Fire Detection CNN Models

GROUP #5

DR. NAHED A. ALOWIDI

RYAM ABDULWAS

GHADIA ALSAYED

SHAHAD HATEM BALKHASIR

ASMA SALEH ALWAAL

RAWAN ALGHAMDI

RAHAF KOSHK

GHAIDA KHALID ALZAHRANI

WEJDAN ALSULAMI

CPCS432 2023, Dec 21

Summary

A fire is an unusual event that, over a relatively short period of time, can cause significant damage to property and human life. Human error or system failure is the main cause of fire disasters, resulting in significant loss of life as well as other damage. Traditional fire alarms use sensors that must be close together in order to be triggered. To check the fire requires human intervention. To overcome these limitations, real-time vision-based fire detection has been enabled in monitoring devices. Once a fire appears on any camera, the approaches can detect it and control the fire area.

To address fire detection through image classification, this project conducted research on related methods for fire recognition in images. The team explored the potential of deep learning to solve this problem and summarized the results for comprehensive understanding. Four different computer vision models: — InceptionV3, VGG16, Resnet50, and YOLO — were chosen for this task, each with unique architectures and pros and cons that are discussed and highlighted in the project report.

Using the Kaggle dataset, the team trained each model and evaluated it based on accuracy and loss metrics, splitting the dataset into training and testing subsets. Leveraging transfer learning from the ImageNet dataset facilitates faster and more effective training.

After training, YOLO achieved 67% accuracy, VGG16 reached 95.98%, InceptionV3 achieved 97%, and Resnet50 achieved 97.51% accuracy in fire image classification. Despite performance differences across models, Resnet50 demonstrated superior accuracy in fire image classification compared to other models and emerged as the best-performing model for this task.

Contents

1. Introduction

1.1. Description of the project idea

Imagine a world where fires are detected and extinguished before they can cause severe damage or injury, firefighters are immediately alerted to fires to respond quickly and save lives, and businesses and homes are protected from devastating fires. The environment is safeguarded from the harmful effects of wildfires.

Computer vision is a field of artificial intelligence that teaches machines to see and understand the world by using cameras and sophisticated algorithms to detect fires faster and more accurately than humans to monitor large areas 24/7. This technology has the potential to revolutionize fire detection and response. Traditional fire detection systems rely on smoke or heat detectors, which can be slow to respond and prone to false alarms. As a result, an effective automated method of identifying fires is required.

This project endeavors for the detection of fires using images taken from surveillance cameras. The dataset is preprocessed to make it suitable for transfer learning tasks. Different pre-trained convolutional neural network (CNN) variants are utilized, including Yolo, Resnet50, VGG16, and Inception-v3.

The goal of this work is to create a machine-learning model that will determine the presence of fire in any environment from surveillance cameras. Finally, we will translate the problem into the language of machine learning with the task of binary classification, preprocessing of the input data, as well as the construction of a machine learning model, which will be done using TensorFlow and Keras.

1.2. Related Work

Deep learning-based fire detection, leveraging Convolutional Neural Networks (CNNs), revolutionizes our ability to identify and respond to fire incidents with remarkable accuracy. This technology is crucial for safeguarding lives and property, reducing damage, and enhancing the efficiency of controlling fire-related incidents. The importance of early fire detection is paramount in our lives, providing preemptive warnings for swift responses and mitigating potential dangers. Many research endeavors are actively seeking to advance mechanized models for fire detection, contributing to the ongoing evolution and improvement of automated systems dedicated to enhancing the accuracy and efficacy of fire detection mechanisms:

[19], a comprehensive fire detection and alert system is presented within a deep learning framework. The study employs various models, including CNN, Inception ResNet, and Inception V3, alongside a modified Inception-V3 model. Notably, the modified Inception-V3 model stands out, demonstrating superior performance as shown in the table below.

[2] This research examined four machine/deep learning architectures and algorithms, as well as two new modified models, for fire detection by using specific dataset. Before being compared the models for performance, they were fine-tuned for optimal performance hyper-parameters.

[3] The result of the paper is that training Faster Region-based Convolutional Neural Networks (Faster R-CNN) achieves excellent detection effect, reducing time cost and improving proposal quality. However, the paper concludes that YOLO (You Only Look Once) is a convolutional neural network that supports end-to-end training and testing and can detect and recognize multiple targets in images with certain accuracy, indicating that YOLO is superior in terms of efficiency and accuracy compared to Faster R-CNN.

[4] The research compares the performance of different YOLO series models for object detection in UAVs. The study evaluates YOLOv3, YOLOv3-tiny, YOLOv3-SPP3, YOLOv4, and YOLOv4-tiny using mAP and FPS as evaluation metrics. Results show that YOLOv4 achieves an mAP of 87.48%, which is 14.2% higher than YOLOv3, with an FPS of 72. The research concludes that YOLOv3-tiny is suitable for real-time, lightweight, and high-precision applications.

[5] The article investigates the use of YOLO models for real-time steel surface defect detection. YOLOv5, YOLOX, and YOLOv7 models are compared using the NEU-DET dataset. The trained models are evaluated on Nvidia Jetson Nano and Jetson Xavier AGX devices for real-time performance. Optimization techniques such as exporting to TensorRT and reducing image size are applied to improve detection speed. The study addresses the challenges of limited datasets and real-time application in steel surface defect detection.

[6] The reacher discusses the use of an improved YOLOv8 model for detecting student behavior in classroom videos. The model addresses challenges such as object density, occlusion, and multi-scale scenarios in classroom images. It introduces a novel C2f_Res2block module, along with MHSA and EMA, integrated into the YOLOv8 model. Experimental results show that the improved model outperforms the original YOLOv8, achieving a 4.2% increase in average precision. The article highlights the importance of deep learning and computer vision techniques in analyzing and enhancing teaching effectiveness in smart education.

[7] The reachers provides a comprehensive analysis of the YOLO framework's evolution, from its original version to the latest iterations. It explores the innovations and improvements made in network architecture, training techniques, and performance trade-offs. The review highlights the diverse range of applications where YOLO has been successfully utilized, including robotics, autonomous vehicles, surveillance, agriculture, medicine, and traffic management. The document also discusses object detection metrics such as Average Precision (AP) and Non-Maximum Suppression (NMS).

The paper [9] discusses the development of an accurate fire detection model. The authors use a Residual Network (ResNet) to extract depth features from the image. Depth features are important in image recognition tasks as they provide information about the spatial arrangement of objects in the image. The model can better understand the image by extracting these features and making more accurate predictions. One of the main benefits of using ResNet is its ability to mitigate the problem of vanishing gradients. This is a common issue in deep neural networks where the gradients of the loss function become very small as they are backpropagated through the layers of the network. This can lead to the model needing to be faster to train or even getting stuck. ResNet addresses this issue using "skip connections" or "shortcuts" that allow the gradient to be directly backpropagated to earlier layers. The authors also use an improved version of ResNet to solve the problem of shallow features quickly disappearing. This likely refers to the fact that features learned by early layers in deep

networks can become less important as the network depth increases. Using an improved ResNet, the authors ensure that essential features known by early layers are preserved and used in the final predictions.

[11], The proposed research work utilizes Convolutional Neural Networks (CNNs) and Deep CNNs with transfer learning for fire detection in images. The performance of fire detection is compared with traditional CNN, as well as two deep learning models: VGG16 and MobileNet. To create the dataset, images are downloaded from various internet resources, resulting in a highly imbalanced dataset. The dataset is divided into training and testing data, with 1205 default (non-fire) and 1580 fire images in the training set, and 237 default and 493 fire images in the testing set.

[13], This study proposes a fire detection framework for images utilizing image processing, motion detection, and convolutional neural networks (CNNs). A dataset comprising 1141 fire zone images and 1900 non-fire zone images was created by combining various fire datasets from the literature. Three methods were employed: image processing algorithms to extract flame areas, motion detection to identify flame movements, and CNN models trained using transfer learning. Four different CNN models (Inception V3, SqueezeNet, VGG16, and VGG19) were trained using 3041 images for fire detection. The models were evaluated using a 10-fold cross-validation methodology. The highest classification success rate of 98.8% was achieved by the Inception V3 model. The fire detection framework, combining image processing, motion detection, and CNN models, demonstrated sufficient success in detecting fire. It enables decision-making regarding the presence of fire in images and can contribute to early fire detection and precautionary measures. Further improvement in classification performance can be achieved by enhancing the fire dataset. The framework holds promise for integration into automated fire extinguishing systems.

