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Siamese Network - Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')

plt.tight_layout()
plt.savefig('siamese_training_history.png')
plt.close()

# Evaluate on test data
print("Loading test images for evaluation...")
test_images_by_class, _ = load_images_from_directory(test_dir)

# Create pairs for testing
print("Creating image pairs for testing...")
test_pairs, test_labels = create_pairs(test_images_by_class, num_pairs_per_class=50)
test_pair_data = prepare_pairs_for_model(test_pairs)

# Evaluate
```

```
test_loss, test_acc = siamese_model.evaluate(test_pair_data, test_labels, verbose=0)
print(f"Siamese Network Test Accuracy: {test_acc * 100:.2f}%")
print(f"Training time: {training_time:.2f} seconds")

# Test on a single image
img_test_path = 'Dataset/pest/test/beetle/jpg_33.jpg'
if os.path.exists(img_test_path):
    # Load and preprocess the test image
    img = cv2.imread(img_test_path)
    img = cv2.resize(img, IMG_SIZE)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = img / 255.0

    # Classify the test image
    predicted_class, class_similarities = classify_with_siamese(
        siamese_model, train_images_by_class, img
    )

    print(f"Siamese Network Predicted Class: {predicted_class}")

    # Plot class similarities
    plt.figure(figsize=(10, 5))
    classes = list(class_similarities.keys())
    similarities = list(class_similarities.values())

    plt.bar(classes, similarities)
    plt.xlabel("Class")
    plt.ylabel("Similarity Score")
    plt.title("Siamese Network - Class Similarity Scores")
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.savefig('siamese_class_similarities.png')
    plt.close()
```

```
    return siamese_model, test_acc, training_time

# ----- MODEL 3: GNN Model -----
# Only run if PyTorch and PyTorch Geometric are available
def train_gnn_model():
    try:
        print("\n----- Training Graph Neural Network Model -----")
        # Function to convert an image into a graph
        def image_to_graph(img_path, label):
            img = cv2.imread(img_path)
            if img is None:
                return None

            img = cv2.resize(img, IMG_SIZE)
            img = img_as_float(img)

            try:
                segments = slic(img, n_segments=100, compactness=10)

                nodes = np.unique(segments)
                features = []

                for node in nodes:
                    mask = segments == node
                    if np.sum(mask) > 0:
                        mean_color = np.mean(img[mask], axis=0)
                        std_color = np.std(img[mask], axis=0)
                        features.append(np.concatenate([mean_color, std_color]))

                features = np.array(features)

                # Create spatial edge connections
                edges = []
                for i in range(len(nodes)):
```

```
            for j in range(i + 1, len(nodes)):
                if are_segments_adjacent(segments, nodes[i], nodes[j]):
                    edges.append([i, j])
                    edges.append([j, i])

        if len(edges) == 0:  # Fallback if no adjacency detected
            for i in range(len(nodes)):
                for j in range(i + 1, min(i + 5, len(nodes))):
                    edges.append([i, j])
                    edges.append([j, i])

        edges = torch.tensor(edges, dtype=torch.long).t().contiguous()
        features = torch.tensor(features, dtype=torch.float)

        # Create a single label for the whole graph
        y = torch.tensor(label, dtype=torch.long)

        return Data(x=features, edge_index=edges, y=y)

    except Exception as e:
        print(f"Error processing image: {e}")
        return None

# Check if two segments are adjacent
def are_segments_adjacent(segments, seg1, seg2):
    mask1 = segments == seg1
    mask2 = segments == seg2

    kernel = np.ones((3, 3), np.uint8)
    dilated_mask1 = cv2.dilate(mask1.astype(np.uint8), kernel, iterations=1)

    return np.any(dilated_mask1 & mask2)

# GNN Model
```

```
class GNNModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, dropout_rate=0.3):
        super(GNNModel, self).__init__()
        self.conv1 = GATConv(input_dim, hidden_dim)
        self.conv2 = GATConv(hidden_dim, hidden_dim)
        self.batch_norm1 = nn.BatchNorm1d(hidden_dim)
        self.batch_norm2 = nn.BatchNorm1d(hidden_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, data):
        x, edge_index, batch = data.x, data.edge_index, data.batch

        if batch is None:
            batch = torch.zeros(x.size(0), dtype=torch.long, device=x.device)

        x = self.conv1(x, edge_index)
        x = self.batch_norm1(x)
        x = self.relu(x)
        x = self.dropout(x)

        x = self.conv2(x, edge_index)
        x = self.batch_norm2(x)
        x = self.relu(x)
        x = self.dropout(x)

        x = global_mean_pool(x, batch)
        x = self.fc(x)

        return x

# Function to create dataset from directory
def create_dataset(root_dir, class_indices):
```

```
    dataset = []
    class_counts = {}

    for class_name, idx in class_indices.items():
        class_dir = os.path.join(root_dir, class_name)
        if not os.path.isdir(class_dir):
            continue

        image_files = [f for f in os.listdir(class_dir) if f.endswith(('.jpg', '.jpeg', '.png'))]

        max_images = 50
        selected_files = image_files[:min(max_images, len(image_files))]
        class_counts[class_name] = len(selected_files)

        for img_file in selected_files:
            img_path = os.path.join(class_dir, img_file)
            graph_data = image_to_graph(img_path, idx)
            if graph_data is not None:
                dataset.append(graph_data)

    print(f"Dataset creation summary: {class_counts}")
    return dataset

# Get class indices from TensorFlow's ImageDataGenerator for consistency
datagen = tf.keras.preprocessing.image.ImageDataGenerator(rescale=1./255)
temp_gen = datagen.flow_from_directory(
    train_dir,
    batch_size=1,
    target_size=IMG_SIZE,
    shuffle=True
)
class_indices = temp_gen.class_indices
num_classes = len(class_indices)
```

```
    # Create datasets
    print("Creating training dataset...")
    train_dataset = create_dataset(train_dir, class_indices)
    print("Creating testing dataset...")
    test_dataset = create_dataset(test_dir, class_indices)

    if not train_dataset:
        raise ValueError("No training data could be created.")

    # Split training data into train and validation
    train_data, val_data = train_test_split(train_dataset, test_size=0.2, random_state=42,
shuffle=True)

    # Create data loaders
    train_loader = DataLoader(train_data, batch_size=8, shuffle=True)
    val_loader = DataLoader(val_data, batch_size=8, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=8, shuffle=True)

    # Get feature dimension from data
    input_dim = train_data[0].x.shape[1] if train_data else 6

    # Create and train model
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f"Using device: {device}")
    model = GNNModel(input_dim=input_dim, hidden_dim=64,
output_dim=num_classes).to(device)

    optimizer = optim.Adam(model.parameters(), lr=0.1, weight_decay=1e-4)
    criterion = nn.CrossEntropyLoss()
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5,
patience=5)

    # Training loop with early stopping
    best_val_loss = float('inf')
```

```
patience = 100
counter = 0
early_stop = False
epochs = 500

train_losses = []
val_losses = []
train_accs = []
val_accs = []

start_time = time.time()

for epoch in range(epochs):
    if early_stop:
        print("Early stopping triggered!")
        break

    # Training
    model.train()
    total_loss = 0
    correct = 0
    total = 0

    for batch in train_loader:
        batch = batch.to(device)
        optimizer.zero_grad()

        try:
            output = model(batch)
            loss = criterion(output, batch.y)
            loss.backward()

            torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

```
            optimizer.step()

            total_loss += loss.item()
            _, predicted = output.max(dim=1)
            total += batch.y.size(0)
            correct += predicted.eq(batch.y).sum().item()
        except Exception as e:
            continue

    if total > 0:
        train_loss = total_loss / len(train_loader)
        train_acc = 100.0 * correct / total
        train_losses.append(train_loss)
        train_accs.append(train_acc)
    else:
        continue

    # Validation
    model.eval()
    val_loss = 0
    correct = 0
    total = 0

    with torch.no_grad():
        for batch in val_loader:
            batch = batch.to(device)
            try:
                output = model(batch)
                loss = criterion(output, batch.y)

                val_loss += loss.item()
                _, predicted = output.max(dim=1)
                total += batch.y.size(0)
                correct += predicted.eq(batch.y).sum().item()
```

```
            except Exception as e:
                continue

    if total > 0:
        val_loss = val_loss / len(val_loader)
        val_acc = 100.0 * correct / total
        val_losses.append(val_loss)
        val_accs.append(val_acc)

        scheduler.step(val_loss)

        print(f"Epoch {epoch+1}/{epochs}, "
              f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}%, "
              f"Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.2f}%")

        if val_loss < best_val_loss:
            best_val_loss = val_loss
            counter = 0
            torch.save(model.state_dict(), "best_gnn_model.pth")
        else:
            counter += 1
            if counter >= patience:
                early_stop = True
    else:
        print("Warning: No valid batches in validation epoch")

training_time = time.time() - start_time

# Plot training curves
if train_losses and val_losses:
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Validation Loss')
```

```
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(train_accs, label='Train Accuracy')
    plt.plot(val_accs, label='Validation Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy (%)')
    plt.legend()
    plt.savefig('gnn_training_curves.png')
    plt.close()

# Evaluate best model on test set
model.load_state_dict(torch.load("best_gnn_model.pth"))
model.eval()
correct = 0
total = 0

idx_to_class = {v: k for k, v in class_indices.items()}

with torch.no_grad():
    for batch in test_loader:
        batch = batch.to(device)
        try:
            output = model(batch)
            _, predicted = output.max(dim=1)

            total += batch.y.size(0)
            correct += predicted.eq(batch.y).sum().item()
        except Exception as e:
            continue

if total > 0:
```

```
            test_acc = 100.0 * correct / total
            print(f"GNN Test Accuracy: {test_acc:.2f}%")
        else:
            test_acc = 0
            print("Warning: No valid batches in test evaluation")

        print(f"Training time: {training_time:.2f} seconds")

        return model, test_acc, training_time
    except Exception as e:
        print(f"Could not train GNN model: {e}")
        return None, 0, 0

class PatchExtractor(tf.keras.layers.Layer):
    def __init__(self, patch_size):
        super(PatchExtractor, self).__init__()
        self.patch_size = patch_size

    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(
            images=images,
            sizes=[1, self.patch_size, self.patch_size, 1],
            strides=[1, self.patch_size, self.patch_size, 1],
            rates=[1, 1, 1, 1],
            padding="VALID"
        )
        patch_dims = patches.shape[-1]
        patches = tf.reshape(patches, [batch_size, -1, patch_dims])
        return patches

# ----- MODEL 4: Vision Transformer -----
def build_vit_model(num_classes, image_size=224, patch_size=16, num_heads=8,
                    transformer_layers=8, hidden_units=64, mlp_units=128):
```

```
input_shape = (image_size, image_size, 3)
num_patches = (image_size // patch_size) ** 2
projection_dim = hidden_units
# Input layer
inputs = tf.keras.layers.Input(shape=input_shape)
# Create patches and project them
patches = PatchExtractor(patch_size)(inputs)
patch_projection = tf.keras.layers.Dense(projection_dim)(patches)
# Add positional embeddings
positions = tf.range(start=0, limit=num_patches, delta=1)
position_embedding = tf.keras.layers.Embedding(
    input_dim=num_patches, output_dim=projection_dim
)(positions)
encoded_patches = patch_projection + position_embedding
# Create transformer blocks
for _ in range(transformer_layers):
    # Layer normalization 1
    x1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    # Multi-head attention
    attention_output = tf.keras.layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=projection_dim // num_heads, dropout=0.1
    )(x1, x1)

    # Skip connection 1
    x2 = tf.keras.layers.Add()([attention_output, encoded_patches])

    # Layer normalization 2
    x3 = tf.keras.layers.LayerNormalization(epsilon=1e-6)(x2)

    # MLP
    x4 = tf.keras.layers.Dense(mlp_units, activation="gelu")(x3)
    x4 = tf.keras.layers.Dropout(0.1)(x4)
    x4 = tf.keras.layers.Dense(projection_dim)(x4)
    x4 = tf.keras.layers.Dropout(0.1)(x4)
```

```
        # Skip connection 2
        encoded_patches = tf.keras.layers.Add()([x4, x2])

    # Final layer normalization and global pooling
    representation = tf.keras.layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    representation = tf.keras.layers.GlobalAveragePooling1D()(representation)
    # Classification head
    representation = tf.keras.layers.Dropout(0.3)(representation)
    features = tf.keras.layers.Dense(256, activation="relu")(representation)
    features = tf.keras.layers.BatchNormalization()(features)
    features = tf.keras.layers.Dropout(0.3)(features)
    outputs = tf.keras.layers.Dense(num_classes, activation="softmax")(features)
    # Create the model
    model = tf.keras.Model(inputs=inputs, outputs=outputs)
    # Compile the model
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.1),
        loss=tf.keras.losses.CategoricalCrossentropy(),
        metrics=["accuracy"]
    )

    return model

def train_vit_model(training, validing, testing, num_classes):
    print("\n----- Training Vision Transformer Model -----")
    model = build_vit_model(
        num_classes=num_classes,
        image_size=IMG_SIZE[0],
        patch_size=16,
        num_heads=8,
        transformer_layers=6,
        hidden_units=64,
        mlp_units=128
```

```
    )
    callbacks = [
        EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True, verbose=1),
        ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=5, min_lr=1e-6,
verbose=1)
    ]
    start_time = time.time()
    history = model.fit(
        training,
        validation_data=validing,
        epochs=500,
        callbacks=callbacks,
        verbose=1
    )
    training_time = time.time() - start_time
    # Evaluate on test data
    test_loss, test_acc = model.evaluate(testing, verbose=0)
    print(f"Vision Transformer Test Accuracy: {test_acc * 100:.2f}%")
    print(f"Training time: {training_time:.2f} seconds")
    # Save model and plot training history
    model.save("vit_model.h5")
    plot_training_history(history, "VisionTransformer")
    return model, test_acc, training_time

# Main function to run all models and compare results
def main():
    # Load data
    training, validing, testing, num_classes, class_labels = load_tf_data()
    print(f"Number of classes: {num_classes}")
    print(f"Class labels: {class_labels}")
    # Dictionary to store results
    results = {}
    # Train MobileNetV2 model
    mobilenetv2_model, mobilenetv2_acc, mobilenetv2_time = train_mobilenetv2(training,
```

```
validing, testing, num_classes)
    results["MobileNetV2"] = {"accuracy": mobilenetv2_acc, "training_time":
mobilenetv2_time}
    # Visualize prediction for a sample image
    sample_img = 'Dataset/pest/test/beetle/jpg_33.jpg'
    if os.path.exists(sample_img):
        visualize_prediction(mobilenetv2_model, sample_img, class_labels, "MobileNetV2")
    # Train Siamese Network
    siamese_model, siamese_acc, siamese_time = train_siamese_network()
    results["Siamese"] = {"accuracy": siamese_acc, "training_time": siamese_time}
    # Train GNN model if possible
    try:
        gnn_model, gnn_acc, gnn_time = train_gnn_model()
        if gnn_model is not None:
            results["GNN"] = {"accuracy": gnn_acc, "training_time": gnn_time}
    except Exception as e:
        print(f"Could not train GNN model: {e}")

    # Train Vision Transformer model
    vit_model, vit_acc, vit_time = train_vit_model(training, validing, testing, num_classes)
    results["VisionTransformer"] = {"accuracy": vit_acc, "training_time": vit_time}

    if os.path.exists(sample_img):
        visualize_prediction(vit_model, sample_img, class_labels, "VisionTransformer")

    # Compare model performances
    print("\n----- Model Performance Comparison -----")
    for model_name, metrics in results.items():
        print(f"{model_name}: Accuracy = {metrics['accuracy'] * 100:.2f}%, Training Time =
{metrics['training_time']:.2f} seconds")
    # Plot comparison chart
    plt.figure(figsize=(12, 6))
    # Accuracy comparison
    plt.subplot(1, 2, 1)
```

```
    model_names = list(results.keys())
    accuracies = [results[model]["accuracy"] * 100 for model in model_names]
    plt.bar(model_names, accuracies, color='skyblue')
    plt.ylabel('Accuracy (%)')
    plt.title('Model Accuracy Comparison')
    plt.ylim([0, 100])

    # Training time comparison
    plt.subplot(1, 2, 2)
    training_times = [results[model]["training_time"] for model in model_names]
    plt.bar(model_names, training_times, color='salmon')
    plt.ylabel('Training Time (seconds)')
    plt.title('Model Training Time Comparison')

    plt.tight_layout()
    plt.savefig('model_comparison.png')
    plt.close()

    print("Comparison chart saved as 'model_comparison.png'")

if __name__ == "__main__":
    main()
```

Creating User Interface

```
import streamlit as st
import time
import cv2
import numpy as np
import json
import urllib.parse
import base64
from PIL import Image
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image as keras_image
```

```
from tensorflow.keras.applications.mobilenet_v2 import preprocess_input
import tensorflow as tf
import tempfile
import webbrowser

# Globals
SMS_THRESHOLD = 0.98
NOTIFICATION_PHONE_NUMBER = "917842624422"
model = None
class_labels = None

# Setup
st.set_page_config(page_title="Pest Detection", layout="wide")

# Utility functions
def send_sms_alert(pest_type, confidence):
    message = f" ALERT: {pest_type} detected with {confidence:.1%} confidence! Take
action immediately."
    encoded_message = urllib.parse.quote(message)
    url = f"https://wa.me/{NOTIFICATION_PHONE_NUMBER}?text={encoded_message}"
    webbrowser.open(url)
    st.success("Opened WhatsApp Web. Please click 'Send' manually.")

def load_class_labels():
    try:
        with open('class_labels.json', 'r') as f:
            return json.load(f)
    except FileNotFoundError:
        return ["aphids", "armyworm", "beetle", "bollworm", "grasshopper",
                "mites", "mosquito", "sawfly", "stem_borer"]

def load_detection_model():
    global model, class_labels
```

```
    try:
        model = load_model('mobilenetv2_model.h5')
    except:
        base_model = tf.keras.applications.MobileNetV2(
            input_shape=(224, 224, 3), include_top=False, weights='imagenet')
        x = tf.keras.layers.GlobalAveragePooling2D()(base_model.output)
        output = tf.keras.layers.Dense(9, activation='softmax')(x)
        model = tf.keras.Model(inputs=base_model.input, outputs=output)
    class_labels = load_class_labels()

def predict_from_image(img_data):
    img = cv2.resize(img_data, (224, 224))
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img_array = keras_image.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0)
    img_array = preprocess_input(img_array)

    predictions = model.predict(img_array, verbose=0)
    predicted_idx = np.argmax(predictions[0])
    predicted_class = class_labels[predicted_idx]
    confidence = float(predictions[0][predicted_idx])

    all_probs = sorted(
        [{"class": label, "probability": float(prob)}
         for label, prob in zip(class_labels, predictions[0])],
        key=lambda x: x["probability"],
        reverse=True
    )

    return predicted_class, confidence, all_probs

# Load model
load_detection_model()
```

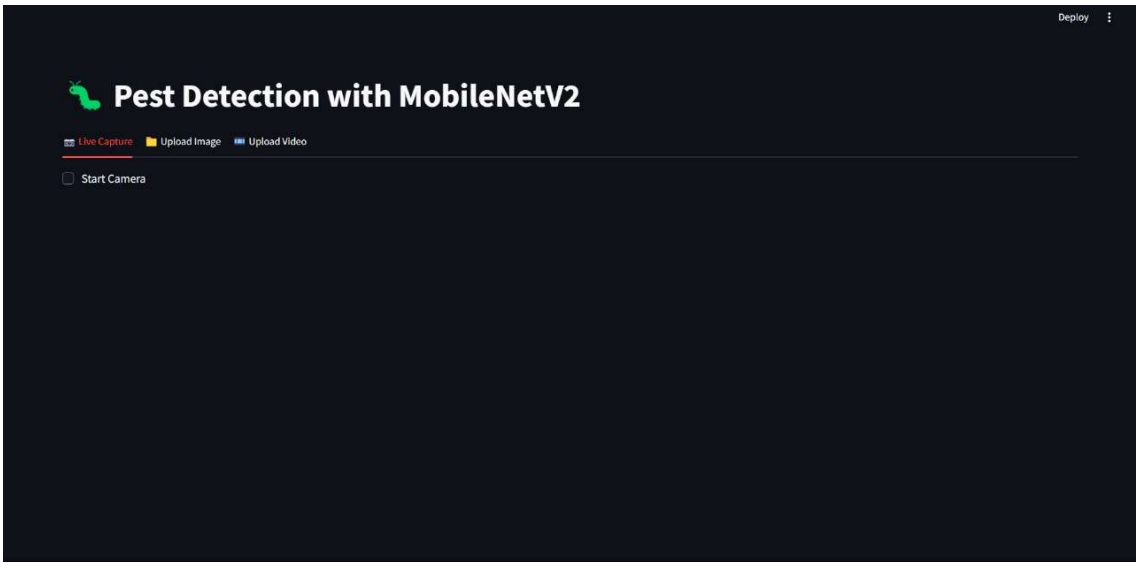
```
# UI
st.title("🐛 Pest Detection with MobileNetV2")
tab1, tab2, tab3 = st.tabs(["📷 Live Capture", "📁 Upload Image", "🎞 Upload Video"])

# Tab 1: Live Capture
with tab1:
    run = st.checkbox('Start Camera')
    frame_window = st.image([])
    pred_text = st.empty()  # Placeholder for predictions

    cap = cv2.VideoCapture(0)
    while run:
        success, frame = cap.read()
        if not success:
            st.error("Failed to capture from camera.")
            break

        pred_class, conf, probs = predict_from_image(frame)
        cv2.putText(frame, f"{pred_class}: {conf:.2f}", (10, 30),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)

        if conf > SMS_THRESHOLD:
            cv2.putText(frame, "ALERT SENT!", (10, 60),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 0, 255), 2)
            send_sms_alert(pred_class, conf)
            time.sleep(2)

        frame_window.image(frame, channels="BGR")

        # Show prediction probabilities
        pred_display = "### Prediction Probabilities:\n"
        for item in probs:
            pred_display += f"- **{item['class']}**: {item['probability']:.2%}\n"
```

