

Forest fire and smoke detection using deep learning

Siddharth Sahni

Capstone Project Report

Golden Gate University – Data Scientist Bootcamp

Table of Contents

1. Background	4
1.1 Introduction.....	4
1.2 Problem Statement	4
1.3 Approach.....	4
1.3.1 Data Preprocessing	4
1.3.2 Model Selection and Training	4
1.3.3 Verification and Evaluation	5
1.3.4 Prediction and Testing	5
2. Setup Image Generator.....	5
2.1 Data Augmentation	5
2.2 Loading Data.....	5
2.3 Handling Class Imbalance.....	5
2.4 Visualization	6
2.5 Test Data	6
3. Model Building.....	6
3.1 Building Overview.....	6
3.1.1 Training History Plot.....	6
3.1.2 Custom Callback.....	6
3.1.3 Training the Model.....	6
3.1.4 Model Evaluation	7
3.2 Models	7
3.2.1 MobileNetV2.....	7
3.2.2 ResNet152.....	8
3.2.3 InceptionV3.....	9
3.2.4 InceptionResNetV2	10
3.2.5 DenseNet169	11
4. Evaluation	11
4.1 Evaluation Techniques.....	11
4.2 Evaluation Models.....	12
4.2.1 For MobileNetV2.....	12
4.2.2 For ResNet152.....	12
4.2.3 For InceptionV3.....	12
4.2.4 For InceptionResNetV2	13

4.2.5 For DenseNet169	13
5. Conclusion.....	14
6. Future Scope	14
6.1 Integration with IoT and Smart Systems.....	14
6.2 Improved Accuracy with Multimodal Data	14
6.3 Enhancing Detection in Challenging Conditions.....	14
6.4 Edge Computing for Faster Response.....	14
6.5 Scalability and Global Implementation	14
6.6 Automatic Incident Control Integration	15
6.7 Incorporating Drone Technology.....	15
7. System Requirements	15
7.1 Hardware Requirements	15
7.2 Software Requirements	16
7.3 Additional Libraries	16
7.4 Development Environment	16
7.5 Internet Connection	16
8. References	17

1. Background

1.1 Introduction

Forest fires have become an increasing environmental threat, causing significant ecological and economic damage globally. Early detection is crucial in preventing widespread devastation and reducing response times. Traditional fire detection methods, such as temperature and smoke sensors, often suffer from limitations like delayed response, high costs, and poor coverage in large forest areas. To overcome these limitations, computer vision techniques combined with deep learning have emerged as powerful tools for fire and smoke detection. This project explores the use of deep learning algorithms for detecting forest fires using surveillance cameras, which offer a cost-effective and efficient solution.

1.2 Problem Statement

Lately, there have been many fire outbreaks which is becoming a growing issue and the damage caused by these types of incidents is tremendous to nature and human interests. Such incidents have highlighted the need for more effective and efficient fire detection systems.

The traditional systems that rely on temperature or smoke sensors have limitations such as slow response time and inefficiency when the fire is at a distance from the detectors.

Moreover, these systems are also costly. As an alternative, researchers are exploring computer vision and image processing techniques as a cost-effective solution. One such method is the use of surveillance cameras to detect fires and alert the relevant parties.

Computer vision-based fire detection, which utilizes image processing techniques, has the potential to be a useful approach in situations where traditional methods are not feasible. The algorithm for fire detection uses visual characteristics of fires such as brightness, color, texture, flicker, and trembling edges to distinguish them from other stimuli.

This project is aimed at building a Fire Detection Model using Deep Learning. The key objective of the project is to identify fires from the images we can get from surveillance system or other resources.

1.3 Approach

The project utilized deep learning models such as **MobileNetV2**, **ResNet152**, **InceptionV3**, **InceptionResNetV2**, and **DenseNet169** to detect fire and smoke from forest images. The dataset used for training consisted of labeled images of smoke, fire, and non-fire scenarios, which were preprocessed and augmented to improve model robustness. The models were trained with performance metrics like categorical accuracy, recall, and precision to ensure high accuracy in detecting fire-related incidents.

The workflow of the approach was as follows:

1.3.1 Data Preprocessing

Collected images were rescaled and augmented to improve generalization. Techniques such as horizontal flipping, rotation, and zoom were applied to simulate diverse real-world conditions.

1.3.2 Model Selection and Training

Multiple pre-trained deep learning models were fine-tuned using the forest fire dataset. These models were trained using the Adam optimizer and a categorical cross-entropy loss function.

1.3.3 Verification and Evaluation

The models were evaluated based on their validation accuracy, loss, precision, and recall. The **best-performing model** (DenseNet169 in this case) was selected for further deployment.

1.3.4 Prediction and Testing

The selected model was tested on unseen data to predict fire and smoke occurrences. Performance was measured in terms of accuracy and the time taken for detection per image, demonstrating the model's practical application for real-time surveillance.

This approach shows promising results and serves as a scalable framework for integrating deep learning with forest monitoring systems for early fire detection.

2. Setup Image Generator

The image generation process for this project involves several key steps to ensure proper data augmentation and balanced training, validation, and testing datasets. The approach begins with defining constants like `batch_size` and `img_dim`, which are used to control the batch size and image dimensions respectively. A helper function `getImgTensor(img_d)` is implemented to return the target size for images, defining a 3-channel RGB image format of the specified dimension (299x299 pixels).