[14], presents a transfer learning model for image classification, which involves pre-processing image data and applying data augmentation techniques like shearing and flipping. The model uses models like VGG16, ResNet50, and DenseNet121 to reduce overfitting and improve classification accuracy.

[15] , This study focused on an intelligent approach using different deep-learning models for fire detection implementing deep learning models. Inceptionv3 has been thoroughly tried, tested, and monitored to evaluate its performance and accuracy as a fire detection system. The model aims, by taking advantage of convolutional neural network layers and pooling layers, to classify images as fire or non-fire. It will send an alert message when a fire image is identified. To train the model, a dataset of 1,820 fire images was used, trained using both InceptionV3 and CNN models. Before training, data augmentation techniques were applied to enhance the dataset, with the aim of improving model performance.

[3] The result of the paper is that training Faster Region-based Convolutional Neural Networks (Faster R-CNN) achieves excellent detection effect, reducing time cost and improving proposal quality. However, the paper concludes that YOLO (You Only Look

Once) is a convolutional neural network that supports end-to-end training and testing and can detect and recognize multiple targets in images with certain accuracy, indicating that YOLO is superior in terms of efficiency and accuracy compared to Faster R-CNN.

The paper [16] discusses the proposed image fire detection algorithms based on Faster-RCNN, R-FCN, SSD, and YOLO v3. These algorithms have shown promising results in terms of accuracy, speed, and robustness. They have the potential to enhance early fire detection systems and contribute to reducing fire losses. Further research and improvements can focus on refining the algorithms, exploring additional datasets, and addressing specific challenges in real-world fire detection scenarios.

The paper addresses [17] the critical issue of ship fires, emphasizing their impact on maritime safety and the limitations of existing detection methods. Proposing a solution, the study introduces a lightweight ship fire detection model based on the modified YOLOv4-tiny algorithm. The model is designed to overcome challenges posed by the ship environment, considering factors like limited space and computational resources. The methodology involves constructing a dataset, applying multi-scale detection, using the SE attention mechanism, and incorporating picture transformation and migration learning. The proposed model is positioned as a practical, efficient solution for real-time ship fire detection in demanding maritime conditions. The paper outlines sections on the current state of fire detection technologies, details of the proposed model, experimental settings, and a comparative analysis of results.

Reference	Dataset	Models	Accuracy
[19]	Fire and No_Fire data set from Kaggle with 980 images for Training and 239 images for Validation.	CNN	95.40%
		Inception-ResNet V2	93.88%
		Inception V3	94.20%
		Modified Inception V3	95.92%
[7]	Create new dataset, includes a total of 986 RGB images of size 250×250	Proposed Modified ResNet-50	100 %
		Xception	100%
		MobileViT	99.797%
		Proposed Modified Xception	99.495%
		ResNet-50	99.495%
		SVM	90.909%
[11]	Dataset is divided into training and testing data, with 1205 default (non-fire) and 1580 fire images in the training set, and 237 default and 493 fire images in the testing set.	CNN	91%
		VGG16	99%
		MobileNet	98%
[13]		InceptionV3	98.8%

	Dataset comprising 1141 fire zone images and 1900 non-fire zone images was created by combining various fire datasets from the literature.	SequeezeNet	97%
		VGG16	97.3%
		VGG19	96.8%
[14]	The dataset is split into train 80% and test data 20% .	Resnet50	92.27%
		VGG16	85.21%
		Densenet121	89.04%
[15]	A dataset containing 1820 images of fire. Before training the model, data augmentation is performed through image zoom and rotation techniques.	Inception V3	99%
		CNN Model	89%
[16]	29,180images were gathered from various sources, including small public fire image/video databases, large public image/video datasets, previous experimental data from research institutions, and the Internet.	YOLOv3	99.62%
		SDD	98.93%
		R-FCN	99.20%
		Faster-RCNN	99.43%
[17]	The dataset was created by collecting multiple fire images featuring a ship's cabin, deck, and dwelling. Additionally, image transformation techniques were applied to expand the dataset.	SSD	82.6%
		OLOv4-tiny	85.1%
		YOLOv3-tiny	76.5%
		I-YOLOv4-tiny + SE	92.8%

1.3. Approach Similarity and Difference

All techniques use the same data and attempt to solve the same problem, which is to develop a deep learning model that identifies if fires exist or not in photos. The significant distinction between each strategy is the procedure each algorithm takes. For example, in VGG16, we loaded the pre-trained model with ImageNet weights as the base model, and then we added custom classification layers on top of the frozen VGG16 base. These additional layers include a flattening layer, followed by four dense layers with 128, 64, 32, and 1 units, respectively.

2. Technical description

2.1. Dataset

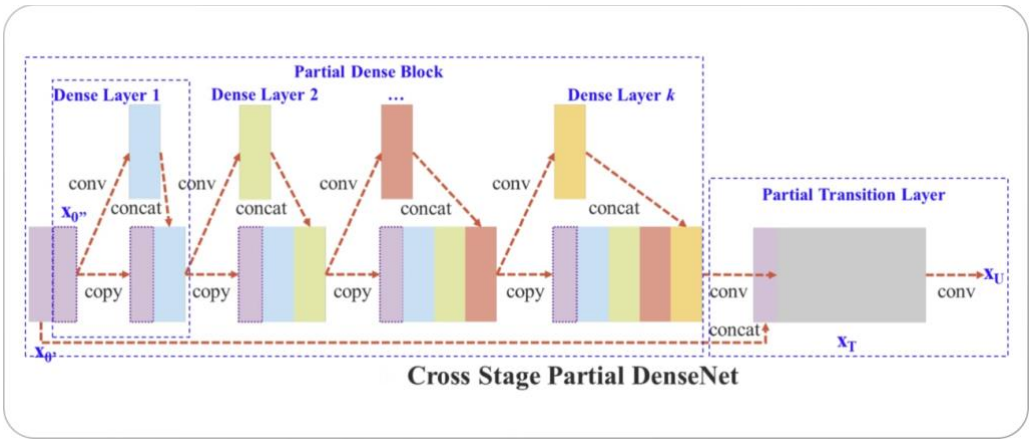
We have extracted a dataset from the Kaggle website, which includes images featuring forests both with and without fires. The dataset is organized into 2 folders (fire and non-fire images) and contains subfolders for each image category (Burning forests / Unburned forests). There are contains 755 outdoor-fire images some of them contain heavy smoke, and 244 nature images (e.g.: forest, tree, grass, river, people, foggy forest, lake, animal, road, and waterfall.

2.2. Methods

2.2.1. YOLO (You Only Look Once)

The YOLO (You Only Look Once) method is a popular real-time object detection algorithm in the field of computer vision and artificial intelligence. YOLO revolutionized object detection by providing a faster and more efficient approach compared to traditional methods. The main idea behind YOLO is to treat object detection as a regression problem, where a single neural network predicts bounding boxes and class probabilities directly from the input image. Instead of dividing the image into a grid and applying classifiers to each grid cell, YOLO uses a single neural network to make predictions across the entire image. However, it's essential to train and fine-tune the YOLO model using diverse fire datasets to ensure its reliability and effectiveness in real-world scenarios.

The YOLO (You Only Look Once) method has emerged as a leading real-time object detection algorithm in computer vision and artificial intelligence. Its innovative approach has revolutionized the field by delivering faster and more efficient object detection compared to traditional methodologies. YOLO addresses object detection as a regression problem, leveraging a single neural network to directly predict both bounding boxes and class probabilities from the input image. This differs from the conventional grid-based approach where classifiers are applied to individual grid cells. By considering the entire image holistically, YOLO achieves remarkable speed and accuracy in detecting objects, including fires. The image below shows YOLO's method of recognizing different objects in the image.



2.2.1.1. Why is Using YOLO Right?

YOLO approach enables real-time object detection, including fire detection, as it quickly processes images and provides accurate bounding box coordinates for precise localization of fires. Additionally, YOLO's efficiency and generalizability make it a suitable choice for fire detection applications, allowing for integration into existing computer vision systems and the ability to detect various types of fires in quick time.

2.2.1.2. Pros and Cons of YOLO

Pros of the YOLO Approach:

- Real-time Detection: YOLO is known for its speed and can perform object detection in real-time, making it suitable for fire detection applications that require quick response.

- Single Pass Detection: YOLO processes the entire image in a single pass, enabling it to capture global context and handle overlapping objects efficiently.

- Transfer Learning: By using a pre-trained YOLO model and fine-tuning it on our fire detection dataset, we can leverage the model's learned features, reducing the need for training from scratch.