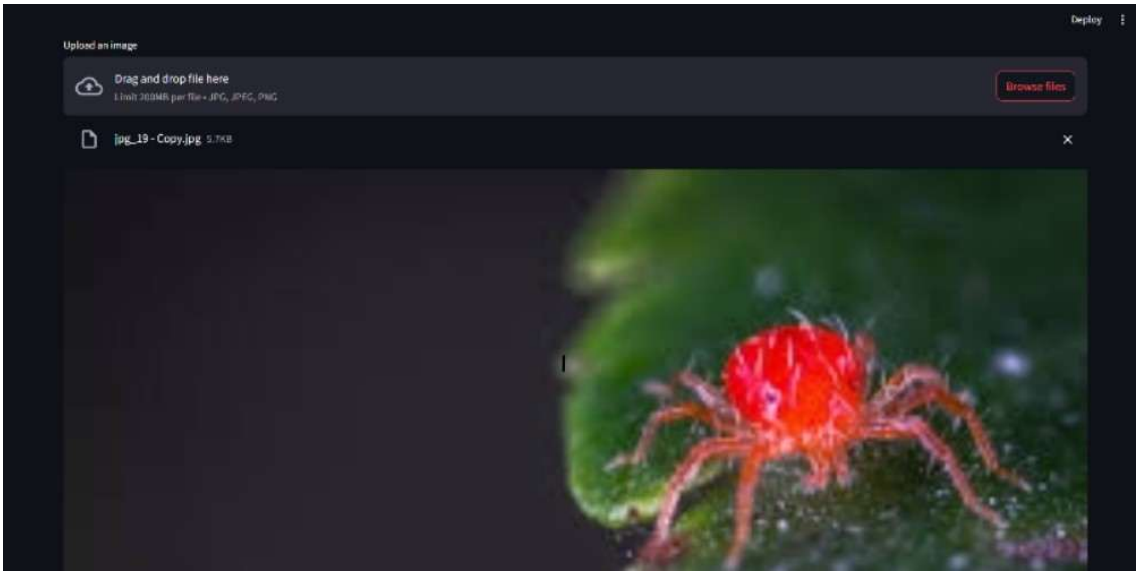
```
            pred_text.markdown(pred_display)

        cap.release()

    # Tab 2: Upload Image
    with tab2:
        uploaded = st.file_uploader("Upload an image", type=["jpg", "jpeg", "png"])
        if uploaded:
            img_pil = Image.open(uploaded).convert("RGB")
            img_np = np.array(img_pil)
            pred_class, conf, probs = predict_from_image(img_np)
            st.image(img_np, caption=f"{pred_class} ({conf:.2f})", use_column_width=True)
            st.write("### All Predictions:")
            for item in probs:
                st.write(f"{item['class']}: {item['probability']:.2f}")
            if conf > SMS_THRESHOLD:
                send_sms_alert(pred_class, conf)

    # Tab 3: Upload Video
    with tab3:
        st.subheader("Detect Pests in Uploaded Video")
        video_file = st.file_uploader("Upload a video (mp4)", type=["mp4"])
        if video_file:
            tfile = tempfile.NamedTemporaryFile(delete=False)
            tfile.write(video_file.read())
            cap = cv2.VideoCapture(tfile.name)
            frame_window = st.image([])
            pred_text = st.empty()  # Placeholder for displaying predictions

            alerted_classes = set()  # Track classes already alerted

            while cap.isOpened():
                ret, frame = cap.read()
                if not ret:
```

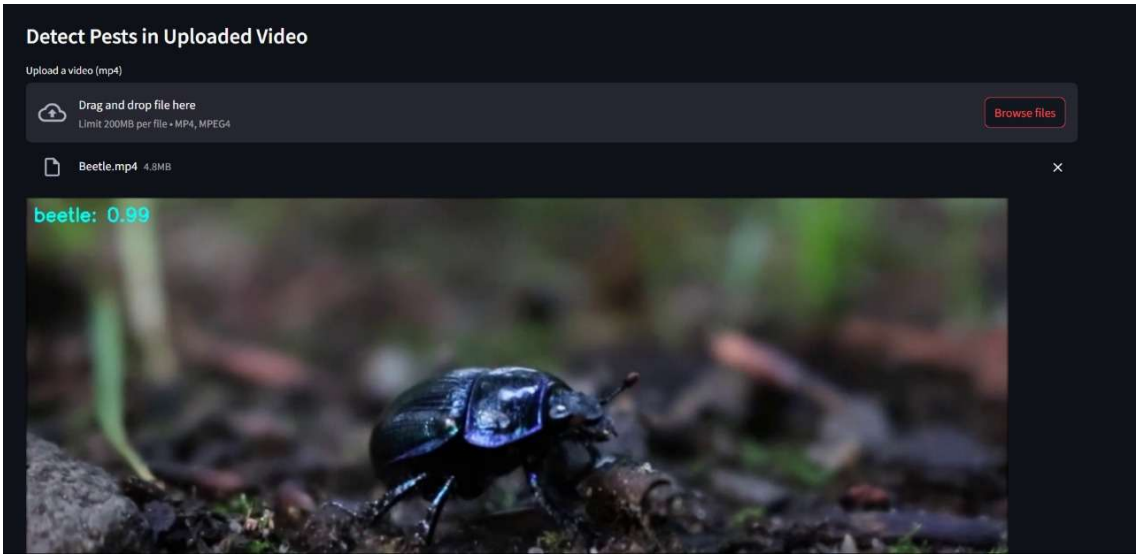
```
        break
    pred_class, conf, probs = predict_from_image(frame)
    cv2.putText(frame, f"{pred_class}: {conf:.2f}", (10, 30),
                cv2.FONT_HERSHEY_SIMPLEX, 0.8, (255, 255, 0), 2)

    # Send alert once per pest type if confidence > 95%
    if conf > 0.95 and pred_class not in alerted_classes:
        send_sms_alert(pred_class, conf)
        alerted_classes.add(pred_class)

    frame_window.image(frame, channels="BGR")

    # Show all class predictions
    pred_display = "### Prediction Probabilities:\n"
    for item in probs:
        pred_display += f"- **{item['class']}**: {item['probability']:.2%}\n"
    pred_text.markdown(pred_display)

cap.release()
```

CHAPTER 4

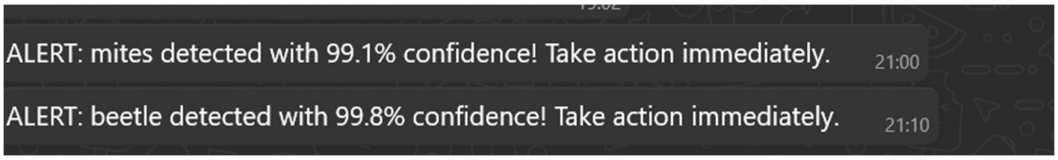
SNAPSHOTS






Detect Pests in Uploaded Video
Upload a video (mp4)
Drag and drop file here
Limit 200MB per file • MP4, MPEG4
Browse files
Beetle.mp4 4.8MB
beetle: 0.99


Prediction Probabilities:
beetle: 99.93%
stem_borer: 0.02%
mites: 0.02%
mosquito: 0.01%
armyworm: 0.01%
sawfly: 0.00%
grasshopper: 0.00%
aphids: 0.00%
bollworm: 0.00%


ALERT: mites detected with 99.1% confidence! Take action immediately.
21:00
ALERT: beetle detected with 99.8% confidence! Take action immediately.
21:10

CHAPTER 5

CONCLUSION AND FUTURE PLANS

Conclusion:

In this project, we designed and implemented a robust, real-time pest detection system tailored for agricultural applications, particularly in data-constrained and resource-limited environments. Our approach explored and compared several advanced machine learning techniques, including **Few-Shot Learning (FSL)** with Siamese Networks, **Graph Neural Networks (GNNs)**, **Vision Transformers (ViT)**, and **Self-Supervised Learning (SSL)** integrated with **MobileNetV2**. The system was built to support three types of input: **live video stream**, **image upload**, and **video file upload**, offering flexible modes of pest detection.

Real-time visual processing was achieved using **OpenCV**, while the frontend was developed using **Streamlit**, providing an interactive and user-friendly interface. For on-device inference and performance optimization, we used **TensorFlow Lite**, and **WhatsApp integration** was implemented for alert delivery to notify users instantly upon pest detection.

Despite successful implementation, we faced several practical challenges:

Hardware Constraints: Limited GPU/CPU access restricted the ability to train models from scratch, requiring reliance on pre-trained models such as MobileNet-SSD.

GNN Limitations: Image-to-graph transformation and GNN training required more computational power than available, limiting the depth of experimentation.

FSL Training Constraints: The inability to run multiple epochs affected the generalization performance of Few-Shot models.

WhatsApp Integration: Automated messaging required premium APIs, and we relied on semi-automated methods due to cost constraints.

Cloud and Deployment Issues: Hosting the solution online was hindered by infrastructure limitations and high-speed internet requirements.

Future Plans:

To address the current limitations and enhance the system's real-world applicability, the following future developments are planned:

Drone Camera Integration: Extend support to **drone-mounted cameras** for large-scale aerial pest monitoring across expansive agricultural lands.

Offline SMS Alert System with Twilio: Incorporate **Twilio SMS** alerts as an alternative to WhatsApp, ensuring timely notifications even in **low-connectivity rural areas**.

Optimized Edge Deployment: Further compress and optimize models for deployment on **low-power edge devices**, eliminating dependency on cloud infrastructure.

Fully Automated WhatsApp Alerts: Explore **open-source or affordable APIs** to enable fully **automated WhatsApp messaging** without manual intervention.

Hybrid Model Pipeline: Design a **hybrid inference pipeline** that intelligently switches between ViT, GNN, and FSL models based on the input type, dataset size, or device constraints to maximize detection accuracy and system efficiency.