2.1 Data Augmentation

To handle overfitting and improve the robustness of the model, data augmentation is performed using the `ImageDataGenerator` class from Keras. The augmentation techniques applied include rescaling pixel values (`rescale=1./255`), random shearing (`shear_range=0.2`), zooming (`zoom_range=0.2`), and horizontal flipping (`horizontal_flip=True`). Additional transformations like random rotation (`rotation_range=45`), width and height shifting, and filling modes are used to introduce more variance into the training data. These augmentations are particularly useful for training deep learning models on limited datasets, as they simulate diverse real-world scenarios.

2.2 Loading Data

The data is split into training and validation sets using `train_datagen.flow_from_directory()`, with the validation set comprising 20% of the total data. Both training and validation datasets are generated with specified target image dimensions (299x299 pixels) and RGB color mode. The images are shuffled during each epoch to avoid bias, and the data is organized into categorical classes, which aligns with the multi-class classification task.

2.3 Handling Class Imbalance

The dataset exhibits imbalanced classes, and to address this, class weights are computed using Scikit-learn's `compute_class_weight()` function. This helps to prevent the model from being biased towards the majority class. The class weights are then assigned to each class, ensuring that the training process is adjusted accordingly to treat all classes fairly, despite their unequal representation.

2.4 Visualization

Before proceeding with model training, sample batches of augmented images are visualized using the `showImage()` function. This step helps verify that the augmentations are being applied correctly and that the images maintain visual clarity, which is crucial for accurate model learning.

2.5 Test Data

A separate `ImageDataGenerator` is used for the test set without any data augmentation, only rescaling the pixel values. The test set includes predefined classes like 'Smoke', 'fire', and 'non fire', which ensures that the model's final performance can be assessed on real-world examples. The test images are not shuffled, as preserving their order is essential for evaluation.

This systematic approach ensures that the image generation process is optimized for the training of a robust and generalized deep learning model.

3. Model Building

3.1 Building Overview

The model training and evaluation process is designed with a comprehensive approach to monitor performance and ensure optimal results. It starts by defining helper functions and callbacks, and then proceeds with training, followed by evaluation on validation and test datasets.

3.1.1 Training History Plot

A function `plotModelHistory(h)` is created to visualize key metrics over the training and validation phases. The function generates a figure with four subplots, displaying loss, categorical accuracy, recall, and precision. Each metric is plotted for both the training and validation sets, making it easy to track the model's progress and identify any issues like overfitting. It dynamically detects the relevant key names from the model's history and prints the maximum values for each metric, allowing for a quick assessment of the model's performance.

3.1.2 Custom Callback

A custom callback, `myCallback`, is implemented to stop training once a specific accuracy threshold (99%) is reached. This ensures that the model doesn't overtrain once it achieves high accuracy. The `on_epoch_end` method checks the `categorical_accuracy` metric and stops training when the threshold is met.

3.1.3 Training the Model

The `trainModel` function orchestrates the training process, leveraging the defined callbacks for early stopping, learning rate reduction, and model checkpointing. The model is compiled with the Categorical Crossentropy loss function, which includes label smoothing to enhance generalization. The metrics tracked include categorical accuracy, recall, and precision, providing a comprehensive view of the model's performance. The model is trained on the augmented dataset generated earlier, with steps per epoch and

validation steps calculated based on batch sizes. Callbacks like ReduceLROnPlateau and EarlyStopping are crucial for adaptive learning and ensuring the model doesn't overfit.

3.1.4 Model Evaluation

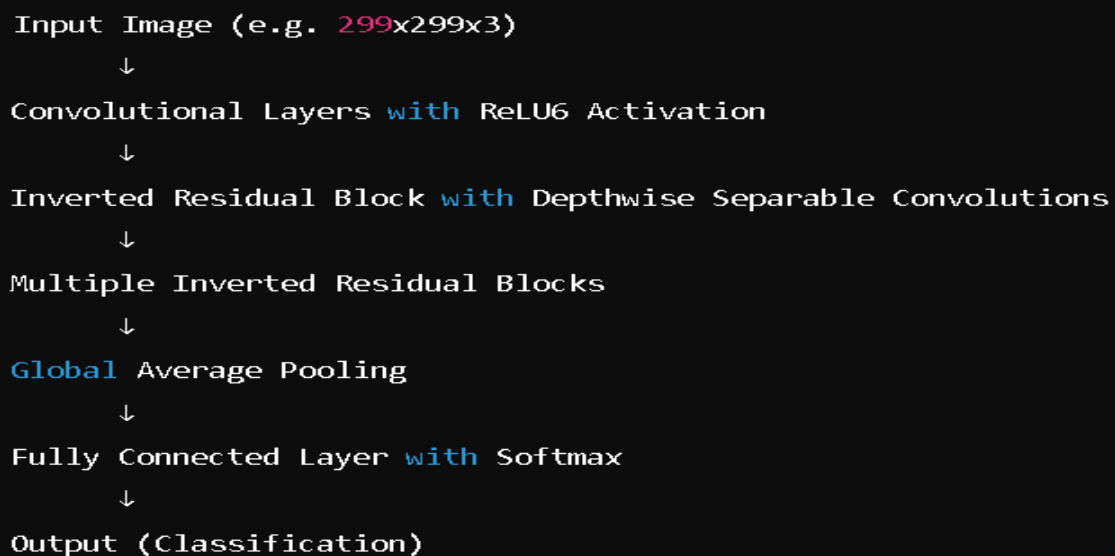
Once the model is trained, it is evaluated on both the validation and test datasets. Two functions, `evaluateModel` and `evaluateModelWithTestData`, handle the evaluation on the validation and test sets respectively. These functions calculate the loss and accuracy, while also measuring the time taken for predictions. This step is essential to assess the generalization capability of the model.