Cons of the YOLO Approach:

- Small Object Detection: YOLO may struggle with detecting small fires or flames, as they can be challenging to represent adequately in the predefined grid cells.

- Occlusion Challenges: If the fire is partially occluded or hidden by
- other objects, YOLO may have difficulty accurately detecting it.

2.2.1.4. YOLO Code

This Python code showcases the usage of Ultralytics' YOLO (You Only Look Once) object detection models. The script demonstrates how to load a pre-trained YOLOv8n model using Ultralytics and perform various operations. Initially, it imports the necessary libraries, including the Ultralytics package, and loads the YOLOv8n model from a pretrained file "yolov8n.pt.""yolov8n.pt." *It then exemplifies training the model for 10 epochs using training data specified in "trainfire.yaml" and subsequently evaluates the model's performance on the validation set using the* `model.val()` *method. There are also commented-out lines that demonstrate how to perform inference on an image (*`model("https://ultralytics.com/images/bus.jpg")`*("https://ultralytics.com/images/bus.jpg")) and how to export the trained model to the ONNX format (*`model.export(format="onnx")`*). These actions illustrate the basic workflow of loading, training, evaluating, and potentially using a YOLO object detection model using the Ultralytics library.*

C: > Users > 1riya > OneDrive > Desktop > AI - project > yolo_fire_train.py > ...

```python
# https://github.com/ultralytics/ultralytics
# pip install onnx
# pip install protobuf==3.20.*

from ultralytics import YOLO # pip install ultralytics

# Load a model
# model = YOLO("yolov8n.yaml")  # build a new model from scratch
model = YOLO("yolov8n.pt")  # load a pretrained model (recommended for training)

# YOLOv8n   640 37.3    80.4    0.99    3.2     8.7
# YOLOv8s   640 44.9    128.4   1.20    11.2    28.6
# YOLOv8m   640 50.2    234.7   1.83    25.9    78.9
# YOLOv8l   640 52.9    375.2   2.39    43.7    165.2
# YOLOv8x   640 53.9    479.1   3.53    68.2    257.8

# Use the model
model.train(data="trainfire.yaml", epochs=10)  # train the model
metrics = model.val()  # evaluate model performance on the validation set
# results = model("https://ultralytics.com/images/bus.jpg")  # predict on an image
# path = model.export(format="onnx")  # export the model to ONNX format
```

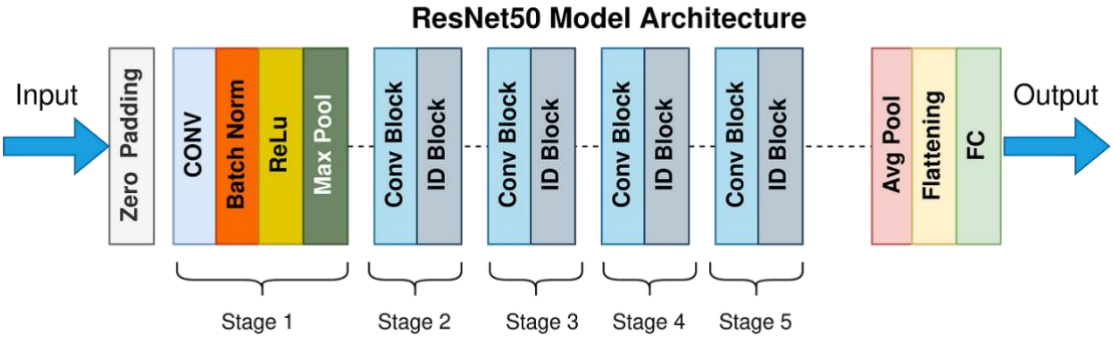
2.2.2. Transfer learning (Resnet50)

ResNet, short for Residual Network, is a convolutional neural network (CNN) architecture devised by He Kaiming in 2015 (He, 2016). The term "ResNet" reflects its reliance on residual blocks embedded within the network's structure. This design was crafted to tackle the challenge of augmenting neural network depth without consistently improving performance outcomes (Wisdomml, 2023).

The architecture is based on the concept of residual learning, which allows for the training of deeper networks without the vanishing gradient problem. The model's architecture comprises 50 layers, including convolutional, pooling, and fully connected layers. It is composed of residual blocks, which contain skip connections that allow for the flow of information from one layer to another without passing through the

intermediate layers to preserve the gradient and allow for the training of deeper networks (GeeksforGeeks, 2023).

In the context of developing a fire detection system, ResNet's underlying principles, including the use of residual blocks and skip connections, can be leveraged to enhance the model's ability to discern intricate features associated with fires.



2.2.1.1. Why is Using Resnet50 Right?

ResNet-50 model is essential because it is a powerful tool for image recognition tasks. It has been widely used in various fields, including medical imaging, autonomous driving, and natural language processing. Its ability to learn complex patterns in images makes it particularly suitable for tasks like fire detection, where the patterns can be very complex and subtle. The model differs from other models using "skip connections" to solve the vanishing gradient problem allows it to be trained on much deeper networks without suffering from the issue of poor performance on earlier layers. Other models, such as VGGNet or Inception, use a different technique requiring more layers to achieve similar performance.

2.2.1.2. Pros and Cons of Resnet50

ResNet-50, a prominent convolutional neural network architecture, has gained widespread recognition for its deep structure and success in image-related tasks. However, like any model, it comes with both advantages and disadvantages.

Pros:

- It has been demonstrated to outperform standard deep neural networks in a variety of benchmark datasets, reaching peak performance.
- It provides more accessible training and faster convergence with jump connections that allow for the direct flow of gradients.
- Transfer learning models can be fine-tuned on a smaller dataset for successful use in real situations when labeled data availability is restricted.

Cons:

- It is more complicated than typical deep neural networks because to the inclusion of hop connections, which can result in increased processing demands and memory needs.
- The complexities of ResNets can make it difficult to understand their internal workings and decision-making processes.

2.2.1.3. Resnet50 Code

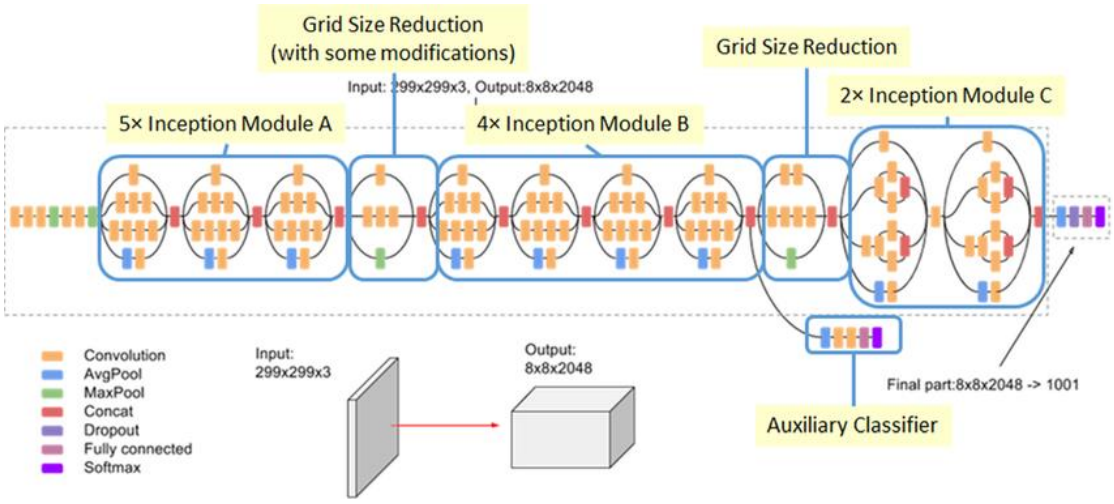
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.applications import ResNet50

# Load pre-trained ResNet50 with imagenet weights
pre_trained_resnet_50 = ResNet50(include_top=False, input_shape=(300, 300, 3), pooling='avg', weights='imagenet')