CHAPTER 6
REFERENCES

[1] K.Thenmozhi, U. Srinivasulu Reddy, Crop pest classification based on deep convolutional neural network and transfer learning, https://doi.org/10.1016/j.compag.2019.104906

[2] Xi Cheng, Youhua Zhang, Yiqiong Chen, Yunzhi Wu, Yi Yue, Pest identification via deep residual learning in complex background, https://doi.org/10.1016/j.compag.2017.08.005

[3] Muhammad Attique Khan, Tallha Akram, Muhammad Sharif, Muhammad Awais, Kashif Javed, Hashim Ali, Tanzila Saba, CCDF: Automatic system for segmentation and recognition of fruit crops diseases based on correlation coefficient and deep CNN features, https://doi.org/10.1016/j.compag.2018.10.013

[4] Chengjun Xie, Jie Zhang, Rui Li, Jinyan Li, Peilin Hong, Junfeng Xia, Peng Chen, Automatic classification for field crop insects via multiple-task sparse representation and multiple-kernel learning, https://doi.org/10.1016/j.compag.2015.10.015

[5] Solemane Coulibaly, Bernard Kamsu-Foguem, Dantouma Kamissoko, Daouda Traore, Deep neural networks with transfer learning in millet crop images, https://doi.org/10.1016/j.compind.2019.02.003

[6] Xi Cheng, Youhua Zhang, Yiqiong Chen, Yunzhi Wu, Yi Yue, Pest identification via deep residual learning in complex background, https://doi.org/10.1016/j.compag.2017.08.005

[7] Zhong-Qiu Zhao, Lin-Hai Ma, Yiu-ming Cheung, Xindong Wu**,** Yuanyan Tang**,** Chun Lung Chen, ApLeaf: An efficient android-based plant leaf identification system, https://doi.org/10.1016/j.neucom.2014.02.077

[8] Adhi Setiawan, Novanto Yudistira, Randy Cahya Wihandika, Large scale pest classification using efficient Convolutional Neural Network with augmentation and regularizers, https://doi.org/10.1016/j.compag.2022.107204

[9] Loris Nanni, Gianluca Maguolo, Fabio Pancino, Insect pest image detection and recognition based on bio-inspired methods, https://doi.org/10.1016/j.ecoinf.2020.101089

[10] Fuji Jian, Sara Doak, Digvir S. Jayas, Paul G. Fields, Noel D.G. White, Comparison of insect detection efficiency by different detection methods, https://doi.org/10.1016/j.jspr.2016.07.008

[11] Bin Yang, Binghan Zhang, Yilong Han, Boda Liu, Jiniming Hu, Yiming Jin, Vision transformer-based visual language understanding of the construction process, https://doi.org/10.1016/j.aej.2024.05.015
[12] Om Uparkar, Jyoti Bharti, R.K. Pateriya, Rajeev Kumar Gupta, Ashutosh Sharma, Vision Transformer Outperforms Deep Convolutional Neural Network-based Model in Classifying X-ray Images, https://doi.org/10.1016/j.procs.2023.01.209
[13] Aiusha V Hujon, Thoudam Doren Singh, Khwairakpam Amitab, Transfer Learning Based Neural Machine Translation of English-Khasi on Low-Resource Settings, https://doi.org/10.1016/j.procs.2022.12.396
[14] Xiaoyan Jiang, Shuihua Wang, Yudong Zhang, Vision transformer promotes cancer diagnosis: A comprehensive review, https://doi.org/10.1016/j.eswa.2024.124113
[15] Junyu Chen, Eric C. Frey, Yufan He, William P. Segars, Ye Li, Yong Du, TransMorph: Transformer for unsupervised medical image registration, https://doi.org/10.1016/j.media.2022.102615
[16] Kyungjin Cho, Jeeyoung Kim, Ki Duk Kim, Sungju Park, Junsik Kim, Jihye Yun, Yura Ahn, Sang Young Oh, Sang Min Lee, Joon Beom Seo, Namkug Kim, MuSiC-ViT: A multi-task Siamese convolutional vision transformer for differentiating change from no-change in follow-up chest radiographs, https://doi.org/10.1016/j.media.2023.102894
[17] Madhuri Gokhale, Sraban Kumar Mohanty, Aparajita Ojha, GeneViT: Gene Vision Transformer with Improved DeepInsight for cancer classification, https://doi.org/10.1016/j.compbiomed.2023.106643
[18] Elyas Asadi Shamsabadi, Chang Xu, Aravinda S. Rao, Tuan Nguyen, Tuan Ngo, D. António Semblano Gouveia Dias-da-Costa, Vision transformer-based autonomous crack detection on asphalt and concrete surfaces, https://doi.org/10.1016/j.autcon.2022.104316
[19] T. Saranya, C. Deisy, S. Sridevi, Efficient agricultural pest classification using vision transformer with hybrid pooled multihead attention, https://doi.org/10.1016/j.compbiomed.2024.108584
[20] Zhenzhe Hechen, Wei Huang, Le Yin, Wenjing Xie, Yixin Zhao, Dilated-Windows-based Vision Transformer with Efficient-Suppressive-self-attention for insect pests classification, https://doi.org/10.1016/j.engappai.2023.107228
[21] Yan Gao, Haijiang Li, Weiqi Fu, Few-shot learning for image-based bridge damage detection, https://doi.org/10.1016/j.engappai.2023.107078
[22] Yinshuo Zhang, Lei Chen, Yuan Yuan, Few-shot agricultural pest recognition based on multimodal masked autoencoder, https://doi.org/10.1016/j.cropro.2024.106993
[23] João Vitor de Andrade Porto, Arlinda Cantero Dorsa, Vanessa Aparecida de Moraes Weber, Karla Rejane de Andrade Porto, Hemerson Pistori, Usage of few-shot learning and meta-learning in agriculture: A literature review, https://doi.org/10.1016/j.atech.2023.100307
[24] Yang Li, Jiachen Yang, Meta-learning baselines and database for few-shot classification in agriculture, https://doi.org/10.1016/j.compag.2021.106055
[25] Masoud Rezaei, Dean Diepeveen, Hamid Laga, Michael G.K. Jones, Ferdous Sohel, Plant disease recognition in a low data scenario using few-shot learning, https://doi.org/10.1016/j.compag.2024.108812
[26] Chunshan Wang, Ji Zhou, Chunjiang Zhao, Jiuxi Li, Guifa Teng, Huarui Wu, Few-shot

vegetable disease recognition model based on image text collaborative representation learning, https://doi.org/10.1016/j.compag.2021.106098
[27] Lucas M. Tassis, Renato A. Krohling, Few-shot learning for biotic stress classification of coffee leaves, https://doi.org/10.1016/j.aiia.2022.04.001
[28] Ben McEwen, Kaspar Soltero, Stefanie Gutschmidt, Andrew Bainbridge-Smith, James Atlas, Richard Green, Active few-shot learning for rare bioacoustic feature annotation, https://doi.org/10.1016/j.ecoinf.2024.102734
[29] Jinchao Pan, Limei Xia, Qiufeng Wu, Yixin Guo, Yiping Chen, Xiaole Tian, Automatic strawberry leaf scorch severity estimation via faster R-CNN and few-shot learning, https://doi.org/10.1016/j.ecoinf.2022.101706
[30] Fangming Zhong, Zhikui Chen, Yuchun Zhang, Feng Xia, Zero- and few-shot learning for diseases recognition of Citrus aurantium L. using conditional adversarial autoencoders, https://doi.org/10.1016/j.compag.2020.105828

CHAPTER 7

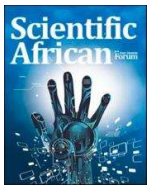
APPENDIX -BASE PAPER

Scienti c African 27 (2025) e02512

Contents lists available ScienceDire

Scientific African

journal www.elsevier.com/locate/sc



Check for updates

Pest classification: Explainable few-shot learning vs. convolutional neural networks vs. transfer learning

Nitiyaa Ragu [a], Jason Teo [b,c,*]

[a] Faculty of Computing and Informatics, Universiti Malaysia Sabah, Sabah, Malaysia

[b] Creative Advanced Machine Intelligence Research Centre, Faculty of Computing and Informatics, Universiti Malaysia Sabah, Kota Kinabalu, Malaysia [c] Evolutionary Computing Laboratory, Faculty of Computing and Informatics, Universiti Malaysia Sabah, Kota Kinabalu, Malaysia

ARTICLEINFO

Editor: DR B Gyampoh

Keywords:
Explainable few-shot learning
Prototypical network
Siamese network Pest detection
Smart agriculture

ABSTRACT

Pests significantly threaten plant yield and overall agricultural productivity, leading to reduced output in the farming industry. Accurate and automated detection of crop insect pests is crucial for effective pest control and optimal utilization of agricultural resources. This study addresses the problem of limited datasets in pest detection by exploring the potential of Explainable Few- Shot Learning (FSL), a machine learning approach that not only enables learning from a small amount of data but also provides interpretable insights into the decision-making process. Unlike traditional pest detection studies that rely on large labeled datasets or black-box models, this research introduces an advanced methodology by integrating explainability techniques such as Grad-CAM into FSL models, specifically Prototypical Network and Siamese Network. This dual approach ensures high accuracy with minimal training data while identifying key image features influencing predictions, thereby enhancing transparency and trust. A comparative analysis was conducted against Convolutional Neural Network (CNN) and transfer learning models using full pest images, half pest images, and Malaysian pest images. This study found that Explainable FSL achieved the highest accuracy of 99.81 % in various scenarios, including 9-way 1-shot, 3-shot, 5- shot, and 10-shot configurations, outperforming both CNN and transfer learning models. These findings demonstrate that Explainable FSL models can significantly improve the accuracy, transparency, and efficiency of pest detection systems, even with limited data. By advancing both the detection capabilities and interpretability of Artificial Intelligence (AI) systems, this research provides a novel contribution to smart agriculture, enabling robust pest detection systems tailored to real-world, data-scarce scenarios.

Introduction

Computer vision plays a crucial role in agriculture by identifying and detecting pests with machine vision equipment [1]. This technology replaces traditional methods that rely on the naked eye, allowing for advanced visual analysis techniques to recognize and categorize pests [2]. This is essential for preventing and controlling harmful infestations, as pests can cause significant damage to crops during growth, leading to diseases and substantial losses in agricultural income. Effective management is crucial to prevent pests from causing further harm.

* Corresponding author.

E-mail address: jtwteo@ums.edu.my (J. Teo).

https://doi.org/10.1016/j.sciaf.2024.e02512
Received 16 October 2023; Received in revised form 16 December 2024; Accepted 17 December 2024
Available online 21 December 2024

Agricultural pests have homogeneous physical characteristics, making it challenging to distinguish them using human observation. Traditional methods are time-consuming, inefficient, subjective, inaccurate, and costly [3]. Non-professional agricultural workers may struggle to accurately identify pest species, hindering effective pest management. Therefore, precise pest identification is crucial for agricultural productivity and advancing the field.

Recent advancements in pest detection have leveraged deep learning and machine learning methods. For instance, PestNet [4,5] introduced a two-stage approach for detecting and identifying insect pests using CNNs. It incorporated features like Channel-Spatial Attention (CSA), Region Proposal Network (RPN), and Position-Sensitive Score Map (PSSM) for enhanced feature extraction and classification. Other studies [6,7] explored traditional machine learning techniques, such as Support Vector Machine (SVM), k-Nearest Neighbor (KNN), and Naïve Bayes (NB), but these approaches often struggled with scalability and accuracy. Transfer learning has also been utilized; for example, Malathi and Gopinath [8] proposed a deep CNN-based model using ResNet-50 to recognize paddy pests, while Pattnaik et al. [9] applied DenseNet169 for classifying tomato pests.

Despite these advancements, existing methods often require large labeled datasets, which are not always available in agricultural settings. FSL addresses this limitation by enabling models to generalize from a small number of labeled examples. This method, characterized by Experience (E), Task (T), and Performance (P) [10], can significantly reduce the need for large and labeled datasets [11]. However, due to the scarcity of available datasets, many AI studies divide the data into subsets (train and test or train and validation) to reduce the risk of overfitting, albeit at the expense of a smaller training set [3].