By using these functions and techniques, the model building process is made more efficient and flexible, ensuring that the best possible version of the model is saved and evaluated systematically.

3.2 Models

3.2.1 MobileNetV2

We utilized **MobileNetV2**, known for its efficiency in image classification tasks, particularly on mobile devices. The architecture is optimized with **inverted residuals** and **linear bottlenecks**, reducing the computational cost while retaining accuracy.



```
graph TD; A[Input Image (e.g. 299x299x3)] --> B[Convolutional Layers with ReLU6 Activation]; B --> C[Inverted Residual Block with Depthwise Separable Convolutions]; C --> D[Multiple Inverted Residual Blocks]; D --> E[Global Average Pooling]; E --> F[Fully Connected Layer with Softmax]; F --> G[Output (Classification)];
```

The diagram illustrates the MobileNetV2 architecture flow. It starts with an 'Input Image (e.g. 299x299x3)', followed by 'Convolutional Layers with ReLU6 Activation'. This is followed by an 'Inverted Residual Block with Depthwise Separable Convolutions', which then leads to 'Multiple Inverted Residual Blocks'. The flow continues to 'Global Average Pooling', then a 'Fully Connected Layer with Softmax', and finally the 'Output (Classification)'.

Fig 1.1

We initialized MobileNetV2 with **ImageNet pre-trained weights** and fine-tuned the later layers on our dataset. Key parameters include:

- **Loss function:** Categorical Crossentropy with label smoothing
- **Optimizer:** Adam
- **Metrics:** Categorical accuracy, recall, and precision

Training metrics, including loss and accuracy, were visualized to ensure proper learning without overfitting.

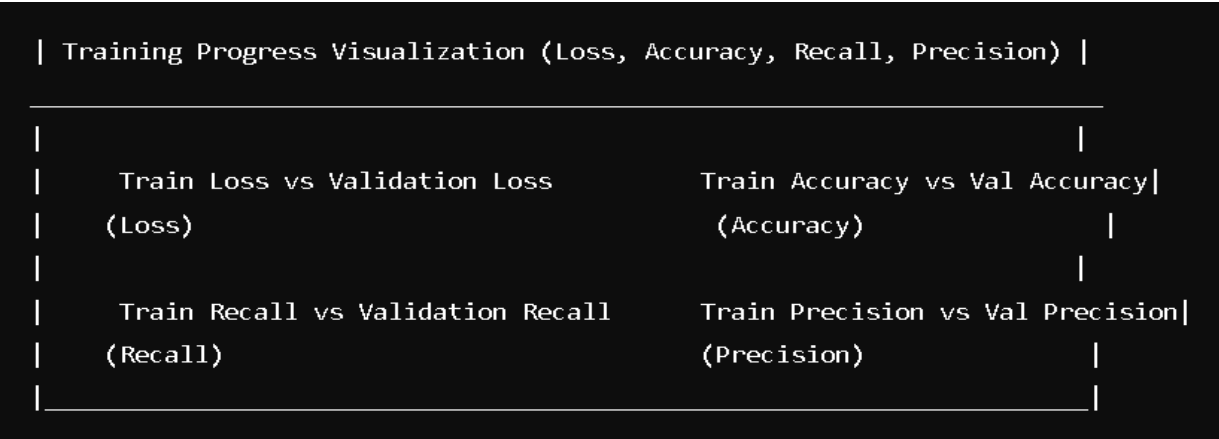


Fig 1.2

This lightweight architecture enables fast inference, making it ideal for real-time applications.

3.2.2 ResNet152

We trained a **ResNet152** model, a deep architecture designed for image classification, known for its ability to prevent the vanishing gradient problem through **skip connections**. ResNet152 stacks multiple layers with **residual blocks**, allowing gradients to flow back through shortcut connections, ensuring efficient learning in very deep networks.

Training Process

- **Loss Function:** Categorical Crossentropy
- **Optimizer:** Adam
- **Metrics:** Categorical Accuracy, Precision, and Recall
- **Fine-tuning:** The model was initialized with **ImageNet** pre-trained weights, and later layers were fine-tuned for our dataset.

Architecture Overview:

ResNet152 consists of convolutional layers, followed by **residual blocks** with bottleneck structures, and ends with **global average pooling** and a **fully connected layer**.

Here’s a working diagram that shows the flow of data through ResNet152, illustrating key components like the residual and bottleneck blocks:

This architecture enables efficient training with high accuracy, suitable for complex image classification tasks.