# Freeze layers up to the 39th layer
for layer in pre_trained_resnet_50.layers[:39]:
    layer.trainable = False

# Build the model
model = Sequential()
model.add(pre_trained_resnet_50)
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.summary()
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(lambda epoch: 1e-8 * 10**(epoch/20))
# Using tensorflow's Model-Checkpoint to save best model having less validation loss
model_checkpoint = tf.keras.callbacks.ModelCheckpoint(filepath='/content/final_model.h5',monitor='val_loss',save_best_only = True)
# Using Adam's optimizer to optimize our model to better learn on our dataset
model.compile(optimizer = tf.keras.optimizers.Adam() ,loss = 'binary_crossentropy',metrics = 'accuracy')
# Early stopping callback
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
```

2.2.3. inceptionV3 use CNNs

Inception v3 is a CNN an architecture devised by Google's research team, that serves the purpose of image classification and recognition. incorporating improvements like factorized convolutions and auxiliary classifiers to enhance gradient flow during training. (Szegedy et al., 2016)

The total number of layers in Inception v3 is about 48, although the precise number of layers varies depending on how the "layers" in the neural network design are interpreted, which include computational components such as convolutional, pooling, and fully connected layers.(*Inception-v3 Convolutional Neural Network - MATLAB Inceptionv3*, n.d.)

However, the architecture's building blocks or modules (beginning modules) of these modules are repeated across the network. The precise number and placement of these components contribute significantly to the structure's depth and intricacy. The repetition of these modules aids in the efficient learning of hierarchical features at various levels. The inceptionV3 model for identifying various things in an image is displayed in the picture below.



2.2.1.1. Why is Using inceptionV3 Right?

The proven effectiveness of InceptionV3 in image classification tasks - especially when trained on datasets such as ImageNet - justifies its often use InceptionV3 in computer vision applications because of its powerful performance and ability to learn a wide range of features at different levels. Using an auxiliary section and type model greatly improves the gradient flow during training, yielding accurate and efficient results.

2.2.1.2. Pros and Cons of inceptionV3

Pros:

- High Accuracy: With competitive performance on multiple benchmarks, this algorithm is renowned for its remarkable accuracy in picture classification tasks.
- Effective Architecture: This strikes a balance between computing efficiency and model complexity, making it appropriate for real-time applications.
- Transfer Learning Capabilities: Effective transfer learning across a range of computer vision applications is made possible by pre-training on large-scale datasets such as ImageNet.
- Multi-scale Feature Extraction: This technique efficiently extracts features at many scales by using parallel routes and a variety of filter sizes, including 1x1, 3x3, and 5x5.
- Factorized Convolutions: Using factorized convolutions lowers processing overhead without sacrificing efficiency (Hindarto, 2023).

Cons:

- The InceptionV3 architecture, although highly efficient, may not always result in immediate quality gains for all tasks, especially if there is not enough labeled data for training.
- The computational cost of InceptionV3 can be relatively high, with a computational cost of 5 billion multiply-adds per inference and using less than 25 million parameters.
- The architecture may not be optimal for all use cases, such as mobile vision and big-data scenarios, where computational efficiency and low parameter count are crucial.
- The use of ensemble models and multi-crop evaluation, although improving performance, can increase the complexity and computational requirements of InceptionV3.
- The introduction of auxiliary classifiers in InceptionV3 may not result in improved convergence early in training and may only slightly improve accuracy near the end of training.
  (Szegedy et al., 2016)

2.2.1.3. inceptionV3 Code

Transfer learning was applied using ImageNet weights, and some layers were adjusted to achieve acceptable results for the fire classification problem. The adjustments of layers were using the base_InceptionV3_model then adding a GlobalAveragePooling2D layer, a Dropout layer of 0.2 ratio, a Dense layer with 512 values, and a Dropout layer of 0.2 rate. Lastly, changing the activation function of the output layer to sigmoid.

```
[ ] from keras.models import Sequential
    from keras.layers import GlobalAveragePooling2D, Dense, Dropout, Flatten

    base_InceptionV3_model = InceptionV3(input_shape=[150, 150,3], include_top=False
                                         , weights="imagenet")
    InceptionV3_model= Sequential()
    InceptionV3_model.add(base_InceptionV3_model)
    InceptionV3_model.add(GlobalAveragePooling2D())
    InceptionV3_model.add(Dropout(0.2))
    InceptionV3_model.add(Dense(512))
    InceptionV3_model.add(Dropout(0.2))
    InceptionV3_model.add(Dense(1, activation= "sigmoid"))
    InceptionV3_model.compile(optimizer='adam', loss= 'binary_crossentropy',metrics=['acc'])
```

InceptionV3 model using ImageNet dataset weights

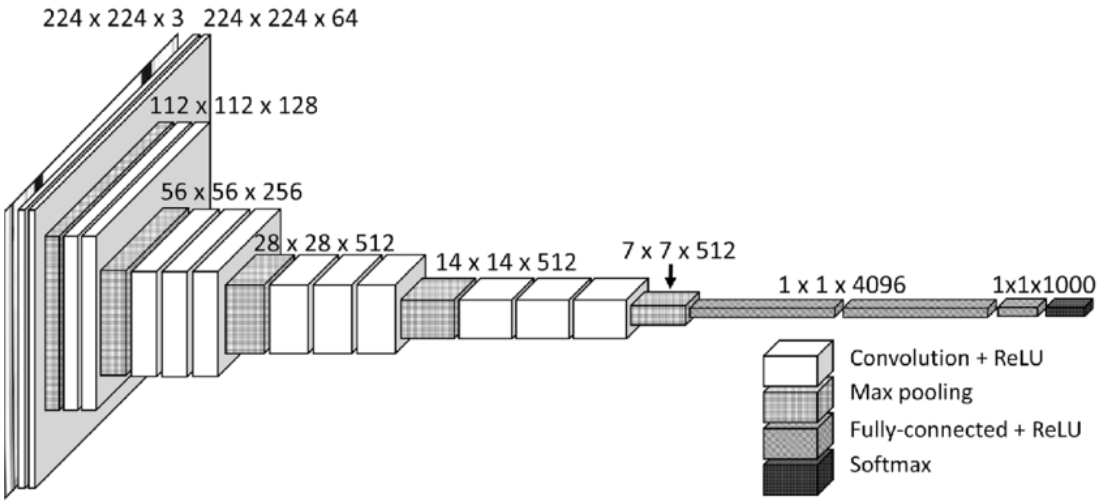
2.2.4. VGG16

VGG16, also known as Visual Geometry Group 16, is a specific architecture of a deep learning model called a convolutional neural network (CNN). It was developed by researchers at the University of Oxford and gained recognition for its effectiveness in image classification tasks.

VGG16 consists of 16 layers, including 13 convolutional layers and 3 fully connected layers. The convolutional layers are responsible for extracting features from the input images, while the fully connected layers are responsible for the final classification based on those extracted features.

The key characteristic of VGG16 is its deep structure, meaning it has many layers stacked on top of each other. This depth allows the model to learn increasingly complex and abstract representations of the input images as it progresses through the layers. VGG16 employs 3x3 convolutional filters with a stride of 1 pixel and padding of 1 pixel, and it utilizes max pooling layers to down sample the spatial dimensions of the learned features.

VGG16 has been widely used as a benchmark model for image classification tasks due to its strong performance. While there have been more advanced architectures developed since VGG16's introduction, it remains popular in the field due to its simplicity and effectiveness.



2.2.1.1. Why is Using VGG16 Right?

Using VGG16 for fire detection is a suitable choice because of its strong performance in image classification tasks and the availability of pre-trained models. With its deep architecture and learned image features, VGG16 can effectively identify visual cues associated with fire. By fine-tuning VGG16 on a fire-related dataset, it can be trained to accurately classify images as fire or non-fire. The advantage of using pre-trained models also allows for easier implementation and transfer learning, making VGG16 a practical option for fire detection applications.

2.2.1.2. Pros and Cons of VGG16

Strong Performance:

- VGG16 has a proven track record of achieving high accuracy in image classification tasks. It has been widely used and benchmarked, showcasing its effectiveness in accurately categorizing images.
- Transfer Learning: VGG16 has been pre-trained on large-scale datasets like ImageNet, enabling effective transfer learning. It can be fine-tuned on specific datasets with limited labeled data, saving time and resources.
- Simplicity: VGG16 has a straightforward architecture, making it easier to understand, implement, and modify compared to more complex models. This simplicity makes it a popular choice for researchers and practitioners who are new to convolutional neural networks or want a solid foundation for learning.
- Availability of Pre-trained Models: Pre-trained VGG16 models are widely available, allowing researchers and developers to leverage the learned representations without the need for extensive computational resources or large annotated datasets. This accessibility makes it convenient for those who may not have the means to train models from scratch.

Cons of VGG16:

- Memory Usage: VGG16's memory requirements can be high, especially when dealing with large images or datasets. This can limit its usability in scenarios where memory availability is limited.
- Overfitting: Like any deep learning model, VGG16 can be prone to overfitting, especially when the training dataset is small or imbalanced. Proper regularization techniques and data augmentation strategies should be employed to mitigate this risk.

Localization Capability: VGG16 primarily focuses on image classification and lacks explicit object localization capabilities. It may not be the most suitable choice for tasks that require precise object localization within images.

2.2.1.3. VGG16 Code

The code defines a model based on the VGG16 architecture pretrained on ImageNet. The base VGG16 model is loaded with its top classification layers removed, and its weights are set to those trained on ImageNet. The layers of the base model are then frozen to prevent further training. A new Sequential model (model2) is created by adding custom classification layers on top of the frozen VGG16 base. These additional layers include a flattening layer, followed by three dense layers with 128, 64, and 32 units, respectively, all using ReLU activation functions. The final dense layer with a single unit and sigmoid activation is added for binary classification.