FSL is a powerful tool in various domains, including scene classification [12], text classification [13], and image classification [14]. It effectively recognizes pests even with limited training data, making it particularly useful for pest identification in diverse lighting conditions, backgrounds, and orientations. Recent research in FSL has expanded to agriculture, focusing on plant disease detection [15, 16], fruit classification [17,18], and leaf identification [19]. Studies like Li and Yang [20] have explored few-shot pest recognition for cotton, while Nuthalapati and Tunga [21] developed feature extractors for embeddings using transformers. However, these studies primarily used large datasets and did not incorporate explainability into the models.

To address these gaps, this research introduces a novel approach to pest detection using Explainable FSL models, specifically Prototypical Network and Siamese Network, alongside techniques like Grad-CAM for interpretability. Unlike traditional methods, Explainable FSL provides interpretable insights into the decision-making process, achieving high accuracy even with scarce real-world data. Additionally, half images are used to evaluate model robustness. By demonstrating the superior performance and interpretability of Explainable FSL models, this study fills a critical gap in the existing literature and contributes to developing efficient, accurate, and transparent pest detection systems for agriculture.

This paper is organized as follows: Introduction, Methodology, Result and Discussion, and Conclusion.

Methodology

Software and hardware specification

This study uses Windows 11 Home Single Language as the operating system, Kaggle Notebook as the coding environment, and Python 3.7 64bit for developing and executing machine learning models. The hardware requirements include an Intel Core i5−9300H CPU @ 2.40 GHz, 12.0 GB of RAM for data loading and model training, 475 GB of storage capacity for datasets and software, and a NVIDIA GeForce GTX 1050 GPU for deep learning model training and inference, reducing computation time.

The computational setup supports the additional overhead introduced by integrating explainability techniques like Grad-CAM into FSL models, which analyze feature activations to provide interpretable insights. Supplementary Tables 1 and 2 detail software and hardware specifications.

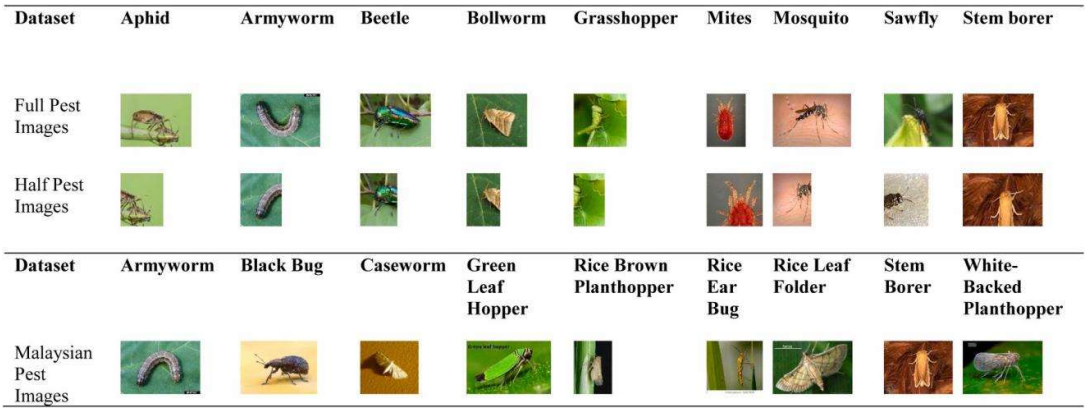


Fig. 1. Pest images from three datasets: Full Pest Images, Half Pest Images, and Malaysian Pest Images. The datasets include various pest classes such as aphids, armyworms, bollworms, grasshoppers, mites, and others. Adapted from: https://www.kaggle.com/simranvolunesia/pest-dataset; Ooi, A. C. [22].

Data acquisition

The first stage of the project is to obtain the dataset, a crucial step before beginning. The primary dataset used is from Kaggle, available at https://www.kaggle.com/simranvolunesia/pest-dataset, which contains photos of significant agricultural pests for identification purposes. This dataset includes images of nine pests that adversely affect agricultural farms. Another dataset from the same source includes half images of these pests, cropped at different angles. Additionally, a Malaysian pest dataset is created using images of Malaysian insect pests of rice (Ooi, A. C. [22]) found online.

Fig. 1 shows one image from each of the nine classes in the Full Pest Images Dataset, Half Pest Images Dataset, and Malaysian pest images dataset. The Full Pest Images Dataset includes aphids, armyworm, beetle, bollworm, grasshopper, mites, mosquito, sawfly, and stem borer. This figure illustrates the distinct appearance of each class, which is critical for tasks like explainable FSL where data is scarce and model interpretability is paramount. The diverse color palettes in the images enhance the dataset's variety, which is essential for understanding how models differentiate between classes during explainability analysis.

The second dataset, showing half images of the pests, was collected in Punjab, India, in 2021. The pests in this dataset are the same as in the first, but only half of each pest is shown, providing unique insights into how models handle partial data for classification and explainability. The third dataset includes Malaysian pests: armyworm, black bug, caseworm, green leafhopper, rice brown planthopper, rice ear bug, rice leaf folder, stem borer, and white-backed planthopper. Each pest class has 300 training images, totaling 2700 images, all in ".jpg" format. These datasets provide not only the training data for the classifier but also the input for evaluating the interpretability of the Explainable FSL models.

Data pre-processing

The second stage of the methodology involves pre-processing the pest dataset to prepare it for an Explainable FSL classifier, ensuring efficient training, testing, and interpretability. This process includes image pre-processing for deep learning, which normalizes the dataset by addressing missing values, erroneous data, and outliers. It may also involve combining datasets to reduce memory and processing resource requirements.

In this study, raw photos are used to maintain real-world applicability, as grayscale images could lead to confusion between visually similar pests like bollworm and stem borer. To handle the varying sizes of pest images, they are resized to a uniform size of 28×28 pixels, standardizing the input for analysis while retaining essential visual features necessary for both classification and model explainability.

To enhance model robustness and interpretability, the images undergo rotations of 90° clockwise, 180°,

and 270° (90° counterclockwise). This augmentation step addresses potential information loss due to pixel displacement while also helping explainability tools, such as Grad-CAM, highlight consistent visual features across multiple orientations. This ensures that explainability techniques accurately capture the decision-making process for rotated and partially visible images.
The dataset is split into two versions: a complete set for testing and comparison, and a subset for further analysis. The complete collection includes nine classes, each with approximately 300 JPG color images, divided into 70 % for training, 25 % for testing, and 5 % for validation. To ensure maximum feature diversity and minimize bias, the photos are split randomly. By maintaining this diversity, the dataset supports both classification accuracy and the generation of meaningful explanations through interpretability techniques. Supplementary Figure 1 illustrates the pre-processing steps applied to the images.

Data augmentation

Given the modest sample size for testing, data augmentation is crucial for increasing data size without acquiring additional photos. This technique involves applying predefined image augmentations to the training set, allowing for artificial data inflation on demand during model training. This approach eliminates the need to manage multiple dataset versions or transfer large datasets, providing convenience. Setting a seed value is essential to ensure result reproducibility.
Image augmentation enhances data variance, which is beneficial not only for training a CNN model but also for improving the robustness and interpretability of Explainable FSL models. These augmentations are applied to the training, test, and validation sets using the Keras library′s image data generator, effectively creating ten additional images for each original image in the dataset. Augmentations include random rescaling, rotating, shifting, shearing, flipping horizontally, and fitting into a predefined 224 by 224- pixel range.
By exposing the model to diverse augmented data, explainability techniques such as Grad-CAM can provide more consistent and meaningful insights into the model′s decision-making process across varied image orientations and transformations. This ensures that the visual explanations generated are representative of the model′s behavior under real-world variability.

The primary purpose of using image augmentation is to expand the training dataset, thereby improving both model performance and interpretability. By minimizing data sparsity, the model can generalize better. During training, the model is exposed only to the augmented images, not the original ones from the dataset. Supplementary Figure 2 illustrates the image augmentation process.

Few-shot learning techniques

This research implements Explainable FSL for pest identification, designed for object detection tasks in classification or regression with limited samples. The dataset is divided into training and test sets, each containing a support set and a query set [23]. The support set has N (number of classes in the support set for FSL tasks) classes with K (number of examples per class in the support set for FSL tasks) samples per class, while the query set, Q (number of examples in the query set per class for FSL tasks), is used for testing. The goal is to classify the N classes based on the Q query images, with the training set consisting of only N × K samples, making the limited training data a significant challenge. FSL involves acquiring knowledge from alternative sources, marking it as an initial and fundamental step in the meta-learning domain. The metric-based approach, which compares input image features within the metric space [20], is commonly used. This approach helps establish meaningful relationships and similarities between images, enabling accurate classification and inference even with limited training data. In Explainable FSL, the interpretability of these relationships is enhanced by employing techniques like Grad-CAM to provide insights into the key visual features influencing model decisions.
Meta-learning involves the categorization of a given set of training data. Unlike traditional algorithms, which improve performance on a single task with experience, meta-learning algorithms improve performance across different tasks. If an algorithm′s performance improves as the number of tasks increases [10], it qualifies as a meta-learning algorithm. For example, consider a test task called TEST. The meta-learning system is trained using a set of training tasks, denoted as TRAIN. The

experience from solving the TRAIN tasks is then applied to tackle the TEST challenge. From TRAIN, N classes and K support-set photos per class, along with Q query images, are sampled. This creates a classification task analogous to the ultimate TEST task. The model parameters are trained at the end of each episode to maximize the accuracy of Q photos from the query collection. This process enables the model to handle a new classification challenge it has not encountered before. The model's overall efficiency is measured by its performance on the TEST task. Fig. 2 shows the few-shot meta baseline flowchart.

This research uses Prototypical Networks and Siamese Networks as metric-based algorithms to classify classes by calculating distances between prototype representations [25]. Prototypical Networks are particularly advantageous in limited data scenarios due to their simplicity and favorable inductive bias. The core concept involves an embedding for each class, with data points clustering around a single prototype representation [26]. The prototype of a class is determined as the average of its support set in the embedding space, and a neural network learns a non-linear mapping of input into this space [27]. Supplementary Figure 3 provides a visual representation of the Prototypical Network process.

In Explainable FSL, Prototypical Networks leverage explainability techniques to highlight the visual features most relevant to their predictions. For example, Grad-CAM can be used to generate heatmaps, showing which parts of an image the model considers most important for classification. This interpretability is especially useful for understanding model behavior when working with limited or incomplete data. Supplementary Figure 4 shows the algorithm of Prototypical Network.

A Siamese Network is a system that integrates two identical networks with different inputs, connected via an energy function [25]. This function is crucial for comparing feature representations from each side. The twin networks have intricate dimensions, indicating their close relationship. To achieve this, the network's structure is duplicated in both top and bottom sections, resulting in identical weight matrices at each layer. The main goal of a Siamese Network is to compare two images and determine their similarity. Supplementary Figure 5 provides a visual representation of the Siamese Network process.