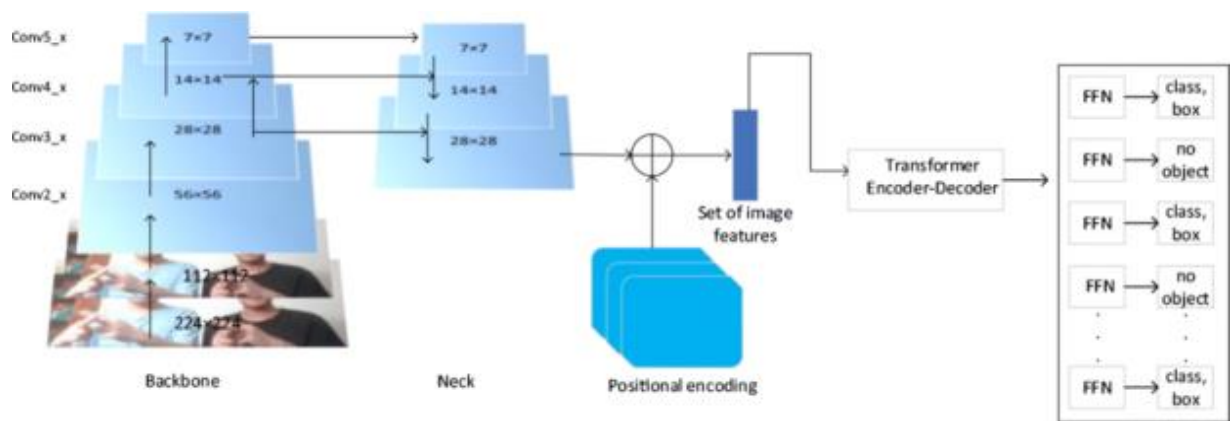


Fig 2.1

3.2.3 InceptionV3

We trained the **InceptionV3** model, which is known for its ability to handle complex image classification tasks using **inception modules**. These modules split the input into different filter sizes (1x1, 3x3, 5x5 convolutions) to capture multiple spatial features and then concatenate them to improve accuracy while minimizing computation.

Training Process

- **Loss Function:** Categorical Crossentropy
- **Optimizer:** Adam
- **Metrics:** Categorical Accuracy, Precision, and Recall
- **Fine-tuning:** Pre-trained on **ImageNet**, and later layers were fine-tuned on our dataset.

Architecture Overview:

InceptionV3 includes convolutional layers followed by multiple **inception modules**. The model ends with **global average pooling** and a fully connected layer for classification.

This architecture efficiently balances accuracy and computation, making it suitable for large-scale image recognition tasks.

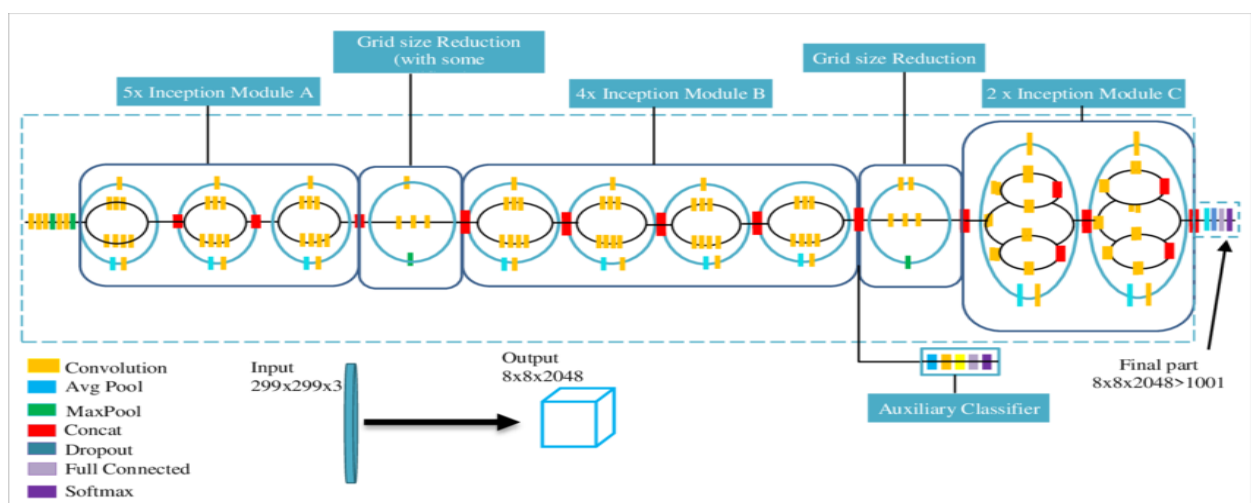


Fig 3.1

3.2.4 InceptionResNetV2

We utilized **InceptionResNetV2**, a hybrid model that combines the advantages of **Inception modules** with **ResNet-style skip connections**. This architecture helps capture multi-scale features using the inception modules while preserving gradient flow with residual connections, making it both accurate and efficient.

Training Process

- **Loss Function:** Categorical Crossentropy
- **Optimizer:** Adam
- **Metrics:** Categorical Accuracy, Precision, Recall
- **Fine-tuning:** Pre-trained on **ImageNet**, with fine-tuning for specific layers on our dataset.

Architecture Overview:

InceptionResNetV2 merges inception modules with residual blocks, followed by **global average pooling** and a fully connected layer for classification. This blend of techniques ensures high accuracy and stable training, especially in deeper networks.