```python
# Load the VGG16 model (pretrained on ImageNet)
base_model = VGG16(include_top = False, weights = "imagenet", input_shape = (224, 224, 3), pooling = "max",
                   classes = 2)
base_model.summary()
# Freeze the base model layers
for layer in base_model.layers:
    layer.trainable = False

# Create a new model by adding custom classification layers on top of the base model
model2 = Sequential()
model2.add(base_model)
model2.add(Flatten())
model2.add(Dense(128, activation='relu'))
model2.add(Dense(64, activation='relu'))
model2.add(Dense(32, activation='relu'))
model2.add(Dense(1, activation='sigmoid'))

# Compile the model
model2.compile(optimizer='adam',
               loss='binary_crossentropy',
               metrics=['accuracy', tf.keras.metrics.Precision(), tf.keras.metrics.Recall(), tf.keras.metrics.F1Score()])
```

3. Experimental results and results discussion

All models will be run for the same period of 10, so that we can compare them in a correct and fair way. We will compare the accuracy and loss of the models by graphing and illustrating the results.

3.1. Evaluate metrics

3.1.1. YOLO metrics

Accuracy	67%
Loss	1.18

3.1.2. Resnet50 metrics

Accuracy	97.51%
Loss	1.9

3.1.3. inceptionV3 metrics

Accuracy	97%
Loss	0.04

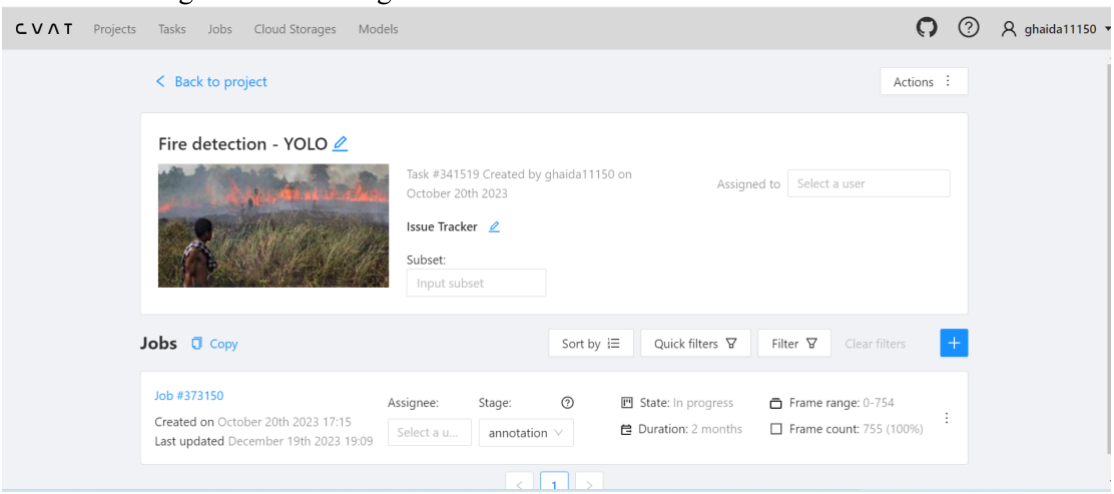
3.1.4. VGG16 metrics

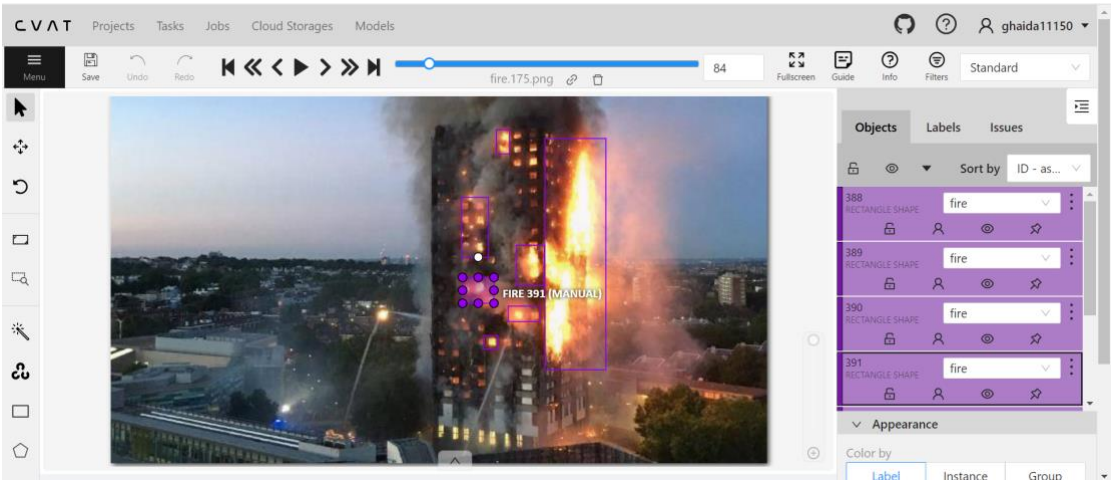
Accuracy	95.98%
Loss	0.10964

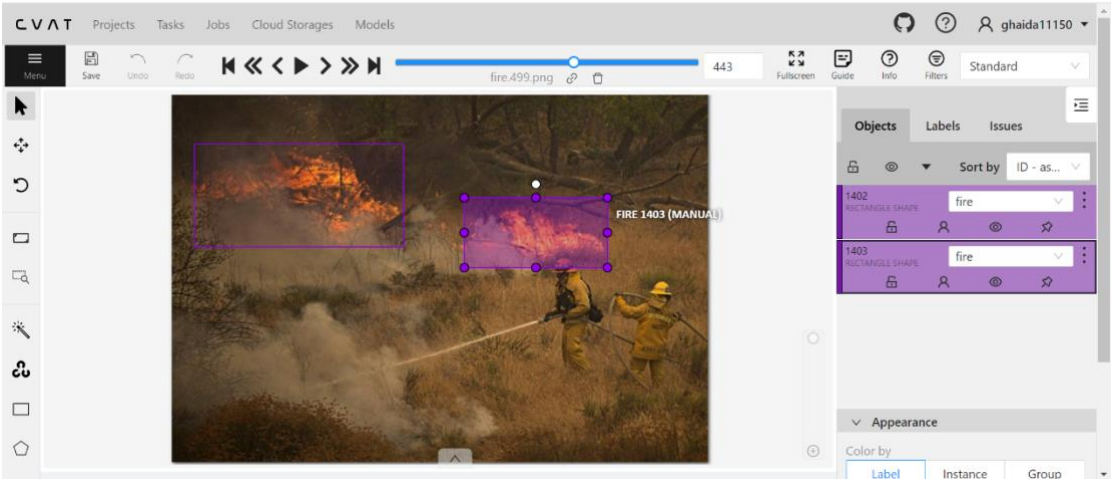
3.2. Experiments

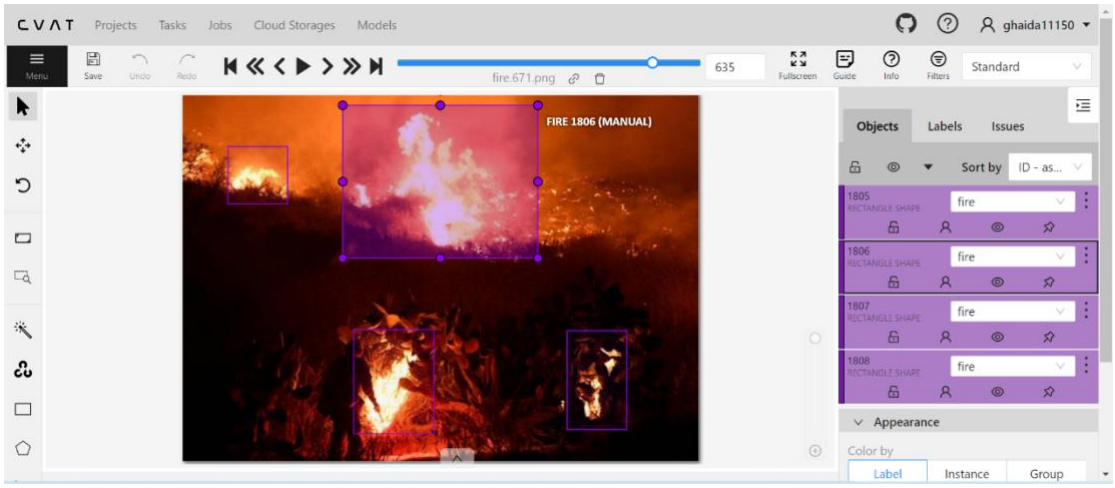
3.2.1. YOLO experiments

After obtaining the dataset from Kaggle, we labeled all the fires in the images without labeling the smoke using CVAT.

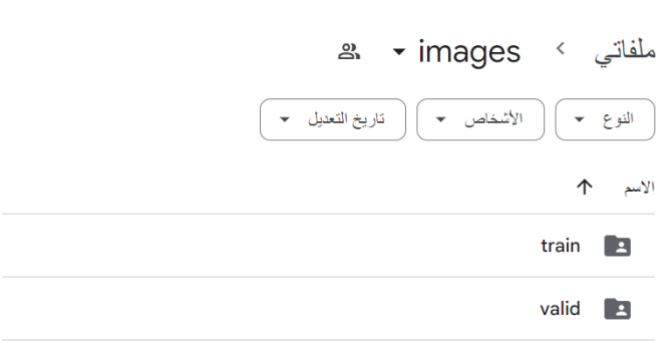




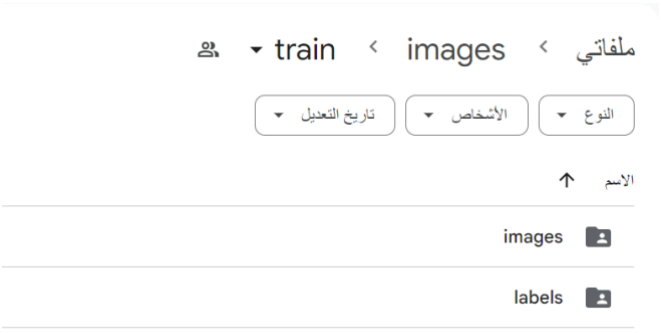




Afterwards, we divided the images into two files: one for training by 80% and one for validation by 20%.



In the training file, we placed two files: one containing the images and another containing the labels.



Afterwards, we took the file paths.