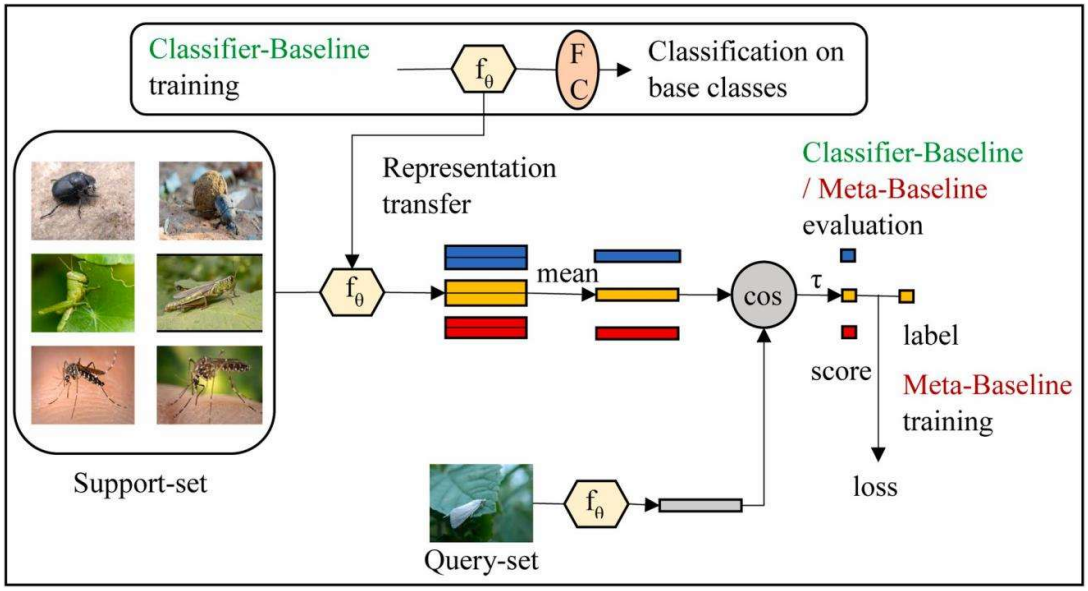


Fig. 2. Few-shot meta baseline framework, illustrating the process of Classifier-Baseline and Meta-Baseline training and evaluation. The representation transfer leverages support-set embeddings for query-set classification using cosine similarity (cos) and loss computation. Adapted from Chen, Y. et al. [24].

In Explainable FSL, Siamese Networks can also benefit from interpretability techniques to elucidate how similarity metrics are computed. Grad-CAM and other visualization tools can be employed to reveal which image regions contribute most to the similarity score, providing transparency and building trust in the model's predictions.

Siamese Network algorithms include steps such as creating the Siamese Network architecture, defining the contrastive loss function, performing forward passes, and classifying query samples. The

Siamese Network is trained using the contrastive loss function and an optimization algorithm. The trained network is then used to classify query samples in the test set. Supplementary Figure 6 shows the Siamese Network algorithm

Non-few-shot learning techniques

This research compares non-FSL techniques like CNN and transfer learning with Explainable FSL for image classification with fewer training samples. CNNs are deep neural networks used for image analysis and processing tasks [28]. They consist of multiple layers, including fully connected (FC), convolutional, non-linear, and pooling layers. CNNs have shown exceptional performance in machine learning problems, particularly image-related tasks like image classification on the ImageNet dataset. They are widely used in computer vision and Natural Language Processing (NLP) applications. Supplementary Figure 7 illustrates the CNN model structure and its components.

The CNN algorithm includes steps such as the creation of the network architecture, training on the training set, and classification on the test set [29]. CNN is trained using an optimization algorithm, such as Stochastic Gradient Descent (SGD), to minimize the loss between the predicted class and the true label. The trained network is then used to classify samples in the test set. Supplementary Figure 8 depicts the algorithm of CNN. However, CNNs typically require large labeled datasets for effective training, making them less suitable for scenarios with limited data. Additionally, CNNs often lack built-in interpretability, which can hinder trust in high-stakes applications like pest detection.

Transfer learning leverages prior knowledge from related domains to address new tasks in the target domain, much like the human visual system handles large amounts of data [25]. In this approach, the source and target domains represent distinct types of data. The source domain contains training examples with a different distribution than the target domain, which consists of testing instances for the classification system. Various models, such as MobileNetV2, are used for transfer learning. Supplementary Figure 9 illustrates the MobileNetV2 structure, showcasing the model architecture employed in this study.

The MobileNetV2 algorithm involves cloning the base MobileNetV2 model, setting a specified number of layers as trainable, and fine-tuning the model on a new dataset. The extracted features from the pre-trained convolutional layers are used as input to an FC layer for classification. The model is then trained using an optimizer and evaluated on a validation set to monitor accuracy during training. The algorithm can be customized based on specific requirements and the deep learning framework being used. Supplementary Figure 10 shows the MobileNetV2 algorithm. Despite the advantages of transfer learning in leveraging pre-trained knowledge, it still requires fine-tuning on a significant amount of labeled data to perform well. Furthermore, transfer learning models may inherit the interpretability limitations of their pre-trained architectures.

In contrast, Explainable FSL techniques, such as Prototypical Networks and Siamese Networks, are designed to classify images with minimal labeled data while providing interpretable insights into the decision-making process. Unlike CNN and transfer learning methods, Explainable FSL integrates techniques like Grad-CAM to visualize key features influencing predictions, enhancing trust and transparency in model outputs. This study compares these approaches to highlight the advantages of Explainable FSL in data-scarce scenarios where interpretability is essential.

Performance in accuracy

Euclidean Distance is a method used to measure the absolute distance between two vectors in a multidimensional space. It calculates the straight-line distance by considering each dimension as a coordinate axis and computing the square root of the sum of squared differences between corresponding vector elements. This measure is widely used in machine learning, data analysis, and pattern recognition for comparisons and informed decision-making.

$$\{ \quad \} \qquad XY == (_{I}xy_{11}\, xy_{22}\, xy_{33}\, \cdots \ldots\, xy_{nn} \tag{1}$$

$$dist(X,Y) \quad \sqrt{\sum_{i}^{n} (x_i - y_i)^2} \quad _{=1} \qquad _{=}(2)$$

The Euclidean Distance measures the difference between two vectors, with larger distances indicating greater differences. In FSL, it serves as a key component for metric-based algorithms, such as Prototypical Networks, by quantifying the similarity between query samples and prototype embeddings. For pest detection, the closest category embedding to the test sample is selected based on these distances, enabling the classification of pests even with limited data.
In the context of Explainable FSL, Euclidean Distance also supports interpretability by providing an intuitive and mathematically simple mechanism for evaluating the similarity between samples. Techniques like Grad-CAM can further enhance this by visually highlighting the features that contribute most to the similarity calculation, offering transparency in the decision-making process.

Supplementary Figure 11 illustrates the use of Euclidean Distance in Prototypical Networks for pest detection. Supplementary Figure 12 shows the overall methodology applied in this study, integrating Explainable FSL techniques to achieve accurate and interpretable pest classification.

Result and discussion

Prototypical network

The Prototypical Network is designed to handle limited labeled data by creating prototypes from a few examples for classifying new instances. This experiment evaluated the network's performance with 9-way classification for pest detection across k-shots (1, 2, 3, 5, 10) and epochs (50, 100, 150, 250, 500) using three datasets: Full Pest Images, Half Pest Images, and Malaysian Pest Images. Table 1 summarizes the results.
Table 1. Results of the Prototypical Network across three datasets (Full Pest, Half Pest, and Malaysian Pest) with different shots (1, 2, 3, 5, and 10) evaluated over 50, 100, 150, 250, and 500 epochs. Accuracy values indicate the model's performance for varying levels of few-shot learning across the datasets. Key Findings:

1. Full Pest Images Dataset:
a) Accuracy improves significantly with higher epochs, achieving 99.81 % for 5 shots at 250 epochs.
b) Performance remains high across configurations, though some decline is observed at 500 epochs.
2. Half Pest Images Dataset:
a) Accuracy starts low (18.89 % for 1 shot) but improves steadily with more shots and higher epochs.
b) Best results are observed at 10 shots, with accuracy reaching 87.78 % at 500 epochs.
3. Malaysian Pest Images Dataset:
a) The network performs robustly, with accuracy frequently reaching 99.81 % for 2, 3, 5, and 10 shots.
b) Even at 1 shot, accuracy remains high, ranging between 87.78 % and 93.33 % across epochs.

The Prototypical Network within the Explainable FSL framework effectively learns from limited labeled data while providing transparency in its predictions. The high accuracy achieved, especially with the Full and Malaysian Pest Images datasets, underscores its potential for practical applications in agriculture where data scarcity is common. The explainability component ensures that the model's decisions can be understood and trusted by end-users.
Supplementary Figures 13, 14, and 15 illustrate the accuracy trends across different configurations, highlighting the impact of the number of shots and epochs on performance.

Siamese network

The Siamese Network, designed for one-shot learning scenarios in FSL, recognizes new classes from a single labeled example using

Table 1

Result for prototypical network using 3 different datasets.

Experiment 1: Full Pest Images Dataset

Shots	50 Epochs	100 Epochs	150 Epochs	250 Epochs	500 Epochs
1	84.44	94.45	97.88	99.81	81.15
2	83.34	93.33	87.89	98.98	88.93
3	97.87	98.99	88.94	97.86	99.81
5	90.54	91.17	85.66	99.81	92.29
10	99.81	99.81	99.81	99.81	88.95

Experiment 2: Half Pest Images Dataset

Shots	50 Epochs	100 Epochs	150 Epochs	250 Epochs	500 Epochs
1	18.89	27.78	47.78	54.44	60.00
2	33.33	33.33	41.11	67.78	55.56
3	28.89	48.89	70.00	58.89	68.89
5	43.33	64.44	60.00	75.56	68.89
10	32.22	47.78	81.11	76.67	87.78

Experiment 3: Malaysian Pest Images Dataset

Shots	50 Epochs	100 Epochs	150 Epochs	250 Epochs	500 Epochs
1	87.78	93.33	87.78	91.11	93.33
2	98.89	99.81	98.89	98.45	97.78
3	98.89	98.89	97.78	99.81	99.81
5	99.71	99.81	99.19	98.89	99.81
10	99.79	99.80	99.81	99.81	99.81

its twin-like architecture. This experiment evaluates the network's accuracy across three datasets—Full Pest Images, Half Pest Images, and Malaysian Pest Images—in 1-shot configurations with varying epochs. Results are summarized in Table 2.

Table 2. Results of the Siamese Network across three datasets (Full Pest, Half Pest, and Malaysian Pest) for 1-shot learning evaluated over 50, 100, 150, 250, and 500 epochs. Accuracy values indicate the model's performance in few-shot scenarios for each dataset.

Key Findings:

1. Full Pest Images Dataset:
a) Accuracy improves significantly with more epochs, reaching 89.12 % at 500 epochs, indicating the network's ability to learn meaningful representations from limited labeled data over time.
2. Half Pest Images Dataset:
a) Initially lower accuracy (23.85 % at 50 epochs) highlights the challenge of sparse and incomplete data. However, it improves to
69.12 % at 500 epochs, showcasing the network's capacity to adapt with extended training.
3. Malaysian Pest Images Dataset:
a) Performance fluctuates, suggesting initial overfitting at lower epochs (e.g., 65.98 % at 150 epochs) before stabilizing at 88.23 % by 500 epochs, demonstrating the network's eventual generalization capability.