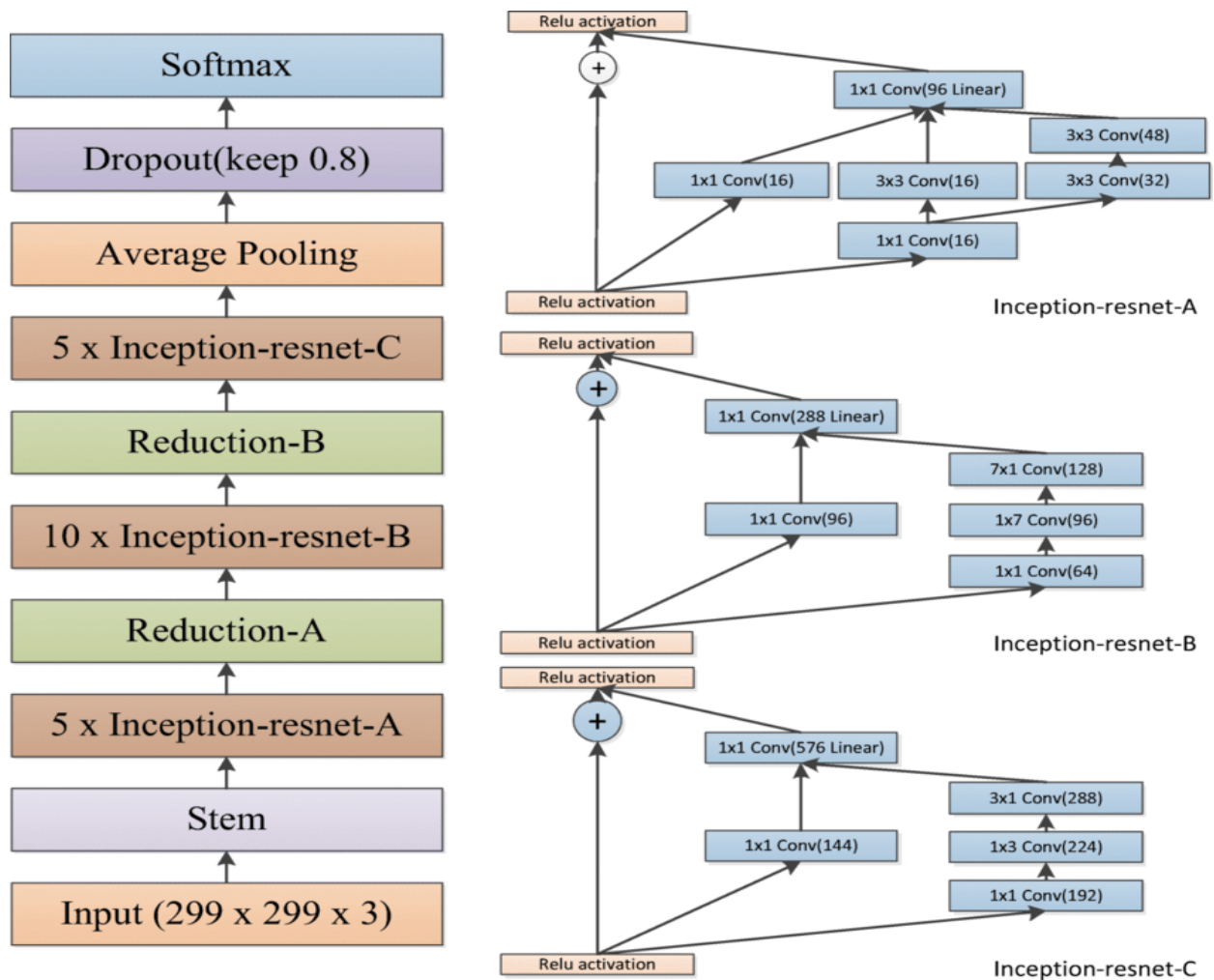


Fig 4.1

3.2.5 DenseNet169

We trained the **DenseNet169** model, which leverages **dense connections** to promote feature reuse. In DenseNet169, each layer is connected to every other layer within a dense block, allowing for better information flow and feature sharing, which reduces the number of parameters while enhancing efficiency.

Training Process

- **Loss Function:** Categorical Crossentropy
- **Optimizer:** Adam
- **Metrics:** Categorical Accuracy, Precision, Recall
- **Fine-tuning:** The model was initialized with **ImageNet** pre-trained weights, and later layers were fine-tuned for our dataset.

Architecture Overview:

DenseNet169 consists of **dense blocks** connected through **transition layers** that compress the feature maps, followed by **global average pooling** and a fully connected layer for classification. This design improves gradient flow and computational efficiency, making it effective for large-scale image recognition tasks.