```yaml
# Train/val/test sets as 1) dir: path/to/imgs, 2) file: path/to/imgs.txt, or 3) list: [path/to/imgs1, path/to/imgs2, ..]
path: C:/Users/1riya/OneDrive/Desktop/AI - project  # dataset root dir
train: images/train  # train images (relative to 'path') 128 images
val: images/valid  # val images (relative to 'path') 128 images
test:  # test images (optional)

# Classes
names:
  0: fire
```

Then we proceeded to import the Ultralytics library from GitHub, and then we started training the model.

```python
# https://github.com/ultralytics/ultralytics
# pip install onnx
# pip install protobuf==3.20.*

from ultralytics import YOLO # pip install ultralytics

# Load a model
# model = YOLO("yolov8n.yaml")  # build a new model from scratch
model = YOLO("yolov8n.pt")  # load a pretrained model (recommended for training)

# YOLOv8n   640 37.3    80.4    0.99    3.2     8.7
# YOLOv8s   640 44.9    128.4   1.20    11.2    28.6
# YOLOv8m   640 50.2    234.7   1.83    25.9    78.9
# YOLOv8l   640 52.9    375.2   2.39    43.7    165.2
# YOLOv8x   640 53.9    479.1   3.53    68.2    257.8

# Use the model
model.train(data="trainfire.yaml", epochs=10)  # train the model
metrics = model.val()  # evaluate model performance on the validation set
# results = model("https://ultralytics.com/images/bus.jpg")  # predict on an image
# path = model.export(format="onnx")  # export the model to ONNX format
```

We noticed that increasing the number of epochs resulted in a higher accuracy rate but took a very long time to run. For example, it took over 4 hours to run 10 epochs, and the number of runs increased as the number of epochs increased.

After running for more than 4 hours, this was the result of the model's fire detection in the images. It didn't recognize all the images because it needed to be trained for a longer period than 10 epochs.

3.2.2. Resnet50 experiments

Transfer learning was applied using ImageNet weights, freezes layers up to the 39th layer to use the pre-trained model on a large dataset, and then fine-tuned it on our smaller dataset for the fire detection project, and then two extra layers were added to the model. The first added layer was a CNN dense layer with relu activation function since it is the best used for image processing problems, and the last layer was a CNN classification layer for the output, the softmax activation function was used. The trainable parameters of Resnet-50 model 24,357,377 out of 24,637,313 parameters.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.applications import ResNet50

# Load pre-trained ResNet50 with imagenet weights
pre_trained_resnet_50 = ResNet50(include_top=False, input_shape=(300, 300, 3), pooling='avg', weights='imagenet')

# Freeze layers up to the 39th layer
for layer in pre_trained_resnet_50.layers[:39]:
    layer.trainable = False

# Build the model
model = Sequential()
model.add(pre_trained_resnet_50)
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.summary()
```

```
=================================================================
Total params: 24637313 (93.98 MB)
Trainable params: 24357377 (92.92 MB)
Non-trainable params: 279936 (1.07 MB)
_________________________________________________________________
```

The dataset was taken and then split into train, and validate. Each set was pre-processed to have size of 300*300, and the data was distributed in 8 batches.

```
# Create training generator with a subset for training data

train_generator = datagen.flow_from_directory(
    train_data,               # Path to the directory containing the training data
    target_size=(300, 300),   # Resize images to 300x300 pixels
    batch_size=8,             # Number of images in each batch
    class_mode='binary',      # Type of classification task (binary in this case)
    subset='training'         # Generate data for the training subset
)

# Create validation generator with a subset for validation data

validation_generator = datagen.flow_from_directory(
    train_data,               # Path to the directory containing the training data
    target_size=(300, 300),   # Resize images to 300x300 pixels
    batch_size=8,             # Number of images in each batch
    class_mode='binary',      # Type of classification task (binary in this case)
    subset='validation'       # Generate data for the validation subset
)
```

Once the model was built and the data is ready, we started to train the model. The following Criterions were followed in training:

Hyperparameters	Value
Optimization Function	Adam
Batch Size	8
Epochs	10
Weights	ImageNet Dataset Weights
Activation Function	Sigmoid

3.2.3. inceptionV3 experiments

In the MobileNet experiment, we used a pre-trained Inceptionv3 model that was tuned to fit our problems. We loaded the pre-trained base model from the Keras library with ImageNet weights, and 150x150 input size. Then as illustrated in the previous section we added one global average pooling layer a Dropout layer of 0.2 ratio, a Dense layer with 512 value, then a adding a Dropout layer of 0.2 rate. Lastly, changing the activation function of the output layer to sigmoid since we have a binary classification in the dataset and use it as the classification layer. We compiled the model with Adam optimizer, binary_crossentropy as loss, and accuracy (acc) as metrics.

```
target_size= (150, 150)
train_generator = train_datagen.flow_from_directory(dataTrain, batch_size = 64,
                                                    class_mode = 'binary', target_size = target_size)

validation_generator = test_datagen.flow_from_directory( dataTest,  batch_size = 64,
                                                        class_mode = 'binary', target_size = target_size)
```

```
[ ]  from keras.models import Sequential
     from keras.layers import GlobalAveragePooling2D, Dense, Dropout, Flatten

     base_InceptionV3_model = InceptionV3(input_shape=[150, 150,3], include_top=False
                                          , weights="imagenet")
     InceptionV3_model= Sequential()
     InceptionV3_model.add(base_InceptionV3_model)
     InceptionV3_model.add(GlobalAveragePooling2D())
     InceptionV3_model.add(Dropout(0.2))
     InceptionV3_model.add(Dense(512))
     InceptionV3_model.add(Dropout(0.2))
     InceptionV3_model.add(Dense(1, activation= "sigmoid"))
     InceptionV3_model.compile(optimizer='adam', loss= 'binary_crossentropy',metrics=['acc'])
```

```
Trained_Model = InceptionV3_model.fit(train_generator, validation_data = validation_generator, verbose=1, epochs=10)
```

Hyperparameters	Value
Optimization Function	Adam
Batch Size	64
Epochs	10
Weights	ImageNet Dataset Weights
Activation Function	Sigmoid