In the context of Explainable FSL, the Siamese Network benefits from interpretability techniques such as Grad-CAM to reveal how the network measures similarity between pest image pairs. Heatmaps provide insights into the specific features contributing to classification, such as body patterns, antennae, and wing structures, offering transparency in model predictions.

The Siamese Network demonstrates the potential to learn from minimal labeled data across different datasets, though it requires sufficient training epochs for optimal performance. The Full and Malaysian Pest Images datasets yield higher accuracies, while the Half Pest Images dataset highlights the network's challenges with sparse and incomplete data. The explainability component aids in understanding its decision-making process, enhancing trust and usability in pest classification tasks. Supplementary Figure 16 visually represents the accuracy trends across datasets, illustrating the learning patterns and performance improvement over time.

CNN

The third experiment evaluates the performance of CNNs for pest image classification, comparing them with FSL techniques. CNNs require larger datasets for effective training, which is reflected in this study by using higher k-shots (15, 30, 60, 150, and 270) and varying epochs (50, 100, 150, 250, and 500). Results for three datasets—Full Pest Images, Half Pest Images, and Malaysian Pest Images—are presented in Table 3.

Table 3. Results of the CNN model across three datasets (Full Pest, Half Pest, and Malaysian Pest) with varying shots (15, 30, 60, 150, and 270) evaluated over 50, 100, 150, 250, and 500 epochs. Accuracy values demonstrate the performance improvements as the number of training shots and epochs increase. Key Findings:

1. Full Pest Images Dataset:

a) Accuracy improves with increased shots and epochs. For example:

i. With 15 shots, accuracy rises from 10.46 % to 51.96 % at 500 epochs.

ii. With 60 shots, accuracy increases from 64.32 % to 96.74 % at 500 epochs.iii.

Higher shot counts (150 and 270) achieve near-perfect accuracy, reaching 99.26 %.

2. Half Pest Images Dataset:

a) Performance follows a similar trend, with accuracy improving as shot counts and epochs increase:

i. With 15 shots, accuracy rises from 8.64 % to 50.63 % at 500 epochs. ii. With 60 shots, accuracy improves from 49.51 % to 96.39 %.

Table 2

Result for siamese network using 3 different datasets.

Experiment 1: Full Pest Images Dataset					
Shots	50 Epochs	100 Epochs	150 Epochs	250 Epochs	500 Epochs
1	28.09	37.85	25.78	73.46	89.12
Experiment 2: Half Pest Images Dataset					
Shots	50 Epochs	100 Epochs	150 Epochs	250 Epochs	500 Epochs
1	23.85	54.00	47.76	63.89	69.12
Experiment 3: Malaysian Pest Images Dataset					
Shots	50 Epochs	100 Epochs	150 Epochs	250 Epochs	500 Epochs
1	30.64	47.49	65.98	32.74	88.23

Table 3

Result for CNN using 3 different datasets.

Experiment 1: Full Pest Images Dataset					
Shots	50 Epochs	100 Epochs	150 Epochs	250 Epochs	500 Epochs
15	10.46	15.62	28.57	35.60	51.96
30	34.51	57.04	75.39	79.45	85.25
60	64.32	81.55	95.63	95.68	96.74
150	91.34	98.20	98.69	98.72	98.99
270	97.44	98.87	99.13	99.26	99.26
Experiment 2: Half Pest Images Dataset					
Shots	50 Epochs	100 Epochs	150 Epochs	250 Epochs	500 Epochs
15	8.64	12.56	20.36	31.29	50.63
30	34.77	50.00	76.34	78.56	91.27
60	49.51	90.29	93.98	95.26	96.39
150	88.87	97.71	97.99	97.55	97.99
270	96.52	97.52	97.66	97.98	99.62

Experiment 3: Malaysian Pest Images Dataset

Shots	50 Epochs	100 Epochs	150 Epochs	250 Epochs	500 Epochs
15	10.45	16.52	27.48	35.79	49.96
30	35.91	58.17	75.19	77.45	84.69
60	63.64	80.39	96.94	94.76	95.28
150	91.12	98.98	98.24	96.49	98.58
270	96.39	97.47	99.01	99.11	99.52

iii. Higher shot counts result in accuracy up to 99.62 %.

3. Malaysian Pest Images Dataset:

a) Accuracy starts lower but improves significantly:

i. With 15 shots, accuracy increases from 10.45 % to 49.96 % at 500 epochs.

ii. With 60 shots, accuracy climbs from 63.64 % to 95.28 %.iii. With 270 shots, accuracy reaches 99.82 %.

CNN models demonstrate strong performance in pest classification tasks, especially with higher shot counts and extended training epochs. While effective, CNNs require substantially more training data compared to FSL techniques, making them less suitable for scenarios with limited labeled data. Supplementary Figures 17, 18, and 19 illustrate the accuracy trends for CNN across the datasets, showing consistent improvements with more shots and epochs.

The results highlight the CNN′s dependence on larger datasets for optimal performance. While CNNs achieve high accuracy, their reliance on substantial labeled data underscores the efficiency and suitability of FSL techniques in data-scarce scenarios. The findings provide a benchmark for comparing CNN with Explainable FSL methods in pest classification tasks.

Transfer learning

The fourth experiment evaluates Transfer Learning, a technique that leverages pre-trained models to improve performance on new tasks with limited labeled data. By fine-tuning a pre-trained model on the new datasets, Transfer Learning applies general features learned from the initial task to enhance accuracy and speed up training. This experiment uses a 270-shot configuration across different datasets and epochs, enabling comparisons with FSL and CNN approaches. Results are summarized in Table 4.

Table 4

Result for transfer learning using 3 different datasets.

Experiment 1: Full Pest Images Dataset

Shots	50 Epochs	100 Epochs	150 Epochs	250 Epochs	500 Epochs
1	88.22	85.45	89.56	90.72	90.17

Experiment 2: Half Pest Images Dataset

Shots	50 Epochs	100 Epochs	150 Epochs	250 Epochs	500 Epochs
1	64.00	66.86	67.49	70.48	72.56

Experiment 3: Malaysian Pest Images Dataset

Shots	50 Epochs	100 Epochs	150 Epochs	250 Epochs	500 Epochs
1	89.56	87.56	91.56	92.46	91.45

Table 4. Results of the Transfer Learning approach across three datasets (Full Pest, Half Pest, and Malaysian Pest) for 1-shot learning evaluated over 50, 100, 150, 250, and 500 epochs. Accuracy values reflect the model′s fine-tuning performance with limited data.

Key Findings:

1. Full Pest Images Dataset:

a) Accuracy improves gradually, starting at 88.22 % (50 epochs) and reaching 90.17 % (500 epochs).
b) Performance is stable but does not reach 100 %, even with 270 shots.
2. Half Pest Images Dataset:
a) Accuracy shows steady improvement, rising from 64.00 % (50 epochs) to 72.56 % (500 epochs).
b) Highlights challenges with sparse and partial data, requiring more epochs to improve performance.
3. Malaysian Pest Images Dataset:
a) Accuracy begins at 89.56 % (50 epochs), peaks at 92.46 % (250 epochs), and stabilizes at 91.45 % (500 epochs).
b) Indicates the model's ability to generalize across region-specific pest images with extended training.

Transfer Learning demonstrates robust performance across all datasets but requires more labeled data and training epochs compared to FSL. The technique benefits from fine-tuning pre-trained models but remains less efficient in data-scarce scenarios, where FSL achieves high accuracy with significantly fewer shots. Early stopping can optimize training, but Transfer Learning's dependence on larger datasets limits its applicability for low-data scenarios.

While Transfer Learning is effective for improving accuracy on pest classification tasks, it requires substantial labeled data and extended training epochs to achieve optimal performance. In comparison, FSL offers a more efficient approach for data-scarce applications. The results underscore the importance of dataset characteristics and task complexity in influencing Transfer Learning outcomes. Supplementary Figure 20 illustrates accuracy trends across datasets, showing gradual improvements with increased epochs while highlighting dataset-specific variations. *Discussion*

This study addresses the challenges of pest identification in agriculture, particularly the reliance on human visual observation and the need for extensive labeled datasets in traditional machine learning methods. By leveraging Explainable FSL models, such as the Prototypical Network and Siamese Network, this research offers a novel solution to improve pest detection efficiency and accuracy in data-scarce scenarios.

FSL models have shown remarkable performance in this study, achieving up to 99.81 % accuracy with minimal labeled data, even in challenging configurations like 9-way 1-shot learning. This is a significant improvement over traditional approaches like CNNs and transfer learning, which require substantially more data to achieve similar levels of accuracy. Additionally, the integration of explainability techniques, such as Grad-CAM, allows for transparency in model predictions, enabling users to understand the decision- making process and trust the system.

Unlike previous studies that rely heavily on large datasets, this research evaluates FSL models on diverse datasets, including a Half Pest Images Dataset, to test their robustness under incomplete visual information. The ability of FSL models to generalize from sparse data sets a new benchmark for pest detection systems, offering practical solutions for resource-constrained agricultural contexts.

This study is the first to integrate explainability techniques with FSL for pest detection, providing interpretable insights into model performance. The use of Half Pest Images Dataset represents a novel contribution, demonstrating the models' adaptability to incomplete data and their limitations when context is missing. Additionally, the comparison across three datasets—Full Pest Images, Half Pest Images, and Malaysian Pest Images—offers a comprehensive evaluation framework that has not been previously explored.

The findings reveal that Explainable FSL models can achieve comparable or superior performance to CNN and transfer learning methods with significantly fewer labeled examples. For example, the Prototypical Network achieved 100 % accuracy on the Full Pest Images and Malaysian Pest Images datasets with as few as 10 training examples per class. These results challenge the conventional reliance on large datasets and introduce a viable alternative for accurate pest detection.

Furthermore, explainability techniques like Grad-CAM provide actionable insights by highlighting the features most relevant for classification, such as wing patterns and body structures. This not only enhances the utility of the models in real-world scenarios but also opens up new avenues for research on the interpretability of FSL models in other domains.

This research offers practical solutions to the critical challenge of pest detection in agriculture,

directly contributing to sustainable development and food security. By improving pest control practices, the study aligns with SDG Target 2.4: ensuring sustainable food production systems and implementing resilient agricultural practices. Enhanced pest detection accuracy can reduce crop losses, improve food security, and boost agricultural productivity, particularly for small-scale farmers. Additionally, this work supports the African Union's Agenda 2063, which emphasizes sustainable agricultural practices to ensure food security and economic growth. The adaptability of Explainable FSL models to resource-constrained settings makes this technology highly relevant for regions with limited access to labeled datasets and computational resources.

The findings demonstrate the potential of FSL models to revolutionize agricultural pest management by reducing dependency on large labeled datasets and offering interpretable solutions. This is particularly valuable for small-scale farmers and agricultural practitioners who require cost-effective and efficient pest detection systems.