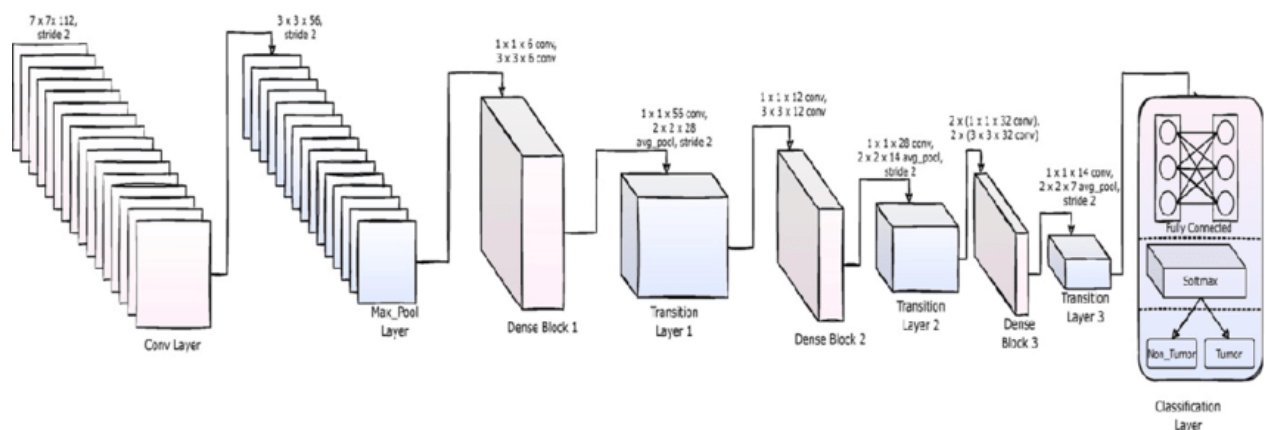


Fig 5.1

4. Evaluation

4.1 Evaluation Techniques

This research will conclude with the model evaluation and comparison with different machine learning model performances in related works. The model evaluation will be based on Recall and accuracy as **recall** is more important than precision. Recall measures the model's ability to detect actual fire/smoke cases, ensuring fewer false negatives. In such a critical scenario, missing a fire (false negative) is riskier than a false alarm (false positive), as undetected fires can lead to catastrophic consequences. Therefore, maximizing recall ensures that most fire events are detected, reducing the chances of missing dangerous incidents. Precision, while useful, is secondary to recall in this context. The confusion matrix will be

generated to calculate True Positive, True Negative, False Positive, False Negative metrics for evaluating model's efficiency.

4.2 Evaluation Models

4.2.1 For MobileNetV2

Max. Training Accuracy: 0.9631544351577759
 Max. Validation Accuracy: 0.954468309879303
 Max. Training Recall (recall): 0.9579458832740784
 Max. Validation Recall (val_recall): 0.947522759437561
 Max. Training Precision (precision): 0.9690878391265869
 Max. Validation Precision (val_precision): 0.9622256755828857

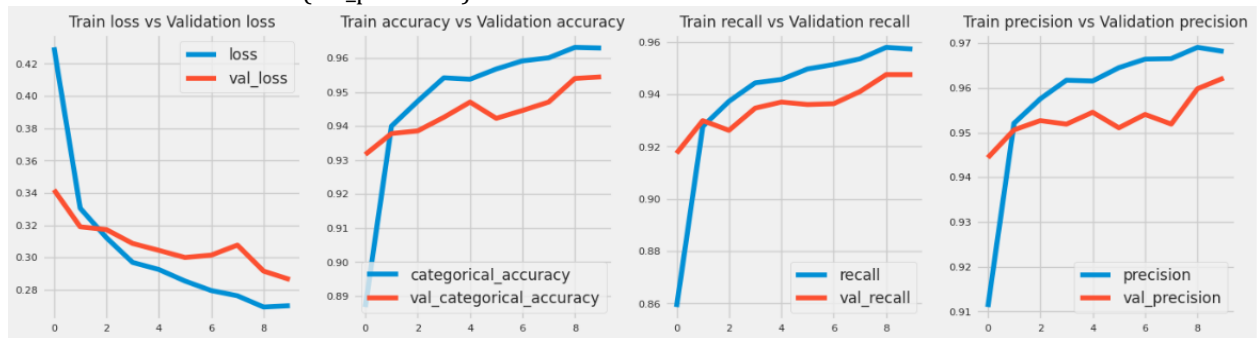


Fig 6.1

4.2.2 For ResNet152

Max. Training Accuracy 0.9713337421417236
 Max. Validation Accuracy 0.9570921659469604
 Max. Training recall 0.9672055244445801
 Max. Validation recall 0.9530791640281677
 Max. Training precision 0.974916398525238
 Max. Validation precision 0.9618142247200012

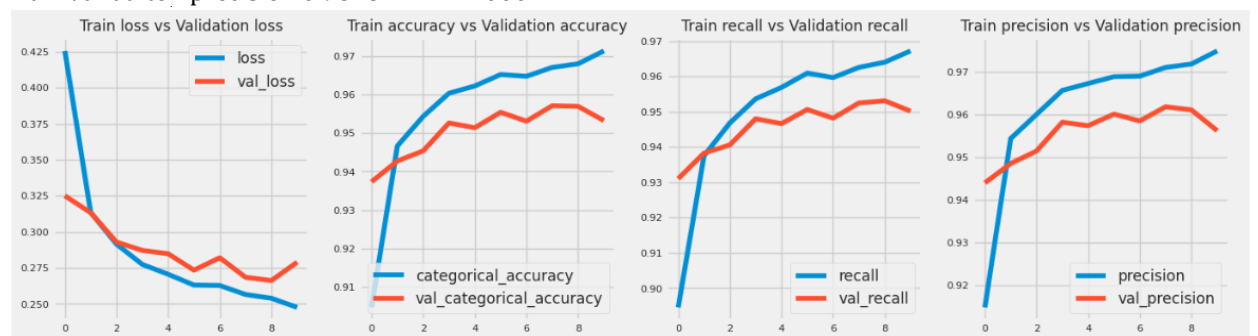


Fig 6.2

4.2.3 For InceptionV3

Max. Training Accuracy: 0.960260808467865
 Max. Validation Accuracy: 0.9478314518928528
 Max. Training Recall (recall_1): 0.9529302716255188
 Max. Validation Recall (val_recall_1): 0.9407315850257874
 Max. Training Precision (precision_1): 0.96620112657547
 Max. Validation Precision (val_precision_1): 0.9551020264625549