3.2.4. VGG16 experiments

In the VGG16 experiment, we utilized a pre-trained VGG16 model with ImageNet weights, which was loaded from the Keras library. To tailor the model for our specific problem, we extended the architecture by appending a Dense layer with 128 units, followed by a Dense layer with 64 units, another Dense layer with 32 units, and a final Dense layer with 1 unit. Notably, we applied the rectified linear unit (ReLU) activation function to all Dense layers except the last one. Given the binary nature of our dataset, we modified the activation function of the output layer to sigmoid, serving as the classification layer. The model was compiled using the Adam optimizer, binary_crossentropy as the loss function, and accuracy, F1 score and Precision as evaluation metrics. Subsequently, the model underwent training for 10 epochs.

```python
# Load the VGG16 model (pretrained on ImageNet)
base_model = VGG16(include_top = False, weights = "imagenet", input_shape = (224, 224, 3), pooling = "max",
                   classes = 2)
base_model.summary()
# Freeze the base model layers
for layer in base_model.layers:
    layer.trainable = False

# Create a new model by adding custom classification layers on top of the base model
model2 = Sequential()
model2.add(base_model)
model2.add(Flatten())
model2.add(Dense(128, activation='relu'))
model2.add(Dense(64, activation='relu'))
model2.add(Dense(32, activation='relu'))
model2.add(Dense(1, activation='sigmoid'))

# Compile the model
model2.compile(optimizer='adam',
               loss='binary_crossentropy',
               metrics=['accuracy', tf.keras.metrics.Precision(), tf.keras.metrics.Recall(), tf.keras.metrics.F1Score()])
```

```python
# Set the test split ratio
test_split = 0.2

# Create the training dataset
train_dataset = tf.keras.preprocessing.image_dataset_from_directory(
    train_data_dir,
    validation_split=test_split,
    subset="training",
    seed=200,
    image_size=target_size,
    batch_size=batch_size,
    shuffle=True,
)

# Create the test dataset
test_dataset = tf.keras.preprocessing.image_dataset_from_directory(
    test_data_dir,
    validation_split=test_split,
    subset="validation",
    seed=44,
    image_size=target_size,
    batch_size=batch_size,
    shuffle=False,
```

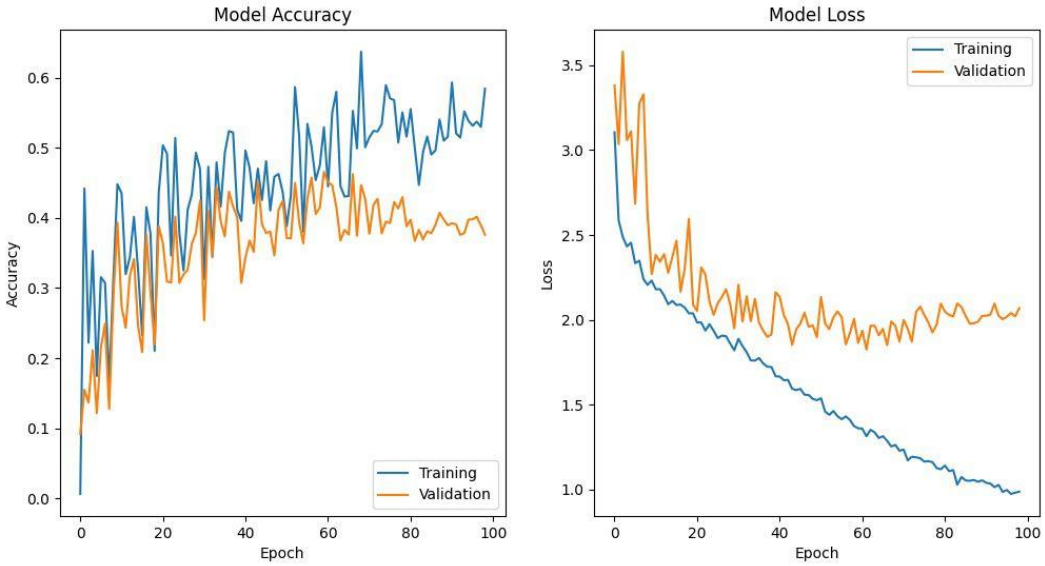
```python
# Define early stopping
early_stopping = EarlyStopping(
    patience = 3,
    restore_best_weights=True,  # Restore model weights from the epoch with the best value of the monitored quantity
)

# Train the model with early stopping
history = model2.fit(
    train_dataset,
    epochs=10,
    validation_data=train_dataset,
    callbacks=[early_stopping],  # Add the early stopping callback
```

Hyperparameters	Value
Optimization Function	Adam
Batch Size	32
Epochs	10
Weights	ImageNet Dataset Weights
Activation Function	Sigmoid

3.3. Results and results discussion

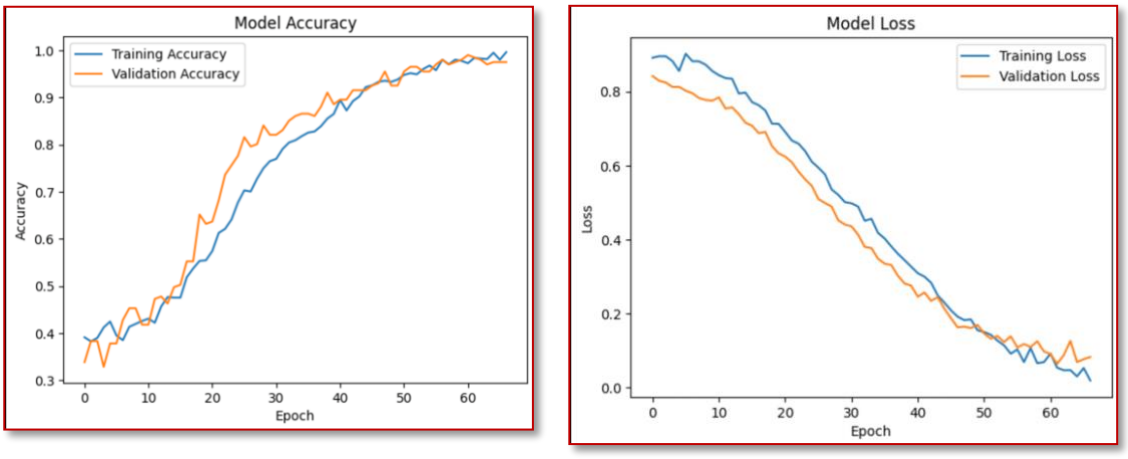
3.3.1. YOLO results

After training the model for 100 epochs, the test results showed fluctuating values. The test accuracy reached 58%, with the best accuracy recorded at 67% and a loss of 2.1%. The best loss value achieved was 1.18%. The following figure illustrates the loss and accuracy of both the training and testing results.



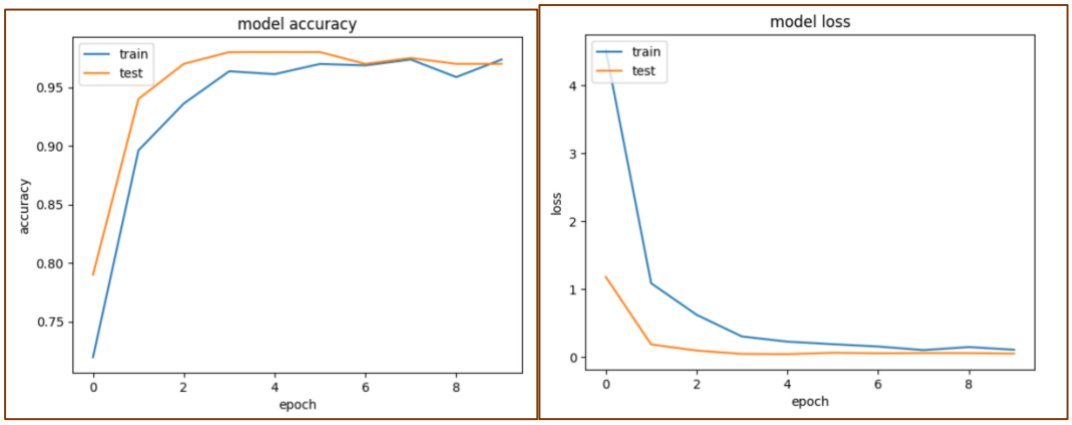
3.3.2. Resnet50 results

After training the model for 100 epochs and then testing it, the accuracy of testing reached 97% with loss of 4%. The following figure illustrates the loss and the accuracy of both training and test results.