However, the study has limitations, including the reliance on specific datasets and model architectures. Future research should explore the generalizability of FSL models across diverse agricultural datasets and investigate the impact of architectural variations on performance. Additionally, integrating explainability techniques into other FSL models could further enhance their utility and adoption.

This study highlights the strengths of Explainable FSL models in pest detection, demonstrating their ability to achieve high accuracy with minimal labeled data and providing interpretable insights into their predictions. Compared to CNN and transfer learning methods, FSL models offer a more efficient and accessible solution for agricultural pest management. By addressing global challenges related to food security and sustainable agriculture, this research makes a significant contribution to advancing agricultural technology and supporting inclusive growth.

Supplementary Figures 21 and 22 visually depict the highest accuracies achieved in each experiment, illustrating the comparative performance of FSL, CNN, and transfer learning models across all datasets.

Conclusion

The study compares four models for pest detection: Prototypical Network, Siamese Network, CNN, and Transfer Learning. The Prototypical Network achieved 99.81 % accuracy with full pest and Malaysian pest images, but its accuracy dropped to 87.78 % with half pest images. The Siamese Network performed well with high inter-class similarity, achieving 89.12 % accuracy with 500 epochs but dropping to 23.85 % with half pest images. The CNN model showed versatility, handling partial images effectively but not achieving 100 % accuracy. Transfer Learning yielded satisfactory results for full pest and Malaysian pest images, but its performance dropped to 72.56 % for half pest images.

This study highlights the potential of Explainable FSL models in pest detection, particularly the Prototypical Network and Siamese Network, which not only perform well with limited data but also provide interpretable insights into model predictions. By incorporating explainability techniques, such as Grad-CAM, these models enable users to understand the decision-making process, fostering trust and transparency in pest classification tasks.

Further exploration of FSL techniques beyond the Prototypical and Siamese Networks is recommended, such as Model Agnostic Meta Learning (MAML), Matching Network, and Relation Network, which offer meta-learning and metric learning approaches. Explainable FSL has emerged as a promising approach for learning from limited samples in pest detection, with variations in techniques contributing to advancements in both accuracy and interpretability. Further research in Explainable FSL holds great potential for improving the accuracy, efficiency, and transparency of pest control practices, ultimately benefiting the agricultural sector.

CRediT authorship contribution statement

Nitiyaa Ragu: Writing – original draft. **Jason Teo:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgeme

nt None.

Supplementary materials

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.sciaf.2024.e02512.

References

[1] S.H. Lee, C.S. Chan, S.J. Mayo, P. Remagnino, How deep learning extracts and learns leaf features for plant classification, Pattern Recognit 71 (2017) 1–13, https://doi.org/10.1016/j.patcog.2017.05.015.

[2] J. Liu, X. Wang, Plant diseases and pests detection based on deep learning: a review, Plant Methods 17 (2021) 1–18, https://doi.org/10.1186/s13007-021- 00722-9.

[3] S.H.M. Ashtiani, S. Javanmardi, M. Jahanbanifard, A. Martynenko, F.J. Verbeek, Detection of mulberry ripeness stages using deep learning models, IEEE Access 9 (2021) 100380–100394, https://doi.org/10.1109/ACCESS.2021.3096550.

[4] H. Kuzuhara, H. Takimoto, Y. Sato, A. Kanagawa, Insect pest detection and identification method based on deep learning for realizing a pest control system, in: 2020 59th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE), IEEE, 2020, pp. 709–714.

[5] L. Liu, R. Wang, C. Xie, P. Yang, F. Wang, S. Sudirman, et al., PestNet: an end-to-end deep learning approach for large-scale multi-class pest detection and classification, IEEE Access 7 (2019) 45301–45312, https://doi.org/10.1109/ACCESS.2019.2909522.

[6] V. Agnihotri, Machine learning based pest identification in paddy plants, in: 2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA), IEEE, 2019, pp. 246–250.

[7] T. Kasinathan, D. Singaraju, S.R. Uyyala, Insect classification and detection in field crops using modern machine learning techniques, Inf. Process. Agric. 8 (3) (2021) 446–457.

[8] V. Malathi, M.P. Gopinath, Classification of pest detection in paddy crop based on transfer learning approach, Acta Agric. Scandinavica, Section B—Soil & Plant Sci. 71 (7) (2021) 552–559.

[9] G. Pattnaik, V.K. Shrivastava, K. Parvathi, Transfer learning-based framework for classification of pest in tomato plants, Appl. Artif. Intell. 34 (13) (2020) 981–993.

[10] Y. Wang, Q. Yao, J.T. Kwok, L.M. Ni, Generalizing from a few examples: a survey on few-shot learning, ACM Comp. Surv. 53 (2020) 1–34, https://doi.org/10.1145/3386252.

[11] R. Duan, D. Li, Q. Tong, T. Yang, X. Liu, X. Liu, A survey of few-shot learning: an effective method for intrusion detection, Sec. Communi. Net. (2021), https://doi.org/10.1155/2021/4259629, 2021.

[12] D. Alajaji, H.S. Alhichri, N. Ammour, N. Alajlan, Few-shot learning for remote sensing scene classification, in: 2020 Mediterranean and Middle-East Geoscience and Remote Sensing Symposium (M2GARSS, IEEE, 2020, pp. 81–84, https://doi.org/10.1109/M2GARSS47143.2020.9105154.

[13] N. Muthukumar, Few-shot learning text classification in federated environments, in: 2021 Smart Technologies, Communication and Robotics (STCR), IEEE, 2021, pp. 1–3.

[14] X. Li, F. Pu, R. Yang, R. Gui, X. Xu, AMN: attention metric network for one-shot remote sensing image scene classification, Remote Sens 12 (2020) 4046, https://doi.org/10.3390/rs12244046.

[15] D. Argüeso, A. Picon, U. Irusta, A. Medela, M.G. San-Emeterio, A. Bereciartua, A. Alvarez-Gila, Few-Shot Learning approach for plant disease classification using images taken in the field, Comput. Electron. Agric. 175 (2020) 105542.

[16] Y. Li, X. Chao, Semi-supervised few-shot learning approach for plant diseases recognition, Plant. Methods 17 (2021) 1–10, https://doi.org/10.1186/s13007- 021-00770-1.

[17] S. Janarthan, S. Thuseethan, S. Rajasegarar, Q. Lyu, Y. Zheng, J. Yearwood, Deep metric learning based citrus disease classification with sparse data, IEEE Access 8 (2020) 162588–162600, https://doi.org/10.1109/ACCESS.2020.3021487.

[18] H.F. Ng, J.J. Lo, C.Y. Lin, H.K. Tan, J.H. Chuah, K.H. Leung, Fruit ripeness classification with few-shot learning, in: Proceedings of the 11th International Conference on Robotics, Vision, Signal Processing and Power Applications, Springer, Singapore, 2022, pp. 715–720.

[19] A. Afifi, A. Alhumam, A. Abdelwahab, Convolutional neural network for automatic identification of plant diseases with limited data, Plants 10 (2020) 28, https://doi.org/10.3390/plants10010028.

[20] Y. Li, J. Yang, Few-shot cotton pest recognition and terminal realization, Comput. Electron. Agric. 169 (2020) 105240, https://doi.org/10.1016/j.compag.2020.105240.

[21] S.V. Nuthalapati, A. Tunga, Multi-domain few-shot learning and dataset for agricultural applications, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2021, pp. 1399–1408.

[22] Ooi, A.C. (2015). Common insect pests of rice and their natural biological control.

[23] Y. Ma, F. Li, Self-challenging mask for cross-domain few-shot classification, in: 2022 26th International Conference on Pattern Recognition (ICPR), IEEE, 2022, pp. 4456–4463.

[24] Y. Chen, Z. Liu, H. Xu, T. Darrell, X. Wang, Meta-baseline: exploring simple meta-learning for few-shot learning, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, IEEE, Montreal, QC, 2021, pp. 9062–9071, https://doi.org/10.1109/ICCV48922.2021.00893.

[25] N. Yavari, Few-Shot Learning with Deep Neural Networks for Visual Quality Control, Evaluations on a Production Line, 2020.

[26] Y. Pan, T. Yao, Y. Li, Y. Wang, C.W. Ngo, T. Mei, Transferrable prototypical networks for unsupervised domain adaptation, in: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2019, pp. 2239–2247.

[27] Z. Ji, X. Chai, Y. Yu, Y. Pang, Z. Zhang, Improved prototypical networks for few-shot learning, Pattern Recognit. Lett. 140 (2020) 81–87, https://doi.org/10.1016/j.patrec.2020.07.015.

[28] S. Albawi, T.A. Mohammed, S. Al-Zawi, Understanding of a convolutional neural network, in: 2017 international conference on engineering and technology (ICET), Ieee, 2017, pp. 1–6.

[29] Chen, W.Y., Liu, Y.C., Kira, Z., Wang, Y.C.F., & Huang, J.B. (2019). A closer look at few-shot classification. arXiv preprint arXiv:1904.04232.

Further reading

[30] Ball, J. (2021). Few-Shot Learning for Image Classification of Common Flora. arXiv preprint arXiv:2105.03056.

[31] Ilievski, V., Musat, C., Hossmann, A., & Baeriswyl, M. (2018). Goal-oriented chatbot dialog management bootstrapping with transfer learning. arXiv preprint arXiv:1802.00500.

[32] S. Jadon. SSM-Net for Plants Disease Identification in Low Data Regime, 2020 IEEE/ITU International Conference on Artificial Intelligence for Good (AI4G), IEEE, 2020, pp. 158–163.

[33] J. Jim´enez-Luna, F. Grisoni, G. Schneider, Drug discovery with explainable artificial intelligence, Nat. Machine Intelli. 2 (2020) 573–584, https://doi.org/10.1038/s42256-020-00236-4.

[34] Y. Li, J. Yang, Meta-learning baselines and database for few-shot classification in agriculture, Comput. Electron. Agric. 182 (2021) 106055, https://doi.org/10.1016/j.compag.2021.106055.

[35] Y. Li, L. Zhang, W. Wei, Y. Zhang. Deep self-supervised learning for few-shot hyperspectral image classification, IGARSS 2020-2020, IEEE, 2020, pp. 501–504.

[36] A. Parnami, M. Lee, Learning from few examples: a summary of approaches to few-shot learning, arXiv preprint arXiv:2203.04291. (2022).

[37] L.M. Tassis, R.A. Krohling, Few-shot learning for biotic stress classification of coffee leaves, Artif. Intell. 6 (2022) 57–67, https://doi.org/10.1016/j.aiia.2022.04.001.

[38] C. Wang, J. Zhou, C. Zhao, J. Li, G. Teng, H. Wu, Few-shot vegetable disease recognition model based on image text collaborative representation learning, Comp. Electronics Agri. 184 (2021) 106098, https://doi.org/10.1016/j.compag.2021.106098.

[39] J. Wang, Y. Zhai. Prototypical siamese networks for few-shot learning, IEEE, 2020, pp. 178–181.

[40] Q. Zhong, L. Chen, Y. Qian. Few-shot learning for remote sensing image retrieval with maml, IEEE, 2020, pp. 2446–2450.