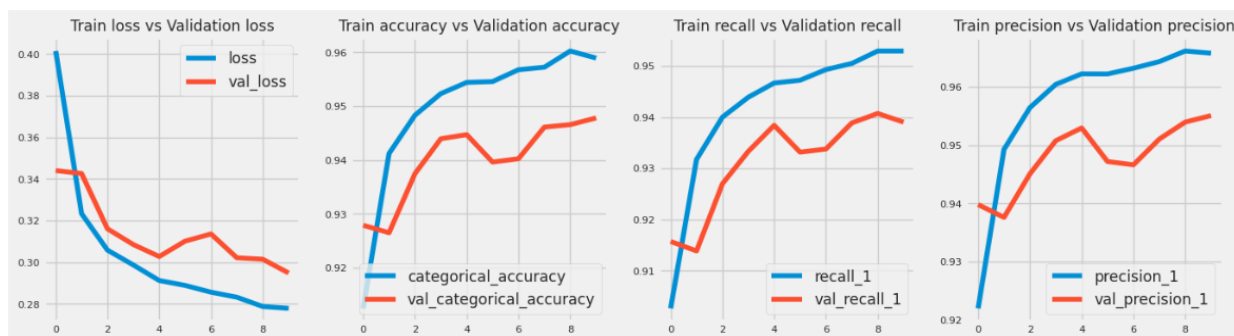


Fig 6.3

4.2.4 For InceptionResNetV2

Max. Training Accuracy: 0.9573286175727844

Max. Validation Accuracy: 0.9458249807357788

Max. Training Recall (recall_2): 0.9493421912193298

Max. Validation Recall (val_recall_2): 0.9376447200775146

Max. Training Precision (precision_2): 0.9636940360069275

Max. Validation Precision (val_precision_2): 0.9532402157783508

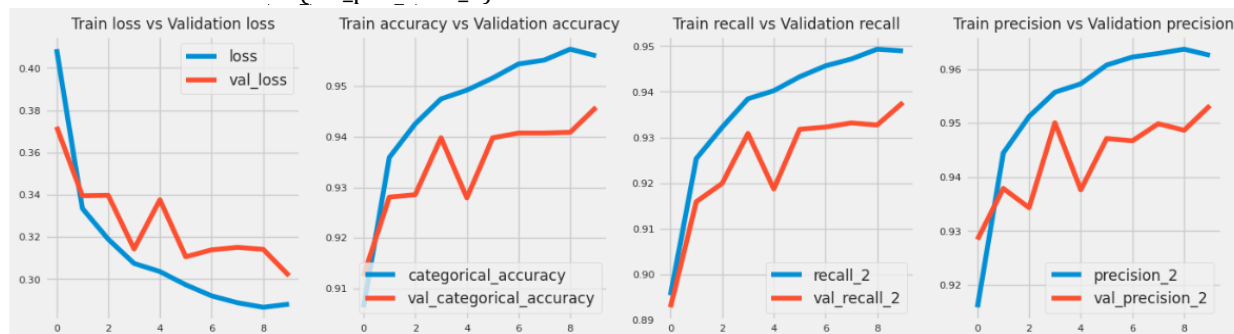


Fig 6.4

4.2.5 For DenseNet169

Max. Training Accuracy: 0.9634245038032532

Max. Validation Accuracy: 0.9597160220146179

Max. Training Recall (recall_3): 0.956672728061676

Max. Validation Recall (val_recall_3): 0.9526161551475525

Max. Training Precision (precision_3): 0.9692752957344055

Max. Validation Precision (val_precision_3): 0.9652091860771179

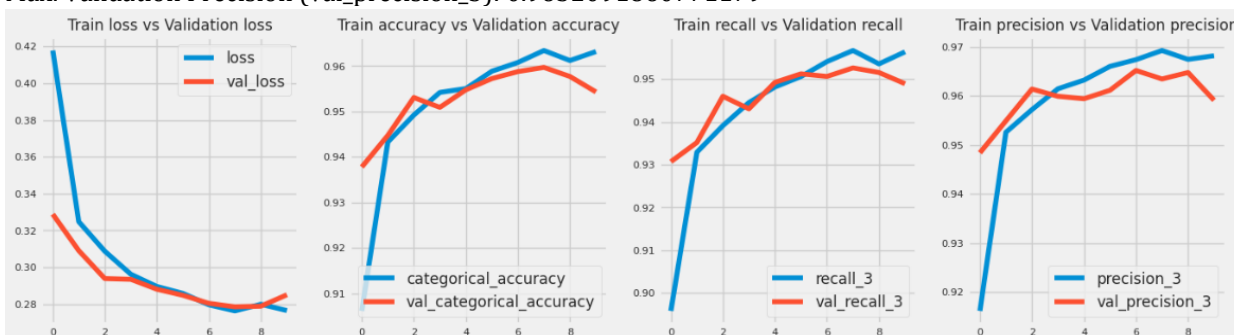


Fig 6.5

5. Conclusion

In this project, various deep learning models, including MobileNetV2, ResNet152, InceptionV3, InceptionResNetV2, and DenseNet169, were trained and evaluated for their ability to detect fire, smoke, and non-fire images. After analyzing their performance based on metrics like **accuracy, recall, and precision**, DenseNet169 emerged as the best-performing model. The primary evaluation criterion was **recall**, given its importance in minimizing the risk of missing actual fire or smoke events (false negatives).

The high recall achieved by DenseNet169 ensures that most fire incidents are detected, making it a reliable model for real-time fire detection in forest environments. This model can be integrated into surveillance systems for early fire detection, helping prevent catastrophic damage. Further improvements can be made by enhancing the dataset or using additional image preprocessing techniques for better generalization across diverse environmental conditions.

6. Future Scope

Given the pressing problem of frequent fire outbreaks and the limitations of traditional fire detection systems, this project on **Deep Learning-based Fire Detection** offers promising advancements. However, several areas can be explored for future improvements:

6.1 Integration with IoT and Smart Systems

The fire detection model can be integrated with **IoT-enabled surveillance systems**. By connecting cameras and sensors to the cloud, real-time alerts could be sent to fire fighters or emergency services, enabling faster responses.