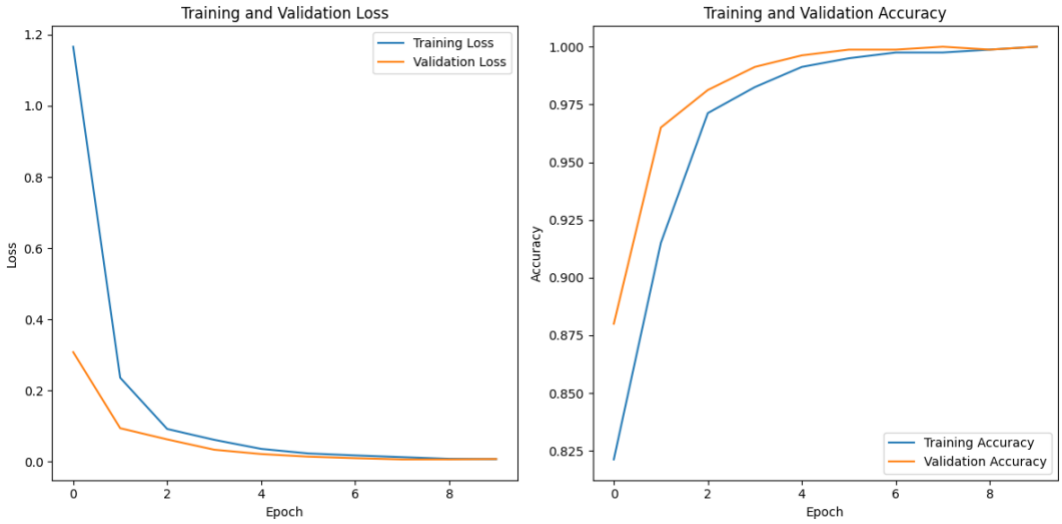
3.3.3. inceptionV3 results

After training and testing the model, the accuracy and loss of the model throughout the 10 epochs are illustrated in the below figure. After many trials to find the best hyperparameters, the batch size was set to 64 and the number of epochs to 10. As shown in the figure, the model's accuracy after 10 epochs was 97.37 % for the training set and 97 % for the testing set. For the loss, the training set loss decreased from around 4.52 to 0.1 whereas the testing set loss decreased from around 1.17 to 0.04.



3.3.4. VGG16 results

After training and testing the model, the accuracy and loss of the model are illustrated in the below figure. After many trials to find the best hyperparameters, the batch size was set to 32 and the number of epochs to 10. As shown in the figure, the model's accuracy after 10 epochs was 100 % for the training set and 95.98% for the testing set. For the loss, the training set loss was 0.00738 whereas the testing set loss was 0.10964.



4. Conclusion

In conclusion, the team used different methods to detect fire and non-fire in imaging dataset taken from Kaggle. Resnet-50 was able to yield an accuracy of 97.51%, while YOLO yielded an accuracy of 67%, InceptionV3 yielded an accuracy of 97 %, and VGG16 yielded an accuracy of approximately 95.98%. We learned how to apply deep learning techniques to real-world problems, despite each approach being different, each approach was unique in its solution to the key issue. The limitations and challenges faced were mostly computational issues with devices not being able to run a heavy deep learning model quickly. Overall, the result of Reset-50 model yield the best results compared to the other chosen models with accuracy of 97.51%.

5. References and bibliography

1. Cheng, L., Li, J., Duan, P., & Wang, M. (2021). A small attentional YOLO model for landslide detection from satellite remote sensing images. Landslides, 18(8), 2751-2765.
2. Jiang, P., Ergu, D., Liu, F., Cai, Y., & Ma, B. (2022). A Review of Yolo algorithm developments. Procedia Computer Science, 199, 1066-1073.
3. Fang, W., Wang, L., & Ren, P. (2019). Tinier-YOLO: A real-time object detection method for constrained environments. IEEE Access, 8, 1935-1944.
4. Wang, S. (2021, June). Research towards YOLO-series algorithms: Comparison and analysis of object detection models for real-time UAV applications. In Journal of physics: Conference series (Vol. 1948, No. 1, p. 012021). IOP Publishing
5. Chen H., Zhou, G., & Jiang, H. (2023). Student Behavior Detection in the Classroom Based on Improved YOLOv8. Sensors, 23(20), 8385
6. Terven, J., Córdova-Esparza, D. M., & Romero-González, J. A. (2023). A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS. Machine Learning and Knowledge Extraction, 5(4), 1680-1716.
7. Wisdomml. (2023, March 30). Understanding ReSNet-50 in depth: architecture, skip connections, and advantages over other networks -. *Wisdom ML*. https://wisdomml.in/understanding-resnet-50-in-depth-architecture-skip-connections-and-advantages-over-other-networks/
8. Davis, M. R., & Shekaramiz, M. (2023). Desert/Forest fire detection using Machine/Deep learning techniques. *Fire*, *6*(11), 418. https://doi.org/10.3390/fire6110418
9. He, K. (2016). *Deep residual learning for image recognition*. https://openaccess.thecvf.com/content_cvpr_2016/html/He_Deep_Residual_Learning_CVPR_2016_paper.html
10. Xiao, Z., Dong, E., & Du, S. (2020). Fire Detection Method Based on Deep Residual Network and Multi-Scale Feature Fusion. *IEEE*. https://doi.org/10.1109/cac51589.2020.9326871
11. GeeksforGeeks. (2023, January 10). *Residual Networks ResNet Deep Learning*. https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/
12. (n.d.). *An Improved Approach for Fire Detection using Deep Learning Models*. Sci-Hub. https://sci-hub.se/https://ieeexplore.ieee.org/abstract/document/9102697
13. Dua, M., Kumar, M., Singh Charan, G., & Sagar Ravi, P. (2020). *An Improved Approach for Fire Detection using Deep Learning Models. 2020 International Conference on Industry 4.0 Technology (I4Tech).*
14. (2021, December 10). *Fire Detection inImages Using FrameworkBased on Image Processing, Motion Detection and Convolutional Neural Network*. Www.Ijisae.org. https://ijisae.org/index.php/IJISAE/article/view/1455/657
15. *Early Detection of Forest Fire using Deep Learning*. (2020, November 16). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/9293722
16. *Bhavani, A., Iswarya, M., & Lokesh, J. (2022a). Real-time fire detection based on CNN and inception V3 algorithms*. *International Journal of Health Sciences (IJHS)*, 13513–13527. https://doi.org/10.53730/ijhs.v6ns2.8618
17. *Hindarto, D. (2023). Revolutionizing Automotive Parts Classification Using InceptionV3 Transfer Learning*. *Informatics Study Program, Faculty of Communication and Informatics Technology, Universitas Nasional,*. https://doi.org/10.35870/ijsecs.v3i3.1784

18. *Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the Inception Architecture for Computer Vision. Zbigniew Wojna University College London.* https://doi.org/10.1109/cvpr.2016.308
19. *Inception-v3 convolutional neural network - MATLAB inceptionv3*. (n.d.). https://www.mathworks.com/help/deeplearning/ref/inceptionv3.html#d126e135100
20. *Li, P., & Zhao, W. (2020). Image fire detection algorithms based on convolutional neural networks. Case Studies in Thermal Engineering, 19, 100625*. https://doi.org/10.1016/j.csite.2020.100625
21. *Biswas, A., Ghosh, S. K., & Ghosh, A. (2023, January 1). Early Fire Detection and Alert System using Modified Inception-v3 under Deep Learning Framework. Procedia Computer Science.* https://doi.org/10.1016/j.procs.2023.01.200
22. *Wu, H., Hu, Y., Wang, W., Mei, X., & Xian, J. (2022). Ship Fire Detection Based on an Improved YOLO Algorithm with a Lightweight Convolutional Neural Network Model. Sensors, 22(19), 7420–7420.* https://doi.org/10.3390/s22197420

7. Appendices

6.1. YOLO source code

https://drive.google.com/drive/folders/1tjfmzlDlmBjTAmiSqe3lUBDNE7hoFWv8?usp=drive_link

6.2. Resnet50 source code

https://colab.research.google.com/drive/11xEzUeCJiJ4ZTxcfg4Xji3YMaYVeu7Uf?usp=sharing

6.3. inceptionV3 source code

https://colab.research.google.com/drive/1jhm5k8h5ZeWa2BsBXct_G8RiBekOdwLE?usp=sharing

6.4. VGG16 source code

https://colab.research.google.com/drive/1dPFLBOn63O7R5Q_-HLCWweA-Z1XRDx2W?usp=sharing