6.2 Improved Accuracy with Multimodal Data

The current model relies primarily on visual cues (brightness, color, texture, flicker). The accuracy of fire detection can be enhanced by integrating additional data sources, such as **thermal imagery**, audio signals, or even **environmental data** (humidity, wind speed). This multimodal approach could reduce false positives and improve detection in various environments.

6.3 Enhancing Detection in Challenging Conditions

Smoke and fire detection in poor lighting, smoke-heavy, or extreme weather conditions can be difficult. Future models could utilize **infrared sensors** or enhanced image processing algorithms that work effectively under such conditions.

6.4 Edge Computing for Faster Response

Deploying the model on **edge devices** (e.g., security cameras with built-in AI capabilities) would reduce the need for data transmission to central servers, enabling faster, real-time fire detection.

6.5 Scalability and Global Implementation

The deep learning model can be trained with diverse datasets from **multiple environments and geographical regions**. This would improve its applicability in various settings, from forests and industrial sites to urban areas. Collaboration with fire management organizations worldwide could further refine the system.

6.6 Automatic Incident Control Integration

In the future, fire detection models can be paired with **automated fire control systems**. This could lead to automatic activation of sprinklers, fire curtains, and alarms, mitigating damage even before human intervention.

6.7 Incorporating Drone Technology

Future implementations can explore drone-based fire detection. Equipped with cameras and fire detection algorithms, drones could monitor large areas such as forests and industrial zones, providing real-time alerts to prevent wildfires.

By focusing on these aspects, the deep learning-based fire detection system could revolutionize fire prevention, making it more **cost-effective, efficient**, and capable of early detection, ultimately saving lives and property.

7. System Requirements

The system requirements for running the project, given the listed libraries and dependencies, include both **hardware** and **software** specifications.

7.1 Hardware Requirements

Processor: A multi-core processor (4 cores or more) is recommended for parallel computations, particularly if handling large datasets and training deep learning models.

- **Recommended:** Intel Core i5/i7 or AMD Ryzen 5/7 series.

Memory (RAM): Minimum of 8 GB RAM is required. However, for training deep learning models and handling large datasets, 16 GB or more is preferred.

GPU (Graphics Processing Unit): A dedicated GPU is highly recommended for deep learning tasks, as it speeds up the training of neural networks.

- **Recommended:** NVIDIA GPUs with CUDA support, such as the NVIDIA GTX 1080 or RTX 2080 or newer, with at least 6-8 GB VRAM.

Storage: At least 50 GB of free storage for datasets, libraries, and models. SSD is preferred over HDD for faster data access.

Display: A screen resolution of 1920x1080 or higher to visualize plots and performance metrics effectively.

7.2 Software Requirements

Operating System:

- **Recommended:** Linux (Ubuntu 20.04 LTS or later), Windows 10, or macOS 10.14+.
- **Preferred:** Linux, for better compatibility with deep learning frameworks.

Python Version: Python 3.6 or above (Python 3.8 or higher is recommended).

Key Python Libraries:

- **TensorFlow:** Version 2.x for deep learning model building and training.
- **Keras:** Integrated with TensorFlow for defining and training neural network models.
- **Pandas:** For data manipulation and analysis.
- **Numpy:** For numerical computations.
- **Matplotlib and Seaborn:** For data visualization.
- **Scikit-learn:** For preprocessing, splitting data, and performance metrics.
- **Scikit-image:** For reading and processing images.

CUDA and cuDNN (if using an NVIDIA GPU):

- CUDA 11.x and cuDNN 8.x for GPU acceleration support with TensorFlow.

7.3 Additional Libraries

- **Warnings and System Libraries:** Libraries like warnings, os, sys, and time are essential for managing system interactions and handling warnings.
- **TFSMLayer (TensorFlow Social Media Layer):** Special TensorFlow layer for advanced deep learning tasks.

7.4 Development Environment

- **Jupyter Notebook:** Ideal for running Python code interactively, debugging, and visualizing results.
- **Anaconda/Miniconda:** For managing Python environments and dependencies easily.

7.5 Internet Connection

A stable internet connection is recommended for downloading datasets, pre-trained models, and dependencies from online repositories.

By meeting these hardware and software requirements, the system will be equipped to handle the project's tasks efficiently, especially when working with large datasets and training deep learning models.

8. References

- Adarsh Vulli, Parvathaneni Naga Srinivasu, Sai Krishna Sashank Madipally, Jana Shafi(2022). Fine-Tuned DenseNet-169 for Breast Cancer Metastasis Prediction Using FastAI and 1-Cycle Policy. DOI:10.3390/s22082988
- Cheng Peng, Yikun Liu, Xinpan Yuan, Qing Chen(2022). Research of image recognition method based on enhanced inception-ResNet-V2. DOI:10.1007/s11042-022-12387-0
- Orlando Iparraguirre-Villanueva, Victor Guevara-Ponce, Ofelia Roque Paredes, Fernando Alex Sierra-Liñan(2022). Convolutional Neural Networks with Transfer Learning for Pneumonia Detection. DOI:10.14569/IJACSA.2022.0130963
- Yu Liu, Parma Nand, Md. Akbar Hossain, Minh Nguyen(2023). Sign language recognition from digital videos using feature pyramid network with detection transformer. DOI:10.1007/s11042-023-14646